



NeHe OpenGL Tutoriály

Aktuální verzi překladů můžete najít na
<http://nehe.opengl.cz/>

Michal Turek - Woq <woq (zavináč) email.cz>
29.02.2004

Předmluva

Tutoriály na mých stránkách mohou obsahovat chyby a neměly by být považovány za nejlepší zdroj pro učení programování v OpenGL. co uděláte s kódem je jen a jen na vás, já jsem se pouze pokoušel usnadnit nováčkům proces učení. Pokud máte o OpenGL opravdu vážný zájem, měli byste obětovat nějaké peníze a investovat např. do OpenGL Red Book (ISBN 0-201-46138-2) nebo OpenGL Blue Book (ISBN 0-201-46140-4). Jsem vlastníkem druhého vydání obou těchto knih a ačkoli mohou být pro nové OpenGL programátory složité na pochopení, jsou určitě nejlepšími knihami o OpenGL vůbec. Další knihou, kterou zmíním, je OpenGL Superbible, názory na ni se všech různí. Je důležité, abyste měli pevné základy programovacího jazyka, který hodláte používat. Ačkoli v tutoriálech komentuji i ne-OpenGL kód, sám se učím, a tudíž jsem ohl zvolit špatnou (ale fungující) nebo ne zrovna vhodnou cestu. Je pouze na vás, co si odnesete a následně budete aplikovat ve svých vlastních projektech. Zkuste si hrát s kódem, čtěte knihy, pokládejte otázky. Poté, co vstřebáte informace na mém webu, zkuste jít na mnohem profesionálnější místa, jako je například OpenGL.org. Navštivte také linky uvedené na mých stránkách. Každý z nich je neuvěřitelným přínosem celé OpenGL komunitě. Většinu z nich vede mnoho talentovaných lidí, kteří neznají pouze OpenGL. Mnohokrát programují mnohem lépe než já. Prosím, mějte toto všechno při brouzdání mým webem na paměti. Doufám, že využijete vše, co mohu nabídnout a také, že v nejbližší době uvidím výsledky vaší práce.

Jedna závěrečná poznámka... pokud naleznete kód, který je nápadně podobný kódu od jiného autora, prosím kontaktujte mě. ujistím vás, že jakýkoli kód, který jsem si půjčil nebo z kterého jsem se učil, pochází buď z MSDN nebo ze stránek vytvořených pro studium lidí podobným způsobem, jako mé stránky pomáhají s OpenGL. Nikdy jsem úmyslně kód nevzal a nikdy bych ho nezveřejnil bez náležitého creditu. Mohly nastat případy, kdy jsem kód získal z free webů a nevěděl, že ho vzali od někoho dalšího, takže pokud se tak stalo, prosím kontaktujte mě. Tento kód buď přepíši nebo úplně odstraním z programu. Většina kódu by měla být originální. Když jsem si něco půjčoval, tak to bylo pouze tehdy, když jsem neměl absolutně žádný nápad, jak nějakou věc vyřešit a i tehdy jsem se mu snažil před vložením do programu porozumět.

Pokud naleznete chybu v jakémkoli z tutoriálů, nezáleží na tom, jak malá může být, prosím dejte mi vědět. Základní kód tutoriálů byl napsán v roce 1997 a je stoprocentně původní. Poté podstoupil mnoho změn a vylepšení. Je více než jisté, že bude upravován i nadále. Pokud nebudu jediným, kdo ho bude modifikovat, osoba odpovědná za změny bude uvedena.

Jeff Molofee - NeHe

Obsah

Lekce 1 - Vytvoření OpenGL okna ve Windows

Napsal: Jeff Molofee - NeHe

Přeložil: Václav Slováček - Wessan

Naučíte se jak nastavit a vytvořit OpenGL okno ve Windows. Program, který vytvoříte zobrazí "pouze" prázdné okno. Černé pozadí nevypadá nic moc, ale pokud porozumíte této lekci, budete mít velmi dobrý základ pro jakoukoliv další práci. Zjistíte jak OpenGL pracuje, jak probíhá vytváření okna a také jak napsat jednoduše pochopitelný kód.

Lekce 2 - Vytváření trojúhelníků a čtyřúhelníků

Napsal: Jeff Molofee - NeHe

Přeložil: Václav Slováček - Wessan

Zdrojový kód z první lekce trochu upravíme, aby program vykreslil trojúhelník a čtverec. Víím, že si asi myslíte, že takovéto vykreslování je banalita, ale až začnete programovat pochopíte, že orientovat se ve 3D prostoru není na představivost až tak jednoduché. Jakékoli vytváření objektů v OpenGL závisí na trojúhelnících a čtvercích. Pokud pochopíte tuto lekci máte napůl vyhráno.

Lekce 3 - Barvy

Napsal: Jeff Molofee - NeHe

Přeložil: Milan Turek

S jednoduchým rozšířením znalostí ze druhé lekce budete moci používat barvy. Naučíte se jak ploché vybarvování, tak i barevné přechody. Barvy rozzáří vzhled aplikace a tím spíše zaujmou diváka.

Lekce 4 - Rotace

Napsal: Jeff Molofee - NeHe

Přeložil: Milan Turek

Naučíme se, jak otáčet objekt okolo os. Trojúhelník se bude otáčet kolem osy y a čtverec kolem osy x. Je jednoduché vytvořit scénu z polygonů. Přidání pohybu ji pěkně oživí.

Lekce 5 - Pevné objekty

Napsal: Jeff Molofee - NeHe

Přeložil: Milan Turek

Rozšířením poslední části vytvoříme skutečné 3D objekty. Narozdíl od 2D objektů ve 3D prostoru. Změníme trojúhelník na pyramidu a čtverec na krychli. Pyramida bude vybarvena barevným přechodem a každou stěnu krychle vybarvíme jinou barvou.

Lekce 6 - Textury

Napsal: Jeff Molofee - NeHe

Přeložil: Milan Turek

Namapujeme bitmapový obrázek na krychli. Použijeme zdrojové kódy z první lekce, protože je jednodušší (a přehlednější) začít s prázdným oknem než složitě upravovat předchozí lekci.

Lekce 7 - Texturové filtry, osvětlení, ovládání pomocí klávesnice

Napsal: Jeff Molofee - NeHe

Přeložil: Jiří Rajský - RAJSOFT junior

V tomto díle se pokusím vysvětlit použití tří odlišných texturových filtrů. Dále pak pohybu objektů pomocí klávesnice a nakonec aplikaci jednoduchých světel v OpenGL. Nebude se jako obvykle navazovat na kód z předchozího dílu, ale začne se pěkně od začátku.

Lekce 8 - Blending

Napsal: Tom Stanis

Přeložil: Jiří Rajský - RAJSOFT junior

Další typ speciálního efektu v OpenGL je blending, neboli průhlednost. Kombinace pixelů je určena alfa hodnotou barvy a použitou funkcí. Nabývá-li alfa 0.0f, materiál zprůhlední, hodnota 1.0f přináší pravý opak.

Lekce 9 - Pohyb bitmap ve 3D prostoru

Napsal: Jeff Molofee - NeHe

Přeložil: Milan Turek

Tento tutoriál vás naučí pohyb objektů ve 3D prostoru a kreslení bitmap bez černých míst, zakrývajících objekty za nimi. Jednoduchou animaci a rozšířené použití blendingu. Teď byste už měli rozumět OpenGL velmi dobře. Naučili jste se vše od nastavení OpenGL okna, po mapování textur za použití světel a blendingu. To byl první tutoriál pro středně pokročilé. A pokračujeme dále...

Lekce 10 - Vytvoření 3D světa a pohyb v něm

Napsal: Lionel Brits - ßetelgeuse

Přeložil: Jiří Rajský - RAJSOFT junior & Michal Turek - Woq

Do současnosti jsme programovali otáčející se kostku nebo pár hvězd. Máte (měli byste mít :-)) základní pojem o 3D. Ale rotující krychle asi nejsou to nejlepší k tvorbě dobrých deathmatchových protivníků! Nečekejte a začněte s Quakem IV ještě dnes! Tyto dny potřebujete k velkému, komplikovanému a dynamickému 3D světu s pohybem do všech směrů, skvělými efekty zrcadel, portálů, deformacemi a třeba také vysokým frameratem. Tato lekce vám vysvětlí základní strukturu 3D světa a pohybu v něm.

Lekce 11 - Efekt vlnící se vlajky

Napsal: Bosco

Přeložil: Michal Turek - Woq

Naučíme se jak pomocí sinusové funkce animovat obrázky. Pokud znáte standardní šetřič Windows "Létající 3D objekty" (i on by měl být programovaný v OpenGL), tak budeme dělat něco podobného.

Lekce 12 - Display list

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Chcete vědět, jak urychlit vaše programy v OpenGL? Jste unaveni z nesmyslného opisování již napsaného kódu? Nejde to nějak jednodušeji? Nešlo by například jedním příkazem vykreslit otexturovanou krychli? Samozřejmě, že jde. Tento tutoriál je určený speciálně pro vás. Předvytvořené objekty a jejich vykreslování jedním řádkem kódu. Jak snadné...

Lekce 13 - Bitmapové fonty

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Často kladená otázka týkající se OpenGL zní: "Jak zobrazit text?". Vždycky jde namapovat texturu textu. Bohužel nad ním máte velmi malou kontrolu. A pokud nejste dobří v blendigu, většinou skončíte smixováním s ostatními obrázky. Pokud byste chtěli znát lehčí cestu k výstupu textu na jakékoli místo s libovolnou barvou nebo fontem, potom je tato lekce určitě pro vás. Bitmapové fonty jsou 2D písma, které nemohou být rotovány. Vždy je uvidíte zepředu.

Lekce 14 - Outline fonty

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Bitmapové fonty nestačí? Potřebujete kontrolovat pozici textu i na ose z? Chtěli byste fonty s hloubkou? Pokud zní vaše odpověď ano, pak jsou 3D fonty nejlepší řešení. Můžete s nimi pohybovat na ose z a tím měnit jejich velikost, otáčet je, prostě dělat vše, co nemůžete s obyčejnými. Jsou nejlepší volbou ke hrám a demům.

Lekce 15 - Mapování textur na fonty

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Po vysvětlení bitmapových a 3D fontů v předchozích dvou lekcích jsem se rozhodl napsat lekci o mapování textur na fonty. Jedná se o tzv. automatické generování koordinátů textur. Po dočtení této lekce budete umět namapovat texturu opravdu na cokoli - zcela snadno a jednoduše.

Lekce 16 - Mlha

Napsal: Christopher Aliotta - Precursor

Přeložil: Michal Turek - Woq

Tato lekce rozšiřuje použitím mlhy lekci 7. Naučíte se používat tři různých filtrů, měnit barvu a nastavit oblast působení mlhy (v hloubce). Velmi jednoduchý a "efektní" efekt.

Lekce 17 - 2D fonty z textur

Napsal: Giuseppe D'Agata & Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

V této lekci se naučíte, jak vykreslit font pomocí texturou omapovaného obdélníku. Dozvíte se také, jak používat pixely místo jednotek. I když nemáte rádi mapování 2D znaků, najdete zde spoustu nových informací o OpenGL.

Lekce 18 - Quadratic

Napsal: GB Schmick - TipTup

Přeložil: Michal Turek - Woq

Představuje se vám báječný svět quadraticů. Jedním řádkem kódu snadno vytváříte komplexní objekty typu koule, disku, válce ap. Pomocí matematiky a trochy plánování lze snadno morphovat jeden do druhého.

Lekce 19 - Časticové systémy

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Chtěli jste už někdy naprogramovat exploze, vodní fontány, planoucí hvězdy a jiné skvělé efekty, nicméně kódování časticových systémů bylo buď příliš těžké nebo jste vůbec nevěděli, jak na to? V této lekci zjistíte, jak vytvořit jednoduchý, ale dobře vypadající časticový systém. Extra přidáme duhové barvy a ovládání klávesnicí. Také se dozvíte, jak pomocí triangle stripu jednoduše vykreslovat velké množství trojúhelníků.

Lekce 20 - Maskování

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Černé okraje obrázků jsme dosud ořezávali blendingem. Ačkoli je tato metoda efektivní, ne vždy transparentní objekty vypadají dobře. Modelová situace: vytváříme hru a potřebujeme celistvý text nebo zakřivený ovládací panel, ale při blendingu scéna prosvítá. Nejlepším řešením je maskování obrázků.

Lekce 21 - Přímký, antialiasing, časování, pravouhlá projekce, základní zvuky a jednoduchá herní logika

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

První opravdu rozsáhlý tutoriál - jak už plyne z gigantického názvu. Doufejme, že taková spousta informací a technik dokáže udělat šťastným opravdu každého. Strávil jsem dva dny kódováním a kolem dvou týdnů psaním tohoto HTML souboru. Pokud jste někdy hráli hru Admiar, lekce vás vrátí do vzpomínek. Úkol hry sestává z vyplnění jednotlivých políček mřížky. Samozřejmě se musíte vyhýbat všem nepřítelům.

Lekce 22 - Bump Mapping & Multi Texturing

Napsal: Jens Schneider

Přeložil: Václav Slováček - Wessan

Pravý čas vrátit se zpátky na začátek a začít si opakovat. Nováčkům v OpenGL se absolutně nedoporučuje! Pokud, ale máte odvalu, můžete zkusit dobrodružství s nadupanou grafikou. V této lekci modifikujeme kód z šesté lekce, aby podporoval hardwarový multi texturing přes opravdu skvělý vizuální efekt nazvaný bump mapping.

Lekce 23 - Mapování textur na kulové kvadratiky

Napsal: GB Schmick - TipTup

Přeložil: Milan Turek

Tento tutoriál je napsán na bázi **lekce 18**. V **lekci 15** (Mapování textur na fonty) jsem psal o automatickém mapování textur. Vysvětlil jsem jak můžeme poprosit OpenGL o automatické generování texturových koordinát, ale protože lekce 15 byla celkem skromná, rozhodl jsem se přidat mnohem více detailů o této technice.

Lekce 24 - Výpis OpenGL rozšíření, ořezávací testy a textury z TGA obrázků

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

V této lekci se naučíte, jak zjistit, která OpenGL rozšíření (extensions) podporuje vaše grafická karta. Vypíšeme je do středu okna, se kterým budeme moci po stisku šipek rolovat. Použijeme klasický 2D texturový font s tím rozdílem, že texturu vytvoříme z TGA obrázku. Jeho největšími přednostmi jsou jednoduchá práce a podpora alfa kanálu. Odbouráním bitmap už nebudeme muset inkudovat knihovnu glaux.

Lekce 25 - Morfování objektů a jejich nahrávání z textového souboru

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

V této lekci se naučíte, jak nahrát souřadnice vrcholů z textového souboru a plynulou transformaci z jednoho objektu na druhý. Nezaměříme se ani tak na grafický výstup jako spíše na efekty a potřebnou matematiku okolo. Kód může být velice jednoduše modifikován k vykreslování linkami nebo polygony.

Lekce 26 - Odrazy a jejich ořezávání za použití stencil bufferu

Napsal: Banu Cosmin - Choko

Přeložil: Milan Turek & Michal Turek - Woq

Tutoriál demonstruje extrémně realistické odrazy za použití stencil bufferu a jejich ořezávání, aby "nevystoupily" ze zrcadla. Je mnohem více pokrokový než předchozí lekce, takže před začátkem čtení doporučuji menší opakování. Odrazy objektů nebudou vidět nad zrcadlem nebo na druhé straně zdi a budou mít barevný nádech zrcadla - skutečné odrazy.

Lekce 27 - Stíny

Napsal: Banu Cosmin - Choko & Brett Porter

Přeložil: Michal Turek - Woq

Představuje se vám velmi komplexní tutoriál na vrhání stínů. Efekt je doslova neuvěřitelný. Stíny se roztahují, ohýbají a zahalují i ostatní objekty ve scéně. Realisticky se pokrouť na stěnách nebo podlaze. Se vším lze pomoci klávesnicí pohybovat ve 3D prostoru. Pokud ještě nejste se stencil bufferem a matematikou jako jedna rodina, nemáte nejmenší šanci.

Lekce 28 - Bezierovy křivky a povrchy, fullscreen fix

Napsal: David Nikdel

Přeložil: Michal Turek - Woq

David Nikdel je osoba stojící za tímto skvělým tutoriálem, ve kterém se naučíte, jak se vytvářejí Bezierovy křivky. Díky nim lze velice jednoduše zakřivit povrch a provádět jeho plynulou animaci pouhou modifikací několika kontrolních bodů. Aby byl výsledný povrch modelu ještě zajímavější, je na něj namapována textura. Tutoriál také eliminuje problémy s fullscreenem, kdy se po návratu do systému neobnovilo původní rozlišení obrazovky.

Lekce 29 - Blitter, nahrávání .RAW textur

Napsal: Andreas Löffler & Rob Fletcher

Přeložil: Václav Slováček - Wessan & Michal Turek - Woq

V této lekci se naučíte, jak se nahrávají .RAW obrázky a konvertují se do textur. Dozvíte se také o blitteru, grafické metodě přenášení dat, která umožňuje modifikovat textury poté, co už byly nahrány do programu. Můžete jím zkopírovat část jedné textury do druhé, blendingem je smíchat dohromady a také roztahovat. Maličko upravíme program tak, aby v době, kdy není aktivní, vůbec nezatěžoval procesor.

Lekce 30 - Detekce kolizí

Napsal: Dimitrios Christopoulos

Přeložil: Michal Turek - Woq

Na podobný tutoriál jste už jistě netrpělivě čekali. Naučíte se základy o detekcích kolizí, jak na ně reagovat a na fyzice založené modelovací efekty (nárazy, působení gravitace ap.). Tutoriál se více zaměřuje na obecnou funkci kolizí než zdrojovým kódům. Nicméně důležité části kódu jsou také popsány. Neočekávejte, že po prvním přečtení úplně všemu z kolizí porozumíte. Je to komplexní námět, se kterým vám pomohu začít.

Lekce 31 - Nahrávání a renderování modelů

Napsal: Brett Porter

Přeložil: Michal Turek - Woq

Další skvělý tutoriál! Naučíte se, jak nahrát a zobrazit otexturovaný Milkshape3D model. Nezdá se to, ale asi nejvíce se budou hodit znalosti o práci s dynamickou pamětí a jejím kopírování z jednoho místa na druhé.

Lekce 32 - Picking, alfa blending, alfa testing, sorting

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

V tomto tutoriálu se pokusím zodpovědět několik otázek, na které jsem denně dotazován. Chcete vědět, jak při kliknutí tlačítkem myši identifikovat OpenGL objekt nacházející se pod kurzorem (picking). Dále byste se chtěli dozvědět, jak vykreslit objekt bez zobrazení určité barvy (alfa blending, alfa testing). Třetí věcí, se kterou si nevíte rady, je, jak řadit objekty, aby se při blendingu správně zobrazily (sorting). Naprogramujeme hru, na které si vše vysvětlíme.

Lekce 33 - Nahrávání komprimovaných i nekomprimovaných obrázků TGA

Napsal: Evan Piphio - Terminate

Přeložil: Michal Turek - Woq

V lekci 24 jsem vám ukázal cestu, jak nahrávat nekomprimované 24/32 bitové TGA obrázky. Jsou velmi užitečné, když potřebujete alfa kanál, ale nesmíte se starat o jejich velikost, protože byste je ihned přestali používat. K diskovému místu nejsou zrovna šetrné. Problém velikosti vyřeší nahrávání obrázků komprimovaných metodou RLE. Kód pro loading a hlavičkové soubory jsou odděleny od hlavního projektu, aby mohly být snadno použity i jinde.

Lekce 34 - Generování terénů a krajin za použití výškového mapování textur

Napsal: Ben Humphrey - DigiBen

Přeložil: Michal Turek - Woq

Chtěli byste vytvořit věrnou simulaci krajiny, ale nevíte, jak na to? Bude nám stačit obyčejný 2D obrázek ve stupních šedi, pomocí kterého deformujeme rovinu do třetího rozměru. Na první pohled těžko řešitelné problémy bývají častokrát velice jednoduché.

Lekce 35 - Přehrávání videa ve formátu AVI

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Přehrávání AVI videa v OpenGL? Na pozadí, povrchu krychle, koule, či válce, ve fullscreenu nebo v obyčejném okně. Co víc si přát...

Lekce 36 - Radial Blur, renderování do textury

Napsal: Dario Corno - rlo

Přeložil: Michal Turek - Woq

Společnými silami vytvoříme extrémně působivý efekt radial blur, který nevyžaduje žádná OpenGL rozšíření a funguje na jakémkoli hardwaru. Naučíte se také, jak lze na pozadí aplikace vyrenderovat scénu do textury, aby pozorovatel nic neviděl.

Lekce 37 - Cel-Shading

Napsal: Sami Hamlaoui - MENTAL

Přeložil: Václav Slováček - Wessan & Michal Turek - Woq

Cel-Shading je druh vykreslování, při kterém výsledné modely vypadají jako ručně kreslené karikatury z komiksů (cartoons). Rozličné efekty mohou být dosaženy miniaturní modifikací zdrojového kódu. Cel-Shading je velmi úspěšným druhem renderingu, který dokáže kompletně změnit duch hry. Ne ale vždy... musí se umět a použít s rozmyslem.

Lekce 38 - Nahrávání textur z resource souboru & texturování trojúhelníků

Napsal: Jeff Molofee - NeHe

Přeložil: Václav Slováček - Wessan

Tento tutoriál jsem napsal pro všechny z vás, kteří se mě v emailech dotazovali na to "Jak mám loadovat texturu ze zdrojů programu, abych měl všechny obrázky uložené ve výsledném .exe souboru?" a také pro ty, kteří psali "Vím, jak otexturovat obdélník, ale jak mapovat na trojúhelník?" Tutoriál není, oproti jiným, extrémně pokrokový, ale když nic jiného, tak se naučíte, jak skrýt vaše precizní textury před okem uživatele. A co víc - budete moci trochu ztížit jejich krádeži :-)

Lekce 39 - Úvod do fyzikálních simulací

Napsal: Erkin Tunca

Přeložil: Václav Slováček - Wessan

V gravitačním poli se pokusíme rozpohybovat hmotný bod s konstantní rychlostí, hmotný bod připojený k pružině a hmotný bod, na který působí gravitační síla - vše podle fyzikálních zákonů. Kód je založen na nejnovějším NeHeGL kódu.

Lekce 40 - Fyzikální simulace lana

Napsal: Erkin Tunca

Přeložil: Michal Turek - Woq

Přichází druhá část dvoudílné série o fyzikálních simulacích. Základy už známe, a proto se pustíme do komplikovanějšího úkolu - klávesnicí ovládat pohyby simulovaného lana. Zatáhneme-li za horní konec, prostřední část se rozhoupe a spodek se vláčí po zemi. Skvělý efekt.

Lekce 41 - Volumetrická mlha a nahrávání obrázků pomocí IPicture

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

V tomto tutoriálu se naučíte, jak pomocí rozšíření EXT_fog_coord vytvořit volumetrickou mlhu. Také zjistíte, jak pracuje IPicture kód a jak ho můžete využít pro nahrávání obrázků ve svých vlastních projektech. Demo sice není až tak komplexní jako některá jiná, nicméně i přesto vypadá hodně efektně.

Lekce 42 - Více viewportů

Napsal: Jeff Molofee - NeHe

Přeložil: Michal Turek - Woq

Tento tutoriál byl napsán pro všechny z vás, kteří se chtěli dozvědět, jak do jednoho okna zobrazit více pohledů na jednu scénu, kdy v každém probíhá jiný efekt. Jako bonus přidám získávání velikosti OpenGL okna a velice rychlý způsob aktualizace textury bez jejího znovuvytváření.

Lekce 43 - FreeType Fonty v OpenGL

Napsal: Sven Olsen

Přeložil: Pavel Hradský a Michal Turek - Woq

Použitím knihovny FreeType Font rendering library můžete snadno vypisovat vyhlazené znaky, které vypadají mnohem lépe než písmena u bitmapových fontů z lekce 13. Náš text bude mít ale i jiné výhody - bezproblémová rotace, dobrá spolupráce s OpenGL výběracími (picking) funkcemi a víceřádkové řetězce.

Lekce 44 - Čočkové efekty

Napsal: Vic Hollis

Přeložil: Michal Turek - Woq

Čočkové efekty vznikají po dopadu paprsku světla např. na objektiv kamery nebo fotoaparátu. Podíváte-li se na záři vyvolanou čočkou, zjistíte, že jednotlivé útvary mají jednu společnou věc. Pozorovateli se zdá, jako by se všechny pohybovaly skrz střed scény. S tímto na mysli můžeme osu z jednoduše odstranit a vytvářet vše ve 2D. Jediný problém související s nepřítomností z souřadnice je, jak zjistit, jestli se zdroj světla nachází ve výhledu kamery nebo ne. Připravte se proto na trochu matematiky.

Lekce 45 - Vertex Buffer Object (VBO)

Napsal: Paul Frazee

Přeložil: Michal Turek - Woq

Jeden z největších problémů jakékoli 3D aplikace je zajištění její rychlosti. Vždy byste měli limitovat množství aktuálně renderovaných polygonů buď řazením, cullingem nebo nějakým algoritmem na snižování detailů. Když nic z toho nepomáhá, můžete zkusit například vertex arrays. Moderní grafické karty nabízejí rozšíření nazvané vertex buffer object, které pracuje podobně jako vertex arrays kromě toho, že nahrává data do vysoce výkonné paměti grafické karty, a tak podstatně snižuje čas potřebný pro rendering. Samozřejmě ne všechny karty tato nová rozšíření podporují, takže musíme implementovat i verzi založenou na vertex arrays.

Lekce 46 - Fullscreenový antialiasing

Napsal: Colt McAnlis - MainRoach

Přeložil: Michal Turek - Woq

Chtěli byste, aby vaše aplikace vypadaly ještě lépe než doposud? Fullscreenové vyhlazování, nazývané též multisampling, by vám mohlo pomoci. S výhodou ho používají ne-realtimové renderovací programy, nicméně s dnešním hardwarem ho můžeme dosáhnout i v reálném čase. Bohužel je implementováno pouze jako rozšíření

ARB_MULTISAMPLE, které nebude pracovat, pokud ho grafická karta nepodporuje.

Lekce 47 - CG vertex shader

Napsal: Owen Bourne

Přeložil: Michal Turek - Woq

Používání vertex a fragment (pixel) shaderů ke "špinavé práci" při renderingu může mít nespočet výhod. Nejvíce je vidět např. pohyb objektů do teď výhradně závislý na CPU, který neběží na CPU, ale na GPU. Pro psaní velice kvalitních shaderů poskytuje CG (přiměřeně) snadné rozhraní. Tento tutoriál vám ukáže jednoduchý vertex shader, který sice něco dělá, ale nebude předvádět ne nezbytné osvětlení a podobné složitější nadstavby. Tak jako tak je především určen pro začátečníky, kteří už mají nějaké zkušenosti s OpenGL a zajímají se o CG.

Lekce 48 - ArcBall rotace

Napsal: Terence J. Grant

Přeložil: Pavel Hradský a Michal Turek - Woq

Nebylo by skvělé otáčet modelem pomocí myši jednoduchým drag & drop? S ArcBall rotacemi je to možné. Moje implementace je založená na myšlenkách Brettona Wadea a Kena Shoemakea. Kód také obsahuje funkci pro rendering toroidu - kompletně i s normálami.

Lekce 1 - Vytvoření OpenGL okna ve Windows

Naučíte se jak nastavit a vytvořit OpenGL okno ve Windows. Program, který vytvoříte zobrazí "pouze" prázdné okno. Černé pozadí nevypadá nic moc, ale pokud porozumíte této lekci, budete mít velmi dobrý základ pro jakoukoliv další práci. Zjistíte jak OpenGL pracuje, jak probíhá vytváření okna a také jak napsat jednoduše pochopitelný kód.

Jsem obyčejný kluk s vášní pro OpenGL. Když jsem o něm poprvé slyšel, vydalo 3Dfx zrychlené ovladače pro Voodoo 1. Hned jsem věděl, že OpenGL je něco, co se musím naučit. Bohužel bylo velice těžké najít nějaké informace, jak v knihách, tak na internetu. Strávil jsem hodiny pokusy o napsání funkčního kódu a přesvědčováním lidí emaily a na IRC. Zjistil jsem, že lidé, kteří rozuměli OpenGL, se považovali za elitu a nechtěli se o své vědomosti dělit. Velice frustrující... Vytvořil jsem tyto tutoriály, aby je zájemci o OpenGL mohli použít, když budou potřebovat pomoc. V každém tutoriálu se vše snažím vysvětlit do detailů, aby bylo jasné, co každý řádek dělá. Snažím se svůj kód psát co nejjednodušeji (nepoužívám MFC)! I absolutní nováček, jak v C++, tak v OpenGL, by měl být schopen tento kód zvládnout a mít další dobré nápady, co dělat dál. Je mnoho tutoriálů o OpenGL. Pokud jste hardcorový OpenGL programátor asi Vám budou připadat příliš jednoduché, ale pokud právě začínáte mají mnoho co nabídnout!

Začnu tento tutoriál přímo kódem. První, co se musí udělat, je vytvořit projekt. Pokud nevíte jak to udělat, neměli byste se učit OpenGL, ale Visual C++. Některé verze Visual C++ vyžadují, aby byl bool změněn na BOOL, true na TRUE a false na FALSE. Pokud to budete mít na paměti neměly by být s kompilací žádné problémy. Potom co vytvoříte novou Win32 Application (NE console application) ve Visual C++, budete potřebovat připojit OpenGL knihovny. Jsou dvě možnosti, jak to udělat: Vyberte Project>Settings, pak zvolte záložku Link a do kolonky Object/Library Modules napište na začátek řádku (před kernel32.lib) OpenGL32.lib Glu32.lib Glaux.lib. Potom klikněte na OK. Nebo napište přímo do kódu programu následující řádky.

```
// Vložení knihoven
#pragma comment (lib,"opengl32.lib")
#pragma comment (lib,"glu32.lib")
#pragma comment (lib,"glaux.lib")
```

Nyní jste připraveni napsat svůj první OpenGL program pro Windows. Začneme vložením hlavičkových souborů.

```
#include <windows.h> // Hlavičkový soubor pro Windows
#include <gl\gl.h> // Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h> // Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h> // Hlavičkový soubor pro Glaux knihovnu
```

Dále potřebujete deklarovat globální proměnné, které chcete v programu použít. Tento program vytváří prázdné OpenGL okno, proto jich nebudeme potřebovat mnoho. Ty, které nyní použijeme jsou ovšem velmi důležité a budete je používat v každém programu založeném na tomto kódu. Nastavíme Rendering Context. Každý OpenGL program je spojen s Rendering Contextem. Rendering Context říká, která spojení volá OpenGL, aby se spojilo s Device Context (kontext zařízení). Nám stačí vědět, že OpenGL Rendering Context je definován jako hRC. Aby program mohl kreslit do okna potřebujete vytvořit Device Context. Ve Windows je Device Context definován jako hDC. Device Context napojí okno na GDI (grafické rozhraní). Proměnná hWnd obsahuje handle přidělený oknu a čtvrtý řádek vytvoří instanci programu.

```
HDC hDC = NULL; // Privátní GDI Device Context
HGLRC hRC = NULL; // Trvalý Rendering Context
HWND hWnd = NULL; // Obsahuje Handle našeho okna
HINSTANCE hInstance; // Obsahuje instanci aplikace
```

První řádek deklaruje pole, které budeme používat na sledování stisknutých kláves. Je mnoho způsobů, jak to udělat, ale takto to dělám já. Je to spolehlivé a můžeme sledovat stisk více kláves najednou. Proměnná active bude použita, aby náš program informovala, zda je jeho okno minimalizováno nebo ne. Když je okno minimalizováno můžeme udělat cokoli od pozastavení činnosti kódu až po opuštění programu. Já použiji pozastavení běhu programu. Díky tomu zbytečně nepoběží na pozadí, když bude minimalizován. Proměnná fullscreen bude obsahovat informaci, jestli náš program běží přes celou obrazovku - v tom případě bude fullscreen mít hodnotu true, když program poběží v okně bude mít hodnotu false. Je důležité, aby proměnná byla globální a tím pádem každá funkce věděla, jestli program běží ve fullscreenu, nebo v okně.

```
bool keys[256]; // Pole pro ukládání vstupu z klávesnice
bool active = TRUE; // Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE; // Ponese informaci o tom, zda je program ve fullscreenu

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); // Deklarace procedury okna
(funkční prototyp)
```

Následující funkce se volá vždy, když uživatel mění velikost okna. I když nejste schopni změnit velikost okna (například ve fullscreenu), bude tato funkce volána alespoň jednou, aby nastavila perspektivní pohled při spuštění programu. Velikost OpenGL scény se bude měnit v závislosti na šířce a výšce okna, ve kterém je zobrazena.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Změna velikosti a inicializace
OpenGL okna
{
    if (height==0) // Zabezpečení proti dělení nulou
    {
        height=1; // Nastaví výšku na jedna
    }

    glViewport(0,0,width,height); // Resetuje aktuální nastavení
}
```

Nastavíme obraz na perspektivní pohled. To znamená, že vzdálenější objekty budou menší. `glMatrixMode(GL_PROJECTION)` ovlivní formu obrazu. Forma obrazu určuje, jak výrazná bude perspektiva. Vytvoříme realisticky vypadající scénu. `glLoadIdentity()` resetuje matici. Vráti ji do jejího původního stavu. Po `glLoadIdentity()` nastavíme perspektivní pohled scény. Perspektiva je vypočítána s úhlem pohledu 45 stupňů a je založena na výšce a šířce okna. Číslo 0.1f je počáteční a 100.0f konečný bod, který říká jak hluboko do obrazovky můžeme kreslit. `glMatrixMode(GL_MODELVIEW)` označuje, že forma pohledu bude znovu změněna. Nakonec znovu resetujeme matici. Pokud předcházejícímu textu nerozumíte, nic si z toho nedělejte, vysvětlím ho celý v dalších tutoriálech. Jediné co nyní musíte vědět je, že následující řádky musíte do svého programu napsat, pokud chcete, aby scéna vypadala pěkně.

```
glMatrixMode(GL_PROJECTION); // Zvolí projekční matici
glLoadIdentity(); // Reset matice
gluPerspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 100.0f); // Výpočet
perspektivy

glMatrixMode(GL_MODELVIEW); // Zvolí matici Modelview
glLoadIdentity(); // Reset matice
}
```

Nastavíme vše potřebné pro OpenGL. Definujeme černé pozadí, zapneme depth buffer, aktivujeme smooth shading (vyhlazené stínování), atd.. Tato funkce se volá po vytvoření okna. Vrací hodnotu, ale tím se nyní nemusíme zabývat, protože naše inicializace není zatím úplně komplexní.

```
int InitGL(GLvoid) // Všechno nastavení OpenGL
{
```

Následující řádek povolí jemné stínování, aby se barvy na polygonech pěkně promíchaly. Více detailů o smooth shading si povíme v jiných tutoriálech.

```
glShadeModel(GL_SMOOTH); // Povolí jemné stínování
```

Nastavíme barvu pozadí prázdné obrazovky. Rozsah barev se určuje ve stupnici od 0.0f do 1.0f. 0.0f je nejtmaší a 1.0f je nejsvětlejší. První parametr ve funkci `glClearColor()` je intenzita červené barvy, druhý zelené a třetí modré. Čím bližší je hodnota barvy 1.0f, tím světlejší složka barvy bude. Poslední parametr je hodnota alpha (průhlednost). Když budeme čistit obrazovku, tak se o průhlednost starat nemusíme. Nyní ji necháme na 0.0f. Můžete vytvářet různé barvy kombinováním světlosti tří základních barev (červené, zelené, modré). Pokud budete mít `glClearColor(0.0f,0.0f,1.0f,0.0f)`, bude obrazovka modrá. Když budete mít `glClearColor(0.5f,0.0f,0.0f,0.0f)`, bude obrazovka středně tmavě červená. Abyste udělali bílé pozadí nastavte všechny hodnoty na nejvyšší hodnotu (1.0f), pro černé pozadí zadejte pro všechny složky 0.0f.

```
glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
```

Následující tři řádky ovlivňují depth buffer. Depth buffer si můžete představit jako vrstvy/hladiny obrazovky. Obsahuje informace, o tom jak hluboko jsou zobrazované objekty. Tento program sice nebude deep buffer používat (nic nevykresluje). Objekty se seřadí tak, aby bližší překrývaly vzdálenější.

```
glClearDepth(1.0f); // Nastavení hloubkového bufferu
glEnable(GL_DEPTH_TEST); // Povolí hloubkové testování
glDepthFunc(GL_LEQUAL); // Typ hloubkového testování
```

Dále oznámíme, že chceme použít nejlepší korekce perspektivy. Jen nepatrně se sníží výkon, ale zlepší se vzhled celé scény

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce
```

Nakonec vrátíme true. Když budeme chtít zjistit zda inicializace proběhla bez problémů, můžeme zkontrolovat, zda funkce vrátila hodnotu true nebo false. Můžete přidat vlastní kód, který vrátí false, když se inicializace nezdaří - např. loading textur. Nyní se tím nebudeme dále zabývat.

```
return TRUE; // Inicializace proběhla v pořádku
```

```
}
```

Do této funkce umístíme všechno vykreslování. Následující tutoriály budou přepisovat především tento a inicializační kód této lekce. (Pokud již nyní rozumíte základům OpenGL, můžete si zde připsat kreslení základních tvarů (mezi `glLoadIdentity()` a `return`). Pokud jste nováček, tak počkejte do dalšího tutoriálu. Jediné co nyní uděláme, je vymazání obrazovky na barvu, pro kterou jste se rozhodli, vymažeme obsah hloubkového bufferu a resetujeme scénu. Zatím nebudeme nic kreslit. Příkaz `return true` nám říká, že při kreslení nenastaly žádné problémy. Pokud z nějakého důvodu chcete přerušit běh programu, stačí přidat `return false` před `return true` - to říká našemu programu, že kreslení scény selhalo a program se ukončí.

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    // Sem můžete kreslit

    return TRUE; // Vykreslení proběhlo v pořádku
}
```

Následující část kódu je volána těsně před koncem programu. Úkolem funkce `KillGLWindow()` je uvolnění renderovacího kontextu, kontextu zařízení a handle okna. Přidal jsem zde nezbytné kontrolování chyb. Když není program schopen něco uvolnit, oznámí chybu, která říká co selhalo. Usnadní složité hledání problémů.

```
GLvoid KillGLWindow(GLvoid) // Zavírání okna
{
```

Zjistíme zda je program ve fullscreenu. Pokud ano, tak ho přepneme zpět do systému. Mohli bychom vypnout okno před opuštěním fullscreenu, ale na některých grafických kartách tím způsobíme problémy a systém by se mohl zhroutit.

```
    if (fullscreen) // Jsme ve fullscreenu?
    {
```

K návratu do původního nastavení systému používáme funkci `ChangeDisplaySettings(NULL,0)`. Jako první parametr zadáme `NULL` a jako druhý `0` - použijeme hodnoty uložené v registrech Windows (původní rozlišení, barevnou hloubku, obnovovací frekvenci, atd.). Po přepnutí zviditelníme kurzor.

```
        ChangeDisplaySettings(NULL,0); // Přepnutí do systému
        ShowCursor(TRUE); // Zobrazí kurzor myši
    }
```

Zkontrolujeme zda máme renderovací kontext (`hRC`). Když ne, program přeskočí část kódu pod ním, který kontroluje, zda máme kontext zařízení.

```
    if (hRC) // Máme rendering kontext?
    {
```

Zjistíme, zda můžeme odpojit `hRC` od `hDC`. Všimněte si, jak kontroluji chyby. Nejdříve programu řeknu, ať odpojí Rendering Context (s použitím `wglMakeCurrent(NULL,NULL)`), pak zkontroluji zda akce byla úspěšná. Takto dám více řádku do jednoho.

```
        if (!wglMakeCurrent(NULL,NULL)) // Jsme schopni oddělit kontexty?
        {
```

Pokud nejsme schopni uvolnit `DC` a `RC`, použijeme zobrazíme zprávu, že `DC` a `RC` nelze uvolnit. `NULL` v parametru znamená, že informační okno nemá žádného rodiče. Text ihned za `NULL` je text, který se vypíše do zprávy. Další parametr definuje text lišty. Parametr `MB_OK` znamená, že chceme mít na chybové zprávě jen jedno tlačítko s nápisem `OK`. `MB_ICONINFORMATION` zobrazí ikonu.

```
            MessageBox(NULL, "Release Of DC And RC Failed.", "SHUTDOWN ERROR", MB_OK |
            MB_ICONINFORMATION);
        }
```

Zkusíme vymazat Rendering Context. Pokud se pokus nezdaří, opět se zobrazí chybová zpráva. Nakonec nastavíme `hRC` a `NULL`.

```
        if (!wglDeleteContext(hRC)) // Jsme schopni smazat RC?
        {
            MessageBox(NULL, "Release Rendering Context Failed.", "SHUTDOWN ERROR", MB_OK |
            MB_ICONINFORMATION);
        }
        hRC=NULL; // Nastaví hRC na NULL
```

```
}
```

Zjistíme zda má program kontext zařízení. Když ano odpojíme ho. Pokud se odpojení nezdaří, zobrazí se chybová zpráva a hDC bude nastaven na NULL.

```
if (hDC && !ReleaseDC(hWnd,hDC))// Jsme schopni uvolnit DC
{
    MessageBox(NULL,"Release Device Context Failed.,"SHUTDOWN ERROR",MB_OK |
    MB_ICONINFORMATION);
    hDC=NULL;// Nastaví hDC na NULL
}
```

Nyní zjistíme zda máme handle okna a pokud ano pokusíme se odstranit okno použitím funkce DestroyWindow(hWnd). Pokud se pokus nezdaří, zobrazí se chybová zpráva a hWnd bude nastaveno na NULL.

```
if (hWnd && !DestroyWindow(hWnd))// Jsme schopni odstranit okno?
{
    MessageBox(NULL,"Could Not Release hWnd.,"SHUTDOWN ERROR",MB_OK |
    MB_ICONINFORMATION);
    hWnd=NULL;// Nastaví hWnd na NULL
}
```

Odregistrováním třídy okna oficiálně uzavřeme okno a předejdeme zobrazení chybové zprávy "Windows Class already registered" při opětovném spuštění programu.

```
if (!UnregisterClass("OpenGL",hInstance))// Jsme schopni odregistrovat třídu okna?
{
    MessageBox(NULL,"Could Not Unregister Class.,"SHUTDOWN ERROR",MB_OK |
    MB_ICONINFORMATION);
    hInstance=NULL;// Nastaví hInstance na NULL
}
}
```

Další část kódu má na starosti vytvoření OpenGL okna. Strávil jsem mnoho času přemýšlením zda mám udělat pouze fullscreen mód, který by vyžadoval méně kódu, nebo jednoduše upravitelnou, uživatelsky příjemnou verzi s variantou jak pro okno tak pro fullscreen, která však vyžaduje mnohem více kódu. Rozhodl jsem se pro druhou variantu, protože jsem dostával mnoho dotazů jako například: Jak mohu vytvořit okno místo fullscreenu? Jak změním popisek okna? Jak změním rozlišení ne formát pixelů? Následující kód dovede všechno. Jak si můžete všimnout funkce vrátí bool a přijímá 5 parametrů v pořadí: název okna, šířku okna, výšku okna, barevnou hloubku, fullscreen (pokud je parametr true program poběží ve fullscreenu, pokud bude false program poběží v okně). Vracíme bool, abychom věděli zda bylo okno úspěšně vytvořeno.

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
```

Za chvíli požádáme Windows, aby pro nás našel pixel format, který odpovídá tomu, který chceme. Toto číslo uložíme do proměnné PixelFormat.

```
GLuint PixelFormat;// Ukládá formát pixelů
```

Wc bude použijeme k uchování informací o struktuře Windows Class. Změnou hodnot jednotlivých položek, lze ovlivnit vzhled a chování okna. Před vytvořením samotného okna se musí zaregistrovat nějaká struktura pro okno.

```
WNDCLASS wc;// Struktura Windows Class
```

DwExStyle a dwStyle ponese informaci o normálních a rozšířených informacích o oknu. Použijí proměnné k uchování stylů, takže mohou měnit vzhled okna, který potřebuji vytvořit (pro fullscreen bez okraje a pro okno okraj).

```
DWORD dwExStyle;// Rozšířený styl okna
DWORD dwStyle;// Styl okna
```

Zjistíme polohu levého horního a pravého dolního rohu okna. Tyto proměnné využijeme k tomu, abychom nakreslili okno v takovém rozlišení, v jakém si ho přejeme mít. Pokud vytvoříme okno s rozlišením 640x480, okraje budou zabírat část našeho rozlišení.

```
RECT WindowRect;// Obdélník okna

WindowRect.left = (long)0;// Nastaví levý okraj na nulu
WindowRect.right = (long)width;// Nastaví pravý okraj na zadanou hodnotu
WindowRect.top = (long)0;// Nastaví horní okraj na nulu
WindowRect.bottom = (long)height;// Nastaví spodní okraj na zadanou hodnotu
```

Přidáme globální proměnné fullscreen, hodnotu fullscreenflag. Takže pokud naše okno poběží ve fullscreenu,

proměnná fullscreen se bude rovnat true. Kdybychom zavírali okno ve fullscreenu, ale hodnota proměnné fullscreen by byla false místo true, jak by měla být, počítač by se nepřepnul zpět do systému, protože by si myslel, že v něm již je. Jednoduše shrnuto, fullscreen vždy musí obsahovat správnou hodnotu.

```
fullscreen = fullscreenflag; // Nastaví proměnnou fullscreen na správnou hodnotu
```

Získáme instanci pro okno a poté definujeme Window Class. CS_HREDRAW a CS_VREDRAW donutí naše okno, aby se překreslovalo, kdykoliv se změní jeho velikost. CS_OWNDC vytvoří privátní kontext zařízení. To znamená, že není sdílen s ostatními aplikacemi. WndProc je procedura okna, která sleduje příchozí zprávy pro program. Žádná extra data pro okno nepoužíváme, takže do dalších dvou položek přiřadíme nulu. Nastavíme instanci a hIcon na NULL, což znamená, že nebudeme pro náš program používat žádnou speciální ikonu a pro kurzor myši používáme standardní šipku. Barva pozadí nás nemusí zajímat (to zařídíme v OpenGL). Nechceme, aby okno mělo menu, takže i tuto hodnotu nastavíme na NULL. Jméno třídy může být libovolné. Já použiji pro jednoduchost "OpenGL".

```
hInstance = GetModuleHandle(NULL); // Získá instanci okna

wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // Překreslení při změně velikosti a vlastní DC
wc.lpfnWndProc = (WNDPROC) WndProc; // Definuje proceduru okna
wc.cbClsExtra = 0; // Žádná extra data
wc.cbWndExtra = 0; // Žádná extra data
wc.hInstance = hInstance; // Instance
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // Standardní ikona
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // Standardní kurzor myši
wc.hbrBackground = NULL; // Pozadí není nutné
wc.lpszMenuName = NULL; // Nechceme menu
wc.lpszClassName = "OpenGL"; // Jméno třídy okna
```

Zaregistrujeme právě definovanou třídu okna. Když nastane chyba a zobrazí se chybové hlášení. Zmáčknutím tlačítka OK se program ukončí.

```
if (!RegisterClass(&wc)) // Registruje třídu okna
{
    MessageBox(NULL, "Failed To Register The Window
    Class.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
    return FALSE; // Při chybě vrátí false
}
```

Nyní si zjistíme zda má program běžet ve fullscreenu, nebo v okně.

```
if (fullscreen) // Budeme ve fullscreenu?
{
```

S přepínáním do fullscreenu, mívají lidé mnoho problémů. Je zde pár důležitých věcí, na které si musíte dávat pozor. Ujistěte se, že šířka a výška, kterou používáte ve fullscreenu je totožná s tou, kterou chcete použít v okně. Další věc je hodně důležitá. Musíte přepnout do fullscreenu předtím než vytvoříte okno. V tomto kódu se o rovnost výšky a šířky nemusíte starat, protože velikost ve fullscreenu i v okně budou stejné.

```
DEVMODE dmScreenSettings; // Mód zařízení

memset(&dmScreenSettings, 0, sizeof(dmScreenSettings)); // Vynulování paměti

dmScreenSettings.dmSize = sizeof(dmScreenSettings); // Velikost struktury Devmode
dmScreenSettings.dmPelsWidth = width; // Šířka okna
dmScreenSettings.dmPelsHeight = height; // Výška okna
dmScreenSettings.dmBitsPerPel = bits; // Barevná hloubka
dmScreenSettings.dmFields = DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;
```

Funkce ChangeDisplaySettings() se pokusí přepnout do módu, který je uložen v dmScreenSettings. Použiji parametr CDS_FULLSCREEN, protože odstraní pracovní lištu ve spodní části obrazovky a nepřesune nebo nezmění velikost okna při přepínání z fullscreenu do systému nebo naopak.

```
// Pokusí se použít právě definované nastavení
if (ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL)
{
```

Pokud právě vytvořený fullscreen mód neexistuje, zobrazí se chybová zpráva s nabídkou spuštění v okně nebo opuštění programu.

```
// Nejde-li fullscreen, může uživatel spustit program v okně nebo ho opustit
if (MessageBox(NULL, "The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use Windowed Mode Instead?", "NeHe
```

```
GL", MB_YESNO|MB_ICONEXCLAMATION) == IDYES)
{
```

Když se uživatel rozhodne pro běh v okně, do proměnné fullscreen se přiřadí false a program pokračuje dále.

```
    fullscreen=FALSE; // Běh v okně
}
else
{
```

Pokud se uživatel rozhodl pro ukončení programu, zobrazí se uživateli zpráva, že program bude ukončen. Bude vrácena hodnota false, která našemu programu říká, že pokus o vytvoření okna nebyl úspěšný a potom se program ukončí.

```
    // Zobrazí uživateli zprávu, že program bude ukončen
    MessageBox(NULL, "Program Will Now Close.", "ERROR", MB_OK|MB_ICONSTOP);

    return FALSE; // Vrátí FALSE
}
}
}
```

Protože pokus o přepnutí do fullscreenu může selhat, nebo se uživatel může rozhodnout pro běh programu v okně, zkontrolujeme ještě jednou zda je proměnná fullscreen true nebo false. Až poté nastavíme typ obrazu.

```
if (fullscreen) // Jsme stále ve fullscreenu?
{
```

Pokud jsme stále ve fullscreenu nastavíme rozšířený styl na WS_EX_APPWINDOW, což donutí okno, aby překrylo pracovní lištu. Styl okna určíme na WS_POPUP. Tento typ okna nemá žádné okraje, což je pro fullscreen výhodné. Nakonec vypneme kurzor myši. Pokud váš program není interaktivní, je většinou vhodnější ve fullscreenu kurzor vypnout. Pro co rozhodnete je na vás.

```
    dwExStyle=WS_EX_APPWINDOW; // Rozšířený styl okna
    dwStyle=WS_POPUP; // Styl okna
    ShowCursor(FALSE); // Skryje kurzor
}
else
{
```

Pokud místo fullscreenu používáme běh v okně, nastavíme rozšířený styl na WS_EX_WINDOWEDGE. To dodá oknu trochu 3D vzhledu. Styl nastavíme na WS_OVERLAPPEDWINDOW místo na WS_POPUP. WS_OVERLAPPEDWINDOW vytvoří okno s lištou, okraji, tlačítky pro minimalizaci a maximalizaci. Budeme moci měnit velikost.

```
    dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // Rozšířený styl okna
    dwStyle=WS_OVERLAPPEDWINDOW; // Styl okna
}
}
```

Přizpůsobíme okno podle stylu, který jsme vytvořili. Přizpůsobení udělá okno v takovém rozlišení, jaké požadujeme. Normálně by okraje překrývaly část okna. S použitím příkazu AdjustWindowRectEx žádná část OpenGL scény nebude překryta okraji, místo toho bude okno uděláno o málo větší, aby se do něj vešly všechny pixely tvořící okraj okna. Ve fullscreenu tato funkce nemá žádný efekt.

```
AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle); // Přizpůsobení velikosti
okna
```

Vytvoříme okno a zkontrolujeme zda bylo vytvořeno správně. Použijeme funkci CreateWindowEx() se všemi parametry, které vyžaduje. Rozšířený styl, který jsme se rozhodli použít. Jméno třídy (musí být stejné jako to, které jste použili, když jste registrovali Window Class). Titulek okna. Styl okna. Horní levá pozice okna (0,0 je nejjistější). Šířka a výška okna. Nechceme mít rodičovské okno ani menu, takže nastavíme tyto parametry na NULL. Zadáme instanci okna a konečně přiřadíme NULL na místo posledního parametru. Všimněte si, že zahrnujeme styly WS_CLIPSIBLINGS a WS_CLIPCHILDREN do stylu, který jsme se rozhodli použít. WS_CLIPSIBLINGS a WS_CLIPCHILDREN jsou potřebné pro OpenGL, aby pracovalo správně. Tyto styly zakazují ostatním oknům, aby kreslily do našeho okna.

```
// Vytvoření okna
if (!(hWnd=CreateWindowEx(dwExStyle, // Rozšířený styl
    "OpenGL", // Jméno třídy
    title, // Titulek
    dwStyle | // Definovaný styl
    WS_CLIPSIBLINGS | // Požadovaný styl
    WS_CLIPCHILDREN, // Požadovaný styl
    0, 0, // Pozice
```

```

WindowRect.right-WindowRect.left, // Výpočet šířky
WindowRect.bottom-WindowRect.top, // Výpočet výšky
NULL, // Žádné rodičovské okno
NULL, // Bez menu
hInstance, // Instance
NULL)) // Nepředat nic do WM_CREATE

```

Dále zkontrolujeme zda bylo vytvořeno. Pokud bylo, hWnd obsahuje handle tohoto okna. Když se vytvoření okna nepovede, kód zobrazí chybovou zprávu a program se ukončí.

```

{
    KillGLWindow(); // Zruší okno
    MessageBox(NULL, "Window Creation Error.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
    return FALSE; // Vrátí chybu
}

```

Vybereme Pixel Format, který podporuje OpenGL, dále zvolíme double buffering a RGBA (červená, zelená, modrá, průhlednost). Pokusíme se najít formát, který odpovídá tomu, pro který jsme se rozhodli (16 bitů, 24 bitů, 32 bitů). Nakonec nastavíme Z-Buffer. Ostatní parametry se nepoužívají nebo pro nás nejsou důležité.

```

static PIXELFORMATDESCRIPTOR pfd // Oznámíme Windows jak chceme vše nastavit
{
    sizeof(PIXELFORMATDESCRIPTOR), // Velikost struktury
    1, // Číslo verze
    PFD_DRAW_TO_WINDOW | // Podpora okna
    PFD_SUPPORT_OPENGL | // Podpora OpenGL
    PFD_DOUBLEBUFFER, // Podpora Double Bufferingu
    PFD_TYPE_RGBA, // RGBA Formát
    bits, // Zvolí barevnou hloubku
    0, 0, 0, 0, 0, 0, // Bitů barev ignorovány
    0, // Žádný alpha buffer
    0, // Ignorován Shift bit
    0, // Žádný akumulární buffer
    0, 0, 0, 0, // Akumulární bitů ignorovány
    16, // 16-bitový hloubkový buffer (Z-Buffer)
    0, // Žádný Stencil Buffer
    0, // Žádný Auxiliary Buffer
    PFD_MAIN_PLANE, // Hlavní vykreslovací vrstva
    0, // Rezervováno
    0, 0, 0 // Maska vrstvy ignorována
};

```

Pokud nenastaly problémy během vytváření okna, pokusíme se připojit kontext zařízení. Pokud ho se nepřipojí, zobrazí se chybové hlášení a program se ukončí.

```

if (!(hDC=GetDC(hWnd))) // Podařilo se připojit kontext zařízení?
{
    KillGLWindow(); // Zavře okno
    MessageBox(NULL, "Can't Create A GL Device
Context.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
    return FALSE; // Ukončí program
}

```

Když získáme kontext zařízení, pokusíme se najít odpovídající Pixel Format. Když ho Windows nenajde formát, zobrazí se chybová zpráva a program se ukončí.

```

if (!(PixelFormat=ChoosePixelFormat(hDC, &pfd))) // Podařilo se najít Pixel Format?
{
    KillGLWindow(); // Zavře okno
    MessageBox(NULL, "Can't Find A Suitable
PixelFormat.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
    return FALSE; // Ukončí program
}

```

Když Windows najde odpovídající formát, tak se ho pokusíme nastavit. Pokud při pokusu o nastavení nastane chyba, opět se zobrazí chybové hlášení a program se ukončí.

```

if(!SetPixelFormat(hDC, PixelFormat, &pfd)) // Podařilo se nastavit Pixel Format?
{
    KillGLWindow(); // Zavře okno
    MessageBox(NULL, "Can't Set The PixelFormat.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
}

```



```

        return FALSE;// Ukončí program
    }

```

Pokud byl nastaven Pixel Format správně, pokusíme se získat Rendering Context. Pokud ho nezískáme, program zobrazí chybovou zprávu a ukončí se.

```

    if (!(hRC=wglCreateContext(hDC))// Podařilo se vytvořit Rendering Context?
    {
        KillGLWindow();// Zavře okno
        MessageBox(NULL,"Can't Create A GL Rendering
        Context. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE;// Ukončí program
    }

```

Pokud nenastaly žádné chyby při vytváření jak Device Context, tak Rendering Context, vše co musíme nyní udělat je aktivovat Rendering Context. Pokud ho nebudeme moci aktivovat, zobrazí se chybová zpráva a program se ukončí.

```

    if(!wglMakeCurrent(hDC,hRC))// Podařilo se aktivovat Rendering Context?
    {
        KillGLWindow();// Zavře okno
        MessageBox(NULL,"Can't Activate The GL Rendering
        Context. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE;// Ukončí program
    }

```

Pokud bylo okno vytvořeno, zobrazíme ho na obrazovce, nastavíme ho, aby bylo v popředí (vyšší priorita) a pak nastavíme zaměření na toto okno. Zavoláme funkci ResizeGLScene() s parametry odpovídajícími výšce a šířce okna, abychom správně nastavili perspektivu OpenGL.

```

    ShowWindow(hWnd,SW_SHOW);// Zobrazení okna
    SetForegroundWindow(hWnd);// Do popředí
    SetFocus(hWnd);// Zaměří fokus
    ReSizeGLScene(width, height);// Nastavení perspektivy OpenGL scény

```

Konečně se dostáváme k volání výše definované funkce InitGL(), ve které nastavujeme osvětlení, loading textur a cokoli jiného, co je potřeba. Můžete vytvořit svou vlastní kontrolu chyb ve funkci InitGL() a vracet true, když vše proběhne bez problémů, nebo false, pokud nastanou nějaké problémy. Například, nastane-li chyba při nahrávání textur, vrátíte false, jako znamení, že něco selhalo a program se ukončí.

```

    if (!InitGL())// Inicializace okna
    {
        KillGLWindow();// Zavře okno
        MessageBox(NULL,"Initialization Failed. ","ERROR",MB_OK|MB_ICONEXCLAMATION);
        return FALSE;// Ukončí program
    }

```

Pokud jsme se dostali až takhle daleko, můžeme konstatovat, že vytvoření okna proběhlo bez problémů. Vrátime true do WinMain(), což říká, že nenastaly žádné chyby. To zabrání programu, aby se sám ukončil.

```

        return TRUE;// Vše proběhlo v pořádku
    }

```

Nyní se vypořádáme se systémovými zprávami pro okno. Když máme zaregistrovanou naši Window Class, můžeme podstoupit k části kódu, která má na starosti zpracování zpráv.

```

LRESULT CALLBACK WndProc(HWND hWnd, // Handle okna
    UINT uMsg, // Zpráva pro okno
    WPARAM wParam, // Doplnkové informace
    LPARAM lParam) // Doplnkové informace
{

```

Napišeme mapu zpráv. Program se bude větvit podle proměnné uMsg, která obsahuje jméno zprávy.

```

    switch (uMsg)// Větvení podle příchozí zprávy
    {

```

Po příchodu WM_ACTIVE, zkontrolujeme, zda je okno stále aktivní. Pokud bylo minimalizováno, nastavíme hodnotu active na false. Pokud je naše okno aktivní, proměnná active bude mít hodnotu true.

```

        case WM_ACTIVATE:// Změna aktivity okna
        {
            if (!HIWORD(wParam))// Zkontroluje zda není minimalizované
            {

```

```

        active=TRUE;// Program je aktivní
    }
    else
    {
        active=FALSE;// Program není aktivní
    }

    return 0;// Návrat do hlavního cyklu programu
}

```

Po příchodu WM_SYSCOMMAND (systémový příkaz) porovnáme wParam s možnými stavy, které mohly nastat. Když je wParam WM_SCREENSAVE nebo SC_MONITORPOWER snaží se systém zapnout spořič obrazovky, nebo přejít do úsporného režimu. Jestliže vrátíme 0 zabráníme systému, aby tyto akce provedl.

```

case WM_SYSCOMMAND:// Systémový příkaz
{
    switch (wParam)// Typ systémového příkazu
    {
        case SC_SCREENSAVE:// Pokus o zapnutí šetřiče obrazovky
        case SC_MONITORPOWER:// Pokus o přechod do úsporného režimu?
            return 0;// Zabrání obojímu
        }

    break;// Návrat do hlavního cyklu programu
}

```

Přišlo-li WM_CLOSE bylo okno zavřeno. Pošleme tedy zprávu pro opuštění programu, která přeruší vykonávání hlavního cyklu. Proměnnou done (ve WinMain()) nastavíme na true, hlavní smyčka se přeruší a program se ukončí.

```

case WM_CLOSE:// Povel k ukončení programu
{
    PostQuitMessage(0);// Pošle zprávu o ukončení
    return 0;// Návrat do hlavního cyklu programu
}

```

Pokud byla stisknuta klávesa, můžeme zjistit, která z nich to byla, když zjistíme hodnotu wParam. Potom zadáme do buňky, specifikované wParam, v poli keys[] true. Díky tomu potom můžeme zjistit, která klávesa je právě stisknutá. Tímto způsobem lze zkontrolovat stisk více kláves najednou.

```

case WM_KEYDOWN:// Stisk klávesy
{
    keys[wParam] = TRUE;// Oznámí to programu
    return 0;// Návrat do hlavního cyklu programu
}

```

Pokud byla naopak klávesa uvolněna uložíme do buňky s indexem wParam v poli keys[] hodnotu false. Tímto způsobem můžeme zjistit zda je klávesa ještě stále stisknuta nebo již byla uvolněna. Každá klávesa je reprezentována jedním číslem od 0 do 255. Když například stisknu klávesu číslo 40, hodnota key[40] bude true, jakmile ji pustím její hodnota se vrátí opět na false.

```

case WM_KEYUP:// Uvolnění klávesy
{
    keys[wParam] = FALSE;// Oznámí to programu
    return 0;// Návrat do hlavního cyklu programu
}

```

Kdykoliv uživatel změní velikost okna, pošle se WM_SIZE. Přečteme LOWORD a HIWORD hodnoty lParam, abychom zjistili jaká je nová šířka a výška okna. Předáme tyto hodnoty do funkce ReSizeGLScene(). Perspektiva OpenGL scény se změní podle nových rozměrů.

```

case WM_SIZE:// Změna velikosti okna
{
    ReSizeGLScene(LOWORD(lParam),HIWORD(lParam)); // LoWord=Šířka, HiWord=Výška
    return 0;// Návrat do hlavního cyklu programu
}
}

```

Zprávy, o které se nestaráme, budou předány funkci DefWindowProc(), takže se s nimi vypořádá systém.

```

return DefWindowProc(hWnd, uMsg, wParam, lParam);// Předání ostatních zpráv systému
}

```

Funkce WinMain() je vstupní bod do aplikace, místo, odkud budeme volat funkce na otevření okna, snímání zpráv a interakci s uživatelem.

```
int WINAPI WinMain(HINSTANCE hInstance, // Instance
    HINSTANCE hPrevInstance, // Předchozí instance
    LPSTR lpCmdLine, // Parametry příkazové řádky
    int nCmdShow) // Stav zobrazení okna
{
```

Deklarujeme dvě lokální proměnné. Msg bude použita na zjišťování, zda se mají zpracovávat nějaké zprávy. Proměnná done bude mít na počátku hodnotu false. To znamená, že náš program ještě nemá být ukončen. Dokud se done rovná false, program poběží. Jakmile se změní z false na true, program se ukončí.

```
    MSG msg; // Struktura zpráv systému
    BOOL done=FALSE; // Proměnná pro ukončení programu
```

Další část kódu je volitelná. Zobrazuje zprávu, která se zeptá uživatele, zda chce spustit program ve fullscreenu. Pokud uživatel vybere možnost Ne, hodnota proměnné fullscreen se změní z výchozího true na false, a tím pádem se program spustí v okně.

```
    // Dotaz na uživatele pro fullscreen/okno
    if (MessageBox(NULL, "Would You Like To Run In Fullscreen Mode?", "Start
        FullScreen?", MB_YESNO | MB_ICONQUESTION) == IDNO)
    {
        fullscreen=FALSE; // Běh v okně
    }
```

Vytvoříme OpenGL okno. Zadáme text titulku, šířku, výšku, barevnou hloubku a true (fullscreen), nebo false (okno) jako parametry do funkce CreateGLWindow(). Tak a je to! Je to pěkně lehké, že? Pokud se okno nepodaří z nějakého důvodu vytvořit, bude vráceno false a program se okamžitě ukončí.

```
    if (!CreateGLWindow("NeHe's OpenGL Framework", 640, 480, 16, fullscreen)) // Vytvoření
        OpenGL okna
    {
        return 0; // Konec programu při chybě
    }
```

Smyčka se opakuje tak dlouho, dokud se done rovná false.

```
    while(!done) // Hlavní cyklus programu
    {
```

První věc, kterou uděláme, je zkontrolování zpráv pro okno. Pomocí funkce PeekMessage() můžeme zjistit zda nějaké zprávy čekají na zpracování bez toho, aby byl program pozastaven. Mnoho programů používá funkci GetMessage(). Pracuje to skvěle, ale program nic nedělá, když nedostává žádné zprávy.

```
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) // Přišla zpráva?
        {
```

Zkontrolujeme, zda jsme neobdrželi zprávu pro ukončení programu. Pokud je aktuální zpráva WM_QUIT, která je způsobena voláním funkce PostQuitMessage(0), nastavíme done na true, čímž přerušíme hlavní cyklus a ukončíme program.

```
            if (msg.message==WM_QUIT) // Obdrželi jsme zprávu pro ukončení?
            {
                done=TRUE; // Konec programu
            }
            else // Předáme zprávu proceduře okna
            {
```

Když zpráva nevyzývá k ukončení programu, tak předáme funkcím TranslateMessage() a DispatchMessage() referenci na tuto zprávu, aby ji funkce WndProc() nebo Windows zpracovaly.

```
                TranslateMessage(&msg); // Přeloží zprávu
                DispatchMessage(&msg); // Odešle zprávu
            }
        }
        else // Pokud nedošla žádná zpráva
        {
```

Pokud zde nebudou již žádné zprávy, překreslíme OpenGL scénu. Následující řádek kontroluje, zda je okno aktivní. Naše scéna je vyrenderována a je zkontrolována vrácená hodnota. Když funkce DrawGLScene() vrátí false nebo je

stisknut ESC, hodnota proměnné done je nastavena na true, což ukončí běh programu.

```
if (active)// Je program aktivní?  
{  
    if (keys[VK_ESCAPE])// Byl stisknut ESC?  
    {  
        done=TRUE;// Ukončíme program  
    }  
    else// Překreslení scény  
    {
```

Když všechno proběhlo bez problémů, prohodíme obsah bufferů (s použitím dvou bufferů předejdeme blikání obrazu při překreslování). Použitím dvojitého bufferingů všechno vykreslujeme do obrazovky v paměti, kterou nevidíme. Jakmile vyměníme obsah bufferů, to co je na obrazovce se přesune do této skryté obrazovky a to, co je ve skryté obrazovce se přenesse na monitor. Díky tomu nevidíme probliknutí.

```
        DrawGLScene();// Vykreslení scény  
        SwapBuffers(hdc);// Prohození bufferů (Double Buffering)  
    }  
}
```

Při stisku klávesy F1 přepneme z fullscreenu do okna a naopak.

```
if (keys[VK_F1])// Byla stisknuta klávesa F1?  
{  
    keys[VK_F1]=FALSE;// Označ ji jako nestisknutou  
    KillGLWindow();// Zruší okno  
    fullscreen=!fullscreen;// Negace fullscreen  
  
    // Znovuvytvoření okna  
    if (!CreateGLWindow("NeHe's OpenGL Framework",640,480,16,fullscreen))  
    {  
        return 0;// Konec programu pokud nebylo vytvořeno  
    }  
}
```

Pokud se proměnná done rovná true, hlavní cyklus se přeruší. Zavřeme okno a opustíme program.

```
KillGLWindow();// Zavře okno  
return (msg.wParam);// Ukončení programu  
}
```

V této lekci jsem se vám pokoušel co nejpodrobněji vysvětlit každý krok při nastavování a vytváření OpenGL programu. Program se ukončí při stisku klávesy ESC a sleduje, zda je okno aktivní či nikoliv.

**napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Václav Slováček - Wessan <horizont (zavináč) host.sk>**

Lekce 2 - Vytváření trojúhelníků a čtyřúhelníků

Zdrojový kód z první lekce trochu upravíme, aby program vykreslil trojúhelník a čtverec. Víím, že si asi myslíte, že takovéto vykreslování je banalita, ale až začnete programovat pochopíte, že orientovat se ve 3D prostoru není na představivost až tak jednoduché. Jakékoli vytváření objektů v OpenGL závisí na trojúhelnících a čtvercích. Pokud pochopíte tuto lekci máte napůl vyhráno.

Chcete-li použít kód z první lekce, tak přepište funkci DrawGLScene(), vše ostatní zůstává nezměněno.

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Vymaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice
```

Použitím glLoadIdentity() se přesunete doprostřed obrazovky. Osa x prochází zleva doprava, osa y zesponu nahoru a osa z od monitoru směrem k vám. Střed OpenGL scény je na souřadnicích, kde se x, y i z rovná nule. Funkce glTranslatef(x, y, z) pohybuje počátkem souřadnicových os. Následující řádek ho posune doleva o 1.5 jednotky. Nepřesouváme se na ose y (0.0f) a posune se směrem dovnitř obrazovky o 6.0 jednotek. Při použití glTranslatef(x, z, z), se nepohybujeme vždy z centra obrazovky, ale z místa, kde jsme se s pomocí této funkce dostali. Zadáváme tedy pouze offset pohybu.

```
    glTranslatef(-1.5f, 0.0f, -6.0f); // Posun doleva a do hloubky
```

Nyní jsme se posunuli do levé poloviny obrazovky a nastavili jsme místo pohledu dostatečně hluboko, takže můžeme vidět celou scénu. Vykreslíme trojúhelník. Posun po ose z určuje jak velké budou vykreslované objekty (perspektiva). glBegin(GL_TRIANGLES) OpenGL říká, že chceme začít kreslit trojúhelníky a glEnd() oznamuje ukončení kreslení trojúhelníků. Kreslení trojúhelníků je na většině grafických karet o hodně rychlejší než kreslení čtverců. Pokud chcete spojovat čtyři body použijete GL_QUADS. Mnohoúhelníky se vytvářejí pomocí GL_POLYGON. Většina karet je ale stejně konvertuje na trojúhelníky. V našem jednoduchém programu nakreslíme pouze jeden trojúhelník. Pokud bychom chtěli nakreslit druhý trojúhelník, stačí přidat další tři body hned za první tři. Všechny šest řádků by bylo mezi glBegin(GL_TRIANGLES) a glEnd() a každá skupina po třech tvoří jeden trojúhelník. Tohle platí stejně i pro čtyřúhelníky, kde jsou skupiny brány po čtyřech. Kreslení mnohoúhelníku je ale už o něčem jiném, protože může být vytvořen z libovolného počtu bodů. Všechny body by byly přiřazeny k jedinému mnohoúhelníku. První řádek po glBegin() definuje první bod trojúhelníku. První parametr ve funkci glVertex3f() určuje souřadnici na ose x, druhý parametr osu y a třetí parametr osu z. V prvním řádku se tedy nepohybujeme po ose x. Bod umístíme pouze o jednu jednotku nahoru na ose y a na ose z necháme opět nulu. Tím nastavíme horní bod. Druhá volání glVertex3f() umísťuje bod posunutý o jednotku vlevo a dolů. Tím vytvoříme druhý vrchol. Třetí funkce umísťuje bod vpravo na x a dolů na y. Máme vytvořený třetí vrchol. Funkcí glEnd() řekneme OpenGL, že už nebudeme umísťovat další body. Zobrazí se vyplněný trojúhelník.

```
    glBegin(GL_TRIANGLES); // Začátek kreslení trojúhelníků
    glVertex3f( 0.0f, 1.0f, 0.0f); // Horní bod
    glVertex3f(-1.0f, -1.0f, 0.0f); // Levý dolní bod
    glVertex3f( 1.0f, -1.0f, 0.0f); // Pravý dolní bod
    glEnd(); // Ukončení kreslení trojúhelníků
```

Na levé straně obrazovky jsem vykreslili trojúhelník. Provedeme translaci doprava a umístíme čtverec. Na ose x se posunujeme o 1.5, abychom dosáhli zpět středu a k tomu přičteme dalších 1.5 a budeme vpravo. Na osách y a z se nepřesunujeme.

```
    glTranslatef(3.0f, 0.0f, 0.0f); // Posun o 3 jednotky doprava
```

Kód pro nakreslení čtverce je velice podobný tomu, který jsme použili pro trojúhelník. Jediný rozdíl v použití GL_QUADS místo GL_TRIANGLES je přidání dalšího vertexu pro čtvrtý bod. Nakreslíme postupně levý horní, pravý horní, pravý dolní a levý dolní vrchol.

```
    glBegin(GL_QUADS); // Začátek kreslení obdélníků
    glVertex3f(-1.0f, 1.0f, 0.0f); // Levý horní bod
    glVertex3f( 1.0f, 1.0f, 0.0f); // Pravý horní bod
    glVertex3f( 1.0f, -1.0f, 0.0f); // Pravý dolní bod
    glVertex3f(-1.0f, -1.0f, 0.0f); // Levý dolní bod
    glEnd(); // Konec kreslení obdélníků
```

```
    return TRUE; // Ukončení funkce
```

```
}
```

To je pro tuto lekci vše. Probrali jsme nejjednodušší kreslení. Doufám, že vás moc neodradilo - to teprve přijde :-]

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Václav Slovák - Wessan <horizont (zavináč) host.sk>

Lekce 3 - Barvy

S jednoduchým rozšířením znalostí ze druhé lekce budete moci používat barvy. Naučíte se jak ploché vybarvování, tak i barevné přechody. Barvy rozzáří vzhled aplikace a tím spíše zaujmou diváka.

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Vymaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(-1.5f, 0.0f, -6.0f); // Posun doleva a do hloubky

    glBegin(GL_TRIANGLES); // Začátek kreslení trojúhelníků
```

Z minulé lekci si pamatujete, že jsme kreslili trojúhelník na levou část obrazovky - to zůstává. Další řádek bude naše první použití příkazu `glColor3f(r,g,b)`. Parametry v závorkách jsou intenzita červené, zelené a modré barvy. Mohou nabývat hodnot od 0 do 1. Pracují stejným způsobem jako u funkce pro barvu pozadí `glClearColor(r, g, b, 1.0f)`. Nastavujeme barvu na čistou červenou (žádná zelená a modrá). S použitím této barvy vykreslíme první vrchol trojúhelníku. Dokud nezměníme barvu, bude mít vše, co nakreslíme červenou barvu.

```
    glColor3f(1.0f, 0.0f, 0.0f); // Červená barva
    glVertex3f(0.0f, 1.0f, 0.0f); // Horní bod
```

Máme umístěn první bod. Teď než umístíme druhý bod, ale předtím změníme barvu na zelenou.

```
    glColor3f(0.0f, 1.0f, 0.0f); // Zelená barva
    glVertex3f(-1.0f, -1.0f, 0.0f); // Levý dolní bod
```

Pravý dolní bod bude modrý. Jakmile provedeme příkaz `glEnd()`, vybarví se trojúhelník. Ale protože má v každém vrcholu jinou barvu, budou se barvy šířit z každého rohu a nakonec se setkají uprostřed, kde se smísí dohromady.

```
    glColor3f(0.0f, 0.0f, 1.0f); // Modrá barva
    glVertex3f(1.0f, -1.0f, 0.0f); // Pravý dolní bod
```

```
glEnd(); // Ukončení kreslení trojúhelníků
```

Vykreslíme čtverec vyplněný modrou barvou. Je důležité zapamatovat si, že cokoli nakreslíme po nastavení barvy, bude vykresleno touto barvou. Každý projekt, který vytváříte používá nějaký způsob vybarvování. Dokonce i ve scénách, kde je vše kresleno pomocí textur, může být funkce `glColor3f()` použita k dodání nádechu požadované barvy.

```
    glTranslatef(3.0f, 0.0f, 0.0f); // Posun o 3 jednotky doprava

    glColor3f(0.5f, 0.5f, 1.0f); // Světle modrá barva

    glBegin(GL_QUADS); // Začátek kreslení obdélníků
    glVertex3f(-1.0f, 1.0f, 0.0f); // Levý horní bod
    glVertex3f( 1.0f, 1.0f, 0.0f); // Pravý horní bod
    glVertex3f( 1.0f, -1.0f, 0.0f); // Pravý dolní bod
    glVertex3f(-1.0f, -1.0f, 0.0f); // Levý dolní bod
    glEnd(); // Konec kreslení obdélníků

    return TRUE; // Ukončení funkce
}
```

V tomto tutoriálu jsem se snažil vysvětlit co nejvíce podrobností o jednobarevném a přechodovém vybarvování mnohoúhelníků. Pohrajte si s tímto kódem, zkuste změnit hodnoty červené, zelené a modré na jiná čísla. Podívejte se co se stane.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Milan Turek <nalim.kerut (zavináč) email.cz>

Lekce 4 - Rotace

Naučíme se, jak otáčet objekt okolo os. Trojúhelník se bude otáčet kolem osy y a čtverec kolem osy x. Je jednoduché vytvořit scénu z polygonů. Přidání pohybu ji pěkně oživí.

Začneme přidáním dvou proměnných pro uložení rotace každého objektu. Deklarujeme je jako globální na začátku programu. Brzy zjistíte, že desetinná čísla jsou nezbytná k programování v OpenGL.

```
GLfloat rtri; // Úhel pro trojúhelník
GLfloat rquad; // Úhel pro čtverec
```

Přepíšeme funkci DrawGLScene, která slouží pro vykreslování.

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Vymaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(-1.5f, 0.0f, -6.0f); // Posun doleva a do hloubky
```

Následující část kódu je nová. Funkce glRotatef(uhel, x_vektor, y_vektor, z_vektor) je odpovědná za rotaci souřadnicových os. Budete ji často používat. Úhel je číslo (obvykle uložené v proměnné), které určuje o kolik stupňů chcete otočit objektem. Parametry x_vektor, y_vektor a z_vektor dohromady se současným počátkem souřadnic určují vektor okolo kterého se objekt bude otáčet.

Pro lepší vysvětlení uvedu vysvětlení na příkladech: **Osa x** - Představte si, že stojíte u vodorovné desky a chcete ji otočit. Pokud zadáte kladnou rotaci [1,0,0] zvedá se vzdálenější část desky a bližší klesá. Při záporné rotaci [-1,0,0] je to naopak. **Osa y** - Opět stejná deska. Při kladné rotaci [0,1,0] se pravá část desky pohybuje od vás a levá část desky k vám. Při záporné rotaci [0,-1,0] je to naopak. **Osa z** - Opět stejná deska. Při kladné rotaci [0,0,1] se zvedá pravá část desky a levá klesá a při záporné [0,0,-1] je to naopak.

Pokud bude rtri 7, následujícím řádkem pootočíme trojúhelníkem o 7° okolo osy y proti směru hodinových ručiček.

```
glRotatef(rtri, 0.0f, 1.0f, 0.0f); // Otočí trojúhelník okolo osy y
```

Další část kódu zůstává nezměněna. Vykreslí vybarvený trojúhelník. Tentokrát ale díky předcházejícímu řádku pootočený okolo osy y.

```
glBegin(GL_TRIANGLES); // Začátek kreslení trojúhelníků

glColor3f(1.0f, 0.0f, 0.0f); // Červená barva
glVertex3f(0.0f, 1.0f, 0.0f); // Horní bod
glColor3f(0.0f, 1.0f, 0.0f); // Zelená barva
glVertex3f(-1.0f, -1.0f, 0.0f); // Levý dolní bod
glColor3f(0.0f, 0.0f, 1.0f); // Modrá barva
glVertex3f(1.0f, -1.0f, 0.0f); // Pravý dolní bod

glEnd(); // Ukončení kreslení trojúhelníků
```

Jistě si v následujícím kódu všimnete, že jsme přidali další volání funkce glLoadIdentity(). Protože byly osy pootočené neukazují do směrů, které předpokládáte. Takže pokud posouváme okolo osy x, můžeme skončit posouváním ve směru původní osy z, záleží na tom, jak moc jsme pootočili. Zkuste odstranit řádek s voláním glLoadIdentity(), ať vidíte co myslím. Jakmile máme resetováno míří osy opět původními směry, tj. x - zleva doprava, y - zdola nahoru, z - z obrazovky k vám. Všimněte si, že posouváme jen o 1,5 na ose x narozdíl od posunu o 3 z předchozí části. Když resetujeme, posune se počátek zpátky do středu obrazovky.

```
glLoadIdentity(); // Reset matice
glTranslatef(1.5f, 0.0f, -6.0f); // Posun počátku

glRotatef(rquad, 1.0f, 0.0f, 0.0f); // Pootočení čtverce okolo osy x
```

Vykreslení je stejné jako v předcházející části. Opět vykreslí čtverec, ale tentokrát pootočený.

```
glColor3f(0.5f, 0.5f, 1.0f); // Světle modrá barva

glBegin(GL_QUADS); // Začátek kreslení obdélníků
glVertex3f(-1.0f, 1.0f, 0.0f); // Levý horní bod
```



```
    glVertex3f( 1.0f, 1.0f, 0.0f); // Pravý horní bod
    glVertex3f( 1.0f,-1.0f, 0.0f); // Pravý dolní bod
    glVertex3f(-1.0f,-1.0f, 0.0f); // Levý dolní bod
glEnd(); // Konec kreslení obdélníků
```

Po každém zobrazení se změní proměnné `rtri` a `rquad`, ve kterých jsou uloženy hodnoty pootočení trojúhelníku a čtverce. Změnou znaménka můžete změnit smysl rotace. Změnou velikosti přičítaných hodnot můžete změnit rychlost rotace.

```
    rtri+=0.2f; // Inkrementace úhlu pootočení trojúhelníku
    rquad-=0.15f; // Inkrementace úhlu pootočení čtverce

    return TRUE; // Ukončení funkce
}
```

Doufám, že jste pochopili, že se vše vykresluje pořád stejně, souřadnice bodů se nikdy nemění. Pokaždé se pouze liší počátek souřadnicových os, jejich natočení nebo měřítko - `glScalef(x,y,z)`. Pokud chcete, aby se objekt otáčel okolo své osy, umístěte ho okolo počátku nebo alespoň na jednu ze souřadnicových os - tato lekce. Při jiném umístění bude chaoticky létat po scéně.

Také je důležité, zda napřed provedete translaci nebo rotaci. Při počátečním pootočení budou osy směřovat jinam než očekáváte a následné posunutí skončí úplně někde jinde než chcete.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Milan Turek <nalim.kerut (zavináč) email.cz>

Lekce 5 - Pevné objekty

Rozšířením poslední části vytvoříme skutečné 3D objekty. Narozdíl od 2D objektů ve 3D prostoru. Změníme trojúhelník na pyramidu a čtverec na krychli. Pyramida bude vybarvena barevným přechodem a každou stěnu krychle vybarvíme jinou barvou.

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Vymaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(-1.5f, 0.0f, -6.0f); // Posun doleva a do hloubky
    glRotatef(rtri, 0.0f, 1.0f, 0.0f); // Otočí pyramidu okolo osy y
}
```

Část kódu vezmeme z předchozí části a vyrobíme pomocí něj 3D objekt. Je jedna věc na kterou jsem často dotazován. Proč se objekty neotáčejí okolo své osy. Vypadá to jako by se létaly po celé obrazovce. Pokud objektu řeknete, aby se otočil okolo osy, otočí se okolo osy souřadnicového systému. Pokud chcete, aby se rotoval okolo své osy, musíte dát počátek souřadnic do jeho středu nebo aspoň tak, aby se souřadnicová osa, okolo které otáčíte, kryla s osou objektu okolo které chcete otočit.

Následující kód vytvoří pyramidu okolo centrální osy. Vrcholek pyramidy je o jednu nahoře od středu, spodek o jednu dolů. Vrchní bod je vpravo uprostřed a dolní body jsou vpravo a vlevo od středu. Všimněte si, že všechny trojúhelníky jsou kresleny ve směru proti hodinovým ručičkám. Je to důležité a bude to vysvětleno v dalších lekcích - např. lekce 11. Teď si pouze zapamatujte, že je dobré kreslit po směru nebo proti směru hodinových ručiček a pokud k tomu nemáte důvod, neměli byste dvě osy prohodit. Začneme kreslením čelní stěny. Protože všechny sdílí horní bod, uděláme jej u všech stěn červený. Barvy na spodních vrcholech se budou střídát. Čelní stěna bude mít levý bod zelený a pravý modrý. Trojúhelník na pravé straně bude mít levý bod modrý a pravý zelený. Prohozením dolních dvou barev na každé stěně uděláme společně vybarvené vrcholy na spodku každé stěny.

```
glBegin(GL_TRIANGLES); // Začátek kreslení pyramidy
glColor3f(1.0f, 0.0f, 0.0f); // Červená
glVertex3f(0.0f, 1.0f, 0.0f); // Horní bod (čelní stěna)
glColor3f(0.0f, 1.0f, 0.0f); // Zelená
glVertex3f(-1.0f, -1.0f, 1.0f); // Levý spodní bod (čelní stěna)
glColor3f(0.0f, 0.0f, 1.0f); // Modrá
glVertex3f( 1.0f, -1.0f, 1.0f); // Pravý spodní bod (čelní stěna)
}
```

Vykreslíme pravou stěnu. Spodní body kreslíme vpravo od středu a horní bod je kreslen o jedna na ose y od středu a pravý střed na ose x. To způsobuje, že se stěna svažuje od horního bodu doprava dolů. Levý bod je tentokrát modrý stejně jako pravý dolní bod čelní stěny, ke kterému přiléhá. Zbylé tři trojúhelníky kresleny ve stejném glBegin(GL_TRIANGLES) a glEnd() jako první trojúhelník. Protože dělám celý objekt z trojúhelníků, OpenGL ví, že každé tři body tvoří trojúhelník. Jakmile nakreslíte tři body a přidáte další body, OpenGL předpokládá, že je třeba kreslit další trojúhelník. Pokud zadáte čtyři body místo tří, OpenGL použije první tři a bude předpokládat, že čtvrtý je začátek dalšího trojúhelníku. Nevykreslí čtverec. Proto si dávejte pozor, aby jste náhodou nepřidali nějaký bod navíc.

```
glColor3f(1.0f, 0.0f, 0.0f); // Červená
glVertex3f( 0.0f, 1.0f, 0.0f); // Horní bod (pravá stěna)
glColor3f(0.0f, 0.0f, 1.0f); // Modrá
glVertex3f( 1.0f, -1.0f, 1.0f); // Levý bod (pravá stěna)
glColor3f(0.0f, 1.0f, 0.0f); // Zelená
glVertex3f( 1.0f, -1.0f, -1.0f); // Pravý bod (pravá stěna)
}
```

Teď vykreslíme zadní stěnu. Opět prohození barev. Levý bod je opět zelený, protože odpovídající pravý bod je zelený.

```
glColor3f(1.0f, 0.0f, 0.0f); // Červená
glVertex3f( 0.0f, 1.0f, 0.0f); // Horní bod (zadní stěna)
glColor3f(0.0f, 1.0f, 0.0f); // Zelená
glVertex3f( 1.0f, -1.0f, -1.0f); // Levý bod (zadní stěna)
glColor3f(0.0f, 0.0f, 1.0f); // Modrá
glVertex3f(-1.0f, -1.0f, -1.0f); // Pravý bod (zadní stěna)
}
```

Nakonec nakreslíme levou stěnu pyramidy. Protože pyramida rotuje okolo osy Y, nikdy nevidíme podstavu. Pokud chcete experimentovat, zkuste přidat ji přidat. Potom pootočte pyramidu okolo osy x a uvidíte zda se vám to povedlo.

```
glColor3f(1.0f, 0.0f, 0.0f); // Červená
```

```

glVertex3f( 0.0f, 1.0f, 0.0f); // Horní bod (levá stěna)
glColor3f(0.0f,0.0f,1.0f); // Modrá
glVertex3f(-1.0f,-1.0f,-1.0f); // Levý bod (levá stěna)
glColor3f(0.0f,1.0f,0.0f); // Zelená
glVertex3f(-1.0f,-1.0f, 1.0f); // Pravý bod (levá stěna)
glEnd(); // Konec kreslení pyramidy

```

Teď vykreslíme krychli. Je tvořena šesti čtverci, které jsou kresleny opět proti směru hodinových ručiček. To znamená, že první bod je pravý horní, druhý levý horní, třetí levý dolní a čtvrtý pravý dolní. Když kreslíme zadní stěnu, může to vypadat, že kreslíme ve směru hodinových ručiček, ale pamatujte, že jsme za krychlí a díváme se směrem k čelní stěně. Takže levá strana obrazovky je pravou stranou čtverce. Tentokrát posouváme krychli trochu dál. Tím velikost více odpovídá velikosti pyramidy a části mohou být oříznuty okraji obrazovky. Můžete si pohrát s nastavením počátku a uvidíte, že posunutím dále se zdá menší a naopak. Důvodem je perspektiva. Vzdálenější objekty se zdají menší.

```

glLoadIdentity(); // Reset matice
glTranslatef(1.5f,0.0f,-7.0f); // Posun počátku vpravo a dozadu
glRotatef(rquad,1.0f,1.0f,1.0f); // Rotace okolo x, y, a z

```

Začneme kreslením vrcholku krychle. Všimněte si, že souřadnice y je vždy jedna. Tím kreslíme stěnu rovnoběžně s rovinou xz. Začneme pravým horním bodem. Ten je o jedna vpravo a o jedna dozadu. Další bod je o jedna vlevo a o jedna dozadu. Poté vykreslíme spodní část čtverce směrem k pozorovateli. Abychom toho dosáhli, narozdíl od posunu do obrazovky, posuneme se o jeden bod z obrazovky.

```

glBegin(GL_QUADS); // Začátek kreslení krychle
glColor3f(0.0f,1.0f,0.0f); // Modrá
glVertex3f( 1.0f, 1.0f,-1.0f); // Pravý horní (horní stěna)
glVertex3f(-1.0f, 1.0f,-1.0f); // Levý horní (horní stěna)
glVertex3f(-1.0f, 1.0f, 1.0f); // Levý dolní (horní stěna)
glVertex3f( 1.0f, 1.0f, 1.0f); // Pravý dolní (horní stěna)

```

Spodní část krychle se kreslí stejným způsobem, jen je posunuta na ose y do -1. Další změna je, že pravý horní bod je tentokrát bod bližší k vám, narozdíl od horní stěny, kde to byl bod vzdálenější. V tomto případě by se nic nestalo pokud by jste pouze zkopírovali předchozí čtyři řádky a změnili hodnotu y na -1, ale později by vám to mohlo přinést problémy například u textur.

```

glColor3f(1.0f,0.5f,0.0f); // Oranžová
glVertex3f( 1.0f,-1.0f, 1.0f); // Pravý horní bod (spodní stěna)
glVertex3f(-1.0f,-1.0f, 1.0f); // Levý horní (spodní stěna)
glVertex3f(-1.0f,-1.0f,-1.0f); // Levý dolní (spodní stěna)
glVertex3f( 1.0f,-1.0f,-1.0f); // Pravý dolní (spodní stěna)

```

Teď vykreslíme čelní stěnu.

```

glColor3f(1.0f,0.0f,0.0f); // Červená
glVertex3f( 1.0f, 1.0f, 1.0f); // Pravý horní (čelní stěna)
glVertex3f(-1.0f, 1.0f, 1.0f); // Levý horní (čelní stěna)
glVertex3f(-1.0f,-1.0f, 1.0f); // Levý dolní (čelní stěna)
glVertex3f( 1.0f,-1.0f, 1.0f); // Pravý dolní (čelní stěna)

```

Zadní stěna.

```

glColor3f(1.0f,1.0f,0.0f); // Žlutá
glVertex3f( 1.0f,-1.0f,-1.0f); // Pravý horní (zadní stěna)
glVertex3f(-1.0f,-1.0f,-1.0f); // Levý horní (zadní stěna)
glVertex3f(-1.0f, 1.0f,-1.0f); // Levý dolní (zadní stěna)
glVertex3f( 1.0f, 1.0f,-1.0f); // Pravý dolní (zadní stěna)

```

Levá stěna.

```

glColor3f(0.0f,0.0f,1.0f); // Modrá
glVertex3f(-1.0f, 1.0f, 1.0f); // Pravý horní (levá stěna)
glVertex3f(-1.0f, 1.0f,-1.0f); // Levý horní (levá stěna)
glVertex3f(-1.0f,-1.0f,-1.0f); // Levý dolní (levá stěna)
glVertex3f(-1.0f,-1.0f, 1.0f); // Pravý dolní (levá stěna)

```

Pravá stěna. Je to poslední stěna krychle. Pokud chcete tak ji vynechejte a získáte krabici. Nebo můžete zkusit nastavit pro každý roh jinou barvu a vybarvit ji barevným přechodem.

```

glColor3f(1.0f,0.0f,1.0f); // Fialová
glVertex3f( 1.0f, 1.0f,-1.0f); // Pravý horní (pravá stěna)
glVertex3f( 1.0f, 1.0f, 1.0f); // Levý horní (pravá stěna)
glVertex3f( 1.0f,-1.0f, 1.0f); // Levý dolní (pravá stěna)

```

```
    glVertex3f( 1.0f,-1.0f,-1.0f); // Pravý dolní (pravá stěna)
glEnd(); // Konec kreslení krychle

    rtri+=0.2f; // Inkrementace úhlu pootočení pyramidy
    rquad-=0.15f; // Inkrementace úhlu pootočení krychle

    return TRUE;
}
```

Na konci tohoto tutoriálu byste měli lépe rozumět jak jsou vytvářeny 3D objekty. Můžete přemýšlet o OpenGL scéně jako o kusu papíru s mnoha průsvitnými vrstvami. Jako gigantická krychle tvořená body. Pokud si dokážete představit v obrazovce hloubku, neměli byste mít problém s vytvářením vlastních 3D objektů.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Milan Turek <nalim.kerut (zavináč) email.cz>

Lekce 6 - Textury

Namapujeme bitmapový obrázek na krychli. Použijeme zdrojové kódy z první lekce, protože je jednodušší (a přehlednější) začít s prázdným oknem než složitě upravovat předchozí lekci.

Porozumění texturám má mnoho výhod. Řekněme, že chcete nechat přeletět přes obrazovku střelu. Až do tohoto tutoriálu byste ji pravděpodobně vytvořili z vybarvených n-úhelníků. S použitím textur můžete vzít obrázek skutečné střely a nechat jej letět přes obrazovku. Co myslíte, že bude vypadat lépe? Fotografie, nebo obrázek poskládaný z trojúhelníků a čtverců? S použitím textur to bude nejen vypadat lépe, ale i váš program bude rychlejší. Střela vytvořená pomocí textury bude jen jeden čtverec pohybující se po obrazovce. Střela tvořená n-úhelníky by mohla být tvořena stovkami, nebo tisíci n-úhelníky. Jeden čtverec pokrytý texturou bude mít mnohem menší nároky.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Přidáme tři nové desetinné proměnné... xrot, yrot a zrot. Tyto proměnné budou použity k rotaci krychle okolo os. Poslední řádek GLuint texture[1] deklaruje prostor pro jednu texturu. Pokud chcete nahrát více než jednu texturu, změňte číslo jedna na číslo odpovídající počet textur, které chcete nahrát.

```
GLfloat xrot;// X Rotace
GLfloat yrot;// Y Rotace
GLfloat zrot;// Z Rotace

GLuint texture[1];// Ukládá texturu
```

Bezprostředně za předcházející kód a před funkci ReSizeGLScene() přidáme následující funkci. Jejím účelem je nahrávání souboru s bitmapou. Pokud soubor neexistuje, vrátí NULL, což vyjadřuje, že textura nemůže být nahrána. Před vysvětlováním kódu je třeba vědět několik **VELMI** důležitých věcí o obrázcích použitých pro textury. Výška a šířka obrázku musí být mocnina dvou, ale nejméně 64 pixelů. Z důvodů kompatibility by neměly být větší než 256 pixelů. Pokud by bitmapa, kterou chcete použít neměla velikost 64, 128 nebo 256, změňte její velikost pomocí editoru obrázků. Existují způsoby jak obejít tyto limity, ale my zůstaneme u standardních velikostí textury. První věc kterou uděláme je deklarace ukazatele na soubor. Na začátku jej nastavíme na NULL.

```
AUX_RGBImageRec *LoadBMP(char *Filename)// Nahraje bitmapu
{
    FILE *File=NULL;// Ukazatel na soubor
```

Dále se ujistíme, že bylo předáno jméno souboru. Je možné zavolat funkci LoadBMP() bez zadání jména souboru, takže to musíme zkontrolovat. Nechceme se snažit nahrát nic. Dále se pokusíme otevřít tento soubor pro čtení, abychom zkontrolovali, zda soubor existuje.

```
    if (!Filename)// Byla předána cesta k souboru?
    {
        return NULL;// Pokud ne, konec
    }

    File=fopen(Filename,"r");// Otevření pro čtení
```

Pokud se nám podařilo soubor otevřít, zjevně existuje. Zavřeme soubor a pomocí funkce auxDIBImageLoad(Filename) vrátíme data obrázku.

```
    if (File)// Existuje soubor?
    {
        fclose(File);// Zavře ho
        return auxDIBImageLoad(Filename);// Načte bitmapu a vrátí na ni ukazatel
```

```
}
```

Pokud se nám soubor nepodařilo otevřít soubor vrátíme NULL, což indikuje, že soubor nemohl být nahrán. Později v programu budeme kontrolovat, zda se vše povedlo v pořádku.

```
return NULL;// Při chybě vrátíme NULL
}
```

Nahraje bitmapu (voláním předchozího kódu) a konvertujeme jej na texturu.

```
int LoadGLTextures()// Loading bitmapy a konverze na texturu
{
```

Deklarujeme bool proměnnou zvanou Status. Použijeme ji k sledování, zda se nám podařilo nebo nepodařilo nahrát bitmapu a sestavit texturu. Její počáteční hodnotu nastavíme na FALSE.

```
int Status=FALSE;// Indikuje chyby
```

Vytvoříme záznam obrázku, do kterého můžeme bitmapu uložit. Záznam bude ukládat výšku, šířku a data bitmapy.

```
AUX_RGBImageRec *TextureImage[1];// Ukládá bitmapu
```

Abychom si byli jisti, že je obrázek prázdný, vynulujeme přidělenou paměť.

```
memset(TextureImage,0,sizeof(void *)*1);// Vynuluje paměť
```

Nahrajeme bitmapu a konvertujeme ji na texturu. TextureImage[0]=LoadBMP("Data/NeHe.bmp") zavolá dříve napsanou funkci LoadBMP(). Pokud se vše podaří, data bitmapy se uloží do TextureImage[0], Status je nastaven na TRUE a začneme sestavovat texturu.

```
if (TextureImage[0]=LoadBMP("Data/NeHe.bmp"))// Nahraje bitmapu a kontroluje vzniklé chyby
{
    Status=TRUE;// Vše je bez problémů
}
```

Teď, když máme nahrána data obrázku do TextureImage[0], sestavíme texturu s použitím těchto dat. První řádek glGenTextures(1, &texture[0]) řekne OpenGL, že chceme sestavit jednu texturu a chceme ji uložit na index 0 pole. Vzpomeňte si, že jsme na začátku vytvořili místo pro jednu texturu pomocí GLuint texture[1]. Druhý řádek glBindTexture(GL_TEXTURE_2D, texture[0]) řekne OpenGL, že texture[0] (první textura), bude 2D textura. 2D textury mají výšku (na ose Y) a šířku (na ose X). Hlavní funkcí glBindTexture() je ukázat OpenGL dostupnou paměť. V tomto případě říkáme OpenGL, že volná paměť je na &texture[0]. Když vytvoříme texturu, bude uložena na tomto paměťovém místě. V podstatě glBindTexture() ukáže do paměti RAM, kde je uložena naše textura.

```
glGenTextures(1, &texture[0]);// Generuje texturu
glBindTexture(GL_TEXTURE_2D, texture[0]);// Typické vytváření textury z bitmapy
```

Vytvoříme 2D texturu (GL_TEXTURE_2D), nula reprezentuje hladinu podrobností obrázku (obvykle se zadává nula). Tři je počet datových komponent. Protože je obrázek tvořen červenou, zelenou a modrou složkou dat, jsou to tři komponenty. TextureImage[0]->sizeX je šířka textury. Pokud znáte šířku, můžete ji tam přímo napsat, ale je jednodušší a univerzálnější nechat práci na počítači. TextureImage[0]->sizeY je analogicky výška textury. Nula je rámeček (obvykle nechán nulový). GL_RGB říká OpenGL, že obrazová data jsou tvořena červenou, zelenou a modrou v tomto pořadí. GL_UNSIGNED_BYTE znamená, že data (jednotlivé hodnoty R, G a B) jsou tvořeny z bezznaménkových bytů a konečně TextureImage[0]->data říká OpenGL, kde vzít data textury. V tomto případě jsou to data uložená v záznamu TextureImage[0].

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);// Vlastní vytváření textury
```

Další dva řádky oznamují OpenGL, jaké použít typy filtrování, když je obrázek větší (GL_TEXTURE_MAG_FILTER) nebo menší (GL_TEXTURE_MIN_FILTER) než originální bitmapa. Já obvykle používám GL_LINEAR pro oba případy. To způsobuje, že textura vypadá hladce ve všech případech. Použití GL_LINEAR požaduje spoustu práce procesoru a video karty, takže když je váš systém pomalý, měli by jste použít GL_NEAREST. Textura filtrovaná pomocí GL_NEAREST bude při zvětšení vypadat kostičkovaně. Lze také kombinovat obojí. GL_LINEAR pro případ zvětšení a GL_NEAREST na zmenšení.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);// Filtrování při zmenšení
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);// Filtrování při zvětšení
}
```

Uvolníme paměť RAM, kterou jsme potřebovali pro uložení dat bitmapy. Ujistíme se, že data bitmapy byla uložena v TextureImage[0]. Pokud ano, ujistíme se, že data byla uložena v poloze data, pokud ano smažeme je. Potom uvolníme

strukturu obrázku.

```
if (TextureImage[0])// Pokud obrázek existuje
{
    if (TextureImage[0]->data)// Pokud existují data obrázku
    {
        free(TextureImage[0]->data);// Uvolní paměť obrázku
    }
    free(TextureImage[0]);// Uvolní strukturu obrázku
}
```

Nakonec vrátíme status. Pokud je všechno v pořádku, obsahuje TRUE. FALSE indikuje chybu.

```
return Status;// Oznámi případné chyby
}
```

Přidáme pár řádků kódu do InitGL. Vypíší celou funkci znovu, takže bude jednoduché najít změny. První řádek if (!LoadGLTextures()) skočí do kódu, který jsme napsali v předchozí části. Nahraje bitmapu a vygeneruje z ní texturu. Pokud z jakéhokoli důvodu selže, tak ukončíme funkci s návratovou hodnotou FALSE. Pokud se texturu podařilo nahrát, povolíme mapování 2D textur - glEnable(GL_TEXTURE_2D). Pokud jej zapomeneme povolit, budou se objekty obvykle zobrazovat jako bílé, což nám asi nebude vyhovovat.

```
int InitGL(GLvoid)// Všechno nastavení OpenGL
{
    if (!LoadGLTextures())// Nahraje texturu
    {
        return FALSE;
    }

    glEnable(GL_TEXTURE_2D);// Zapne mapování textur
    glShadeModel(GL_SMOOTH);// Jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Černé pozadí
    glClearDepth(1.0f);// Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST);// Zapne hloubkové testování
    glDepthFunc(GL_LEQUAL);// Typ hloubkového testování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);// Nejlepší perspektivní korekce
    return TRUE;// Inicializace proběhla v pořádku
}
```

Přejdeme k vykreslování. Pokusíme se o otexturovanou krychli.

```
int DrawGLScene(GLvoid)// Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Vymaže obrazovku a hloubkový
    buffer
    glLoadIdentity();// Reset matice

    glTranslatef(0.0f,0.0f,-5.0f);// Posun do obrazovky

    glRotatef(xrot,1.0f,0.0f,0.0f);// Natočení okolo osy x
    glRotatef(yrot,0.0f,1.0f,0.0f);// Natočení okolo osy y
    glRotatef(zrot,0.0f,0.0f,1.0f);// Natočení okolo osy z
}
```

Následující řádek vybere texturu, kterou chceme použít. Pokud máte více než jednu texturu, vyberete ji úplně stejně, ale s jiným indexem pole glBindTexture(GL_TEXTURE_2D, texture[číslo textury kterou chcete použít]). Tuto funkci nesmíte volat mezi glBegin() a glEnd(). Musíte ji volat vždy před nebo za blokem ohraničeným těmito funkcemi.

```
glBindTexture(GL_TEXTURE_2D, texture[0]);// Zvolí texturu
```

Ke správnému namapování textury na čtyřúhelník se musíte ujistit, že levý horní roh textury je připojen k levému hornímu rohu čtyřúhelníku ap. Pokud rohy textury nejsou připojeny k odpovídajícím rohům čtyřúhelníku, zobrazí se textura natočená, převrácená nebo se vůbec nezobrazí. První parametr funkce glTexCoord2f je souřadnice x textury. 0.0 je levá strana textury, 0.5 střed, 1.0 pravá strana. Druhý parametr je souřadnice y. 0.0 je spodek textury, 0.5 střed, 1.0 vršek. Takže teď víme, že 0.0 na X a 1.0 na Y je levý horní vrchol čtyřúhelníka atd. Vše, co musíme udělat je přiřadit každému rohu čtyřúhelníka odpovídající roh textury. Zkuste experimentovat s hodnotami x a y funkce glTexCoord2f. Změnou 1.0 na 0.5 vykreslíte pouze polovinu textury od 0.0 do 0.5 atd.

```
glBegin(GL_QUADS);
// Přední stěna
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
```

```

glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
// Zadní stěna
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Vrchní stěna
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
// Spodní stěna
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
// Pravá stěna
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Levá stěna
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();

```

Nakonec zvětšíme hodnoty proměnných `xrot`, `yrot` a `zrot`, které určují natočení krychle. Změnou hodnot můžeme změnit rychlost i směr natáčení.

```

xrot+=0.3f;
yrot+=0.2f;
zrot+=0.4f;

return TRUE;
}

```

Po dočtení této lekce byste měli rozumět texturovému mapování. Měli by jste být schopni namapovat libovolnou texturu na libovolný objekt. Až si budete jistí, že tomu rozumíte, zkuste namapovat na každou stěnu krychle jinou texturu

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Milan Turek <nalim.kerut (zavináč) email.cz>

Lekce 7 - Texturové filtry, osvětlení, ovládání pomocí klávesnice

V tomto dílu se pokusím vysvětlit použití tří odlišných texturových filtrů. Dále pak pohybu objektů pomocí klávesnice a nakonec aplikaci jednoduchých světel v OpenGL. Nebude se jako obvykle navazovat na kód z předchozího dílu, ale začne se pěkně od začátku.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Přidáme tři booleovské proměnné. Proměnná light sleduje zda je světlo zapnuté. Proměnné lp a fp nám indikují stisk klávesy 'L' nebo 'F'. Proč je potřebujeme se dozvíme dále. Teď stačí vědět, že zabraňují opakování obslužného kódu při delším držení.

```
bool light;// Světlo ON/OFF
bool lp;// Stisknuto L?
bool fp;// Stisknuto F?

GLfloat xrot;// X Rotace
GLfloat yrot;// Y Rotace
GLfloat xspeed;// Rychlost x rotace
GLfloat yspeed;// Rychlost y rotace
GLfloat z=-5.0f;// Hloubka v obrazovce
```

Následují pole pro specifikaci světla. Použijeme dva odlišné typy. První bude okolní (ambient). Okolní světlo nevychází z jednoho bodu, ale jsou jím nasvíceny všechny objekty ve scéně. Druhým typem bude přímé (diffuse). Přímé světlo vychází z nějakého zdroje a odráží se o povrch. Povrchy objektu, na které světlo dopadá přímo, budou velmi jasné a oblasti málo osvětlené budou temné. To vytváří pěkné stínové efekty po stranách krabice. Světlo se vytváří stejným způsobem jako barvy. Je-li první číslo 1.0f a další dvě 0.0f, dostáváme jasnou červenou. Poslední hodnotou je alfa kanál. Ten tentokrát necháme 1.0f. Červená, zelená a modrá nastavené na stejnou hodnotu vždy vytvoří stín z černé (0.0f) do bílé (1.0f). Bez okolního světla by místa bez přímého světla byla příliš tmavá.

```
GLfloat LightAmbient[]={ 0.5f, 0.5f, 0.5f, 1.0f };// Okolní světlo
```

V dalším řádku jsou hodnoty pro přímé světlo. Protože, jsou všechny hodnoty 1.0f, bude to nejjasnější světlo jaké můžeme získat. Pěkně osvětlí krabici.

```
GLfloat LightDiffuse[]={ 1.0f, 1.0f, 1.0f, 1.0f };// Přímé světlo
```

Nakonec nastavíme pozici světla. Protože chceme aby světlo svítilo na bednu zepředu, nesmíme pohnout světlem na ose x a y. Třetí parametr nám zaručí, že bedna bude osvětlena zepředu. Světlo bude zářit směrem k divákovi. Zdroj světla nevidíme, protože je před monitorem, ale uvidíme jeho odraz od bedny. Poslední číslo definujeme na 1.0f. Určuje koordináty pozice světelného zdroje. Více v další lekci.

```
GLfloat LightPosition[]={ 0.0f, 0.0f, 2.0f, 1.0f };// Pozice světla
```

Proměnná filter bude použita při zobrazení textury. První textura je vytvářena použitím GL_NEAREST. Druhá textura bude GL_LINEAR - filtrování pro úplně hladký obrázek. Třetí textura používá mipmappingu, který tvoří hodně dobrý povrch. Proměnná filter tedy bude nabývat hodnot 0, 1 a 2. GLuint texture[3] ukazuje na tři textury.

```
GLuint filter;// Specifikuje používaný texturový filtr
GLuint texture[3];// Ukládá tři textury
```

Nahrajeme bitmapu a vytvoříme z ní tři různé textury. Tato lekce používá glaux knihovny k nahrávání bitmap. Víme že

Delphi a VC++ mají tuto knihovnu. Co ostatní jazyky, nevím. K tomu už moc říkat nebudu, řádky jsou okomentované a kompletní vysvětlení je v 6 lekci. Nahraje a vytvoří texturu z bitmapy.

```
int LoadGLTextures() // Loading bitmapy a konverze na texturu
{
    int Status=FALSE; // Indikuje chyby
    AUX_RGBImageRec *TextureImage[1]; // Ukládá bitmapu
    memset(TextureImage,0,sizeof(void *)*1); // Vynuluje paměť
```

Nyní nahrajeme bitmapu. Když vše proběhne, data obrázku budou uložena v TextureImage[0], status se nastaví na true a začneme sestavovat texturu.

```
    if (TextureImage[0]=LoadBMP("Data/Crate.bmp")) // Nahraje bitmapu a kontroluje
        vzniklé chyby
    {
        Status=TRUE; // Vše je bez problémů
```

Data bitmapy jsou nahrána do TextureImage[0]. Použijeme je k vytvoření tří textur. Následující řádek oznámí, že chceme sestavit 3 textury a chceme je mít v uloženy v texture[0], texture[1] a texture[2].

```
        glGenTextures(3, &texture[0]); // Generuje tři textury
```

V šesté lekci jsme použili lineární filtrování, které vyžaduje hodně výkonu, ale vypadá velice pěkně. Pro první texturu použijeme GL_NEAREST. Spotřebuje málo výkonu, ale výsledek je relativně špatný. Když ve hře vidíte čtverečkovou texturu, používá toto filtrování, nicméně dobře funguje i na slabších počítačích. Všimněte si že jsme použili GL_NEAREST pro MIN i MAG. Můžeme smíchat GL_NEAREST s GL_LINEAR a textury budou vypadat slušně, ale zároveň nevyžadují vysoký výkon. MIN_FILTER se užívá při zmenšování, MAG_FILTER při zvětšování.

```
        // Vytvoří nelineárně filtrovanou texturu
        glBindTexture(GL_TEXTURE_2D, texture[0]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]-
        >sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
```

Další texturu vytvoříme stejně jako v lekci 6. Lineárně filtrovaná. Jediný rozdíl spočívá v použití texture[1] místo texture [0], protože se jedná o druhou texturu.

```
        // Vytvoří lineárně filtrovanou texturu
        glBindTexture(GL_TEXTURE_2D, texture[1]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]-
        >sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
```

Mipmapping ještě neznáte. Používá se při malém obrázku, kdy mnoho detailů mizí z obrazovky. Takto vytvořený povrch vypadá z blízka dost špatně. Když chcete sestavit mipmapovanou texturu, sestaví se více textur odlišné velikosti a vysoké kvality. Když kreslíte takovou texturu na obrazovku vybere se nejlépe vypadající textura. Nakreslí na obrazovku místo toho, aby změnilo rozlišení původního obrázku, které je příčinou ztráty detailů. V šesté lekci jsem se zmínil o stanovených limitech šířky a výšky - 64, 128, 256 atd. Pro mipmapovanou texturu můžeme použít jakoukoli šířku a výšku bitmapy. Automaticky se změní velikost. Protože toto je textura číslo 3, použijeme texture[2]. Nyní máme v texture [0] texturu bez filtru, texture[1] používá lineární filtrování a texture[2] používá mipmapping.

```
        // Vytvoří mipmapovanou texturu
        glBindTexture(GL_TEXTURE_2D, texture[2]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);

        gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[0]-
        >sizeY, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
    }
```

Můžeme uvolnit všechny paměť zaplněnou daty bitmapy. Otestujeme zda se data nachází v TextureImage[0]. Když tam budou, tak je smažeme. Nakonec uvolníme strukturu obrázku.

```
    if (TextureImage[0]) // Pokud obrázek existuje
    {
        if (TextureImage[0]->data) // Pokud existují data obrázku
        {
            free(TextureImage[0]->data); // Uvolní paměť obrázku
        }
    }
```

```

        free(TextureImage[0]); // Uvolní strukturu obrázku
    }

    return Status; // Oznámi případné chyby
}

```

Nejdůležitější část inicializace spočívá v použití světla.

```

int InitGL(GLvoid) // Všechno nastavení OpenGL
{
    if (!LoadGLTextures()) // Nahraje texturu
    {
        return FALSE;
    }

    glEnable(GL_TEXTURE_2D); // Zapne mapování textur
    glShadeModel(GL_SMOOTH); // Jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST); // Zapne hloubkové testování
    glDepthFunc(GL_LEQUAL); // Typ hloubkového testování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce
}

```

Nastavíme světla - konkrétně light1. Na začátku této lekce jsme definovali okolní světlo do LightAmbient. Použijeme hodnoty nastavené v poli.

```

glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient); // Nastavení okolního světla

```

Hodnoty přímého světla jsou v LightDiffuse.

```

glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse); // Nastavení přímého světla

```

Nyní nastavíme pozici světla. Ta je uložena v LightPosition.

```

glLightfv(GL_LIGHT1, GL_POSITION, LightPosition); // Nastavení pozice světla

```

Nakonec zapneme světlo jedna. Světlo je nastavené, umístěné a zapnuté, jakmile zavoláme glEnable(GL_LIGHTING) rozsvítí se.

```

glEnable(GL_LIGHT1); // Zapne světlo
return TRUE; // Inicializace proběhla v pořádku
}

```

Vykreslíme krychli s texturami. Když nepochopíte co některé řádky dělají, podívejte se do lekce 6.

```

int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Vymaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f, 0.0f, z);

    glRotatef(xrot, 1.0f, 0.0f, 0.0f);
    glRotatef(yrot, 0.0f, 1.0f, 0.0f);
}

```

Další řádek je podobný řádku v lekci 6, ale namísto texture[0] tu máme texture[filter]. Když stiskneme klávesu F, hodnota ve filter se zvýší. Bude-li větší než 2, nastavíme zase 0. Při startu programu bude filter nastaven na 0. Proměnnou filter tedy určujeme, kterou ze tří textur máme použít.

```

glBindTexture(GL_TEXTURE_2D, texture[filter]); // Zvolí texturu

```

Při použití světla musíme definovat normálu povrchu. Je to čára vycházející ze středu polygonu v 90 stupňovém úhlu. Řekne jakým směrem je čelo polygonu. Když ji neurčíte, stane se hodně divných věcí. Povrchy které by měly svítit se nerozsvítí, špatná strana polygonu svítit bude, atd. Normála požaduje bod vycházející z polygonu. Pohled na přední povrch ukazuje že normála je kladná na ose z. To znamená že normála ukazuje k divákovi. Na zadní straně normála jde od diváka, do obrazovky. Když bude kostka otočená o 180 stupňů v na ose x nebo y, přední povrch bude ukazovat do obrazovky a zadní uvidí divák. Bez ohledu na to který povrch je vidět divákem, normála tohoto povrchu jde směrem k němu. Když se tak stane, povrch bude osvětlen. U dalších bodů normály směrem k světlu bude povrch také světlý. Když se posunete do středu kostky, bude tmavý. Normála je bod ven, nikoli dovnitř, proto není světlo uvnitř a tak to má být.

```

glBegin(GL_QUADS);
    // Přední stěna
}

```

```

glNormal3f( 0.0f, 0.0f, 1.0f); // Normála
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
// Zadní stěna
glNormal3f( 0.0f, 0.0f,-1.0f); // Normála
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Horní stěna
glNormal3f( 0.0f, 1.0f, 0.0f); // Normála
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
// Spodní stěna
glNormal3f( 0.0f,-1.0f, 0.0f); // Normála
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
// Pravá stěna
glNormal3f( 1.0f, 0.0f, 0.0f); // Normála
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Levá stěna
glNormal3f(-1.0f, 0.0f, 0.0f); // Normála
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();

xrot+=xspeed;
yrot+=yspeed;
return TRUE;
}

```

Posuneme se dolů k WinMain(). Přidáme kód k zapnutí/vypnutí světla, otáčení, výběr filtru a posun kostky do/z obrazovky. Těsně u konce WinMain() uvidíte příkaz SwapBuffers(hDC). Ihned za tento řádek přidáme kód.

Následující kód zjišťuje, zda je stisknuta klávesa L. Je-li stisknuta ale lp není false, klávesa ještě nebyla uvolněna.

```

// Funkce WinMain()
SwapBuffers(hDC); // Prohození bufferů

if (keys['L'] && !lp) // Klávesa L - světlo
{

```

Když bude lp false, L nebylo stisknuto, nebo bylo uvolněno. Tento trik je použit pro případ, kdy je klávesa držena déle a my chceme, aby se kód vykonal pouze jednou. Při prvním průchodu se lp nastaví na true a proměnná light se invertuje. Při dalším průchodu je už lp true a kód se neprovede až do uvolnění klávesy, které nastaví lp zase na false. Kdyby zde toto nebylo, světlo by při stisku akorát blikalo.

```

lp=TRUE;
light=!light;

```

Nyní se podíváme na proměnnou light. Když bude false, vypneme světlo, když ne zapneme ho.

```

if (!light)
{
    glDisable(GL_LIGHTING); // Vypne světlo
}
else
{
    glEnable(GL_LIGHTING); // Zapne světlo
}

```

```
}
```

Následuje nastavení proměnné lp na false při uvolnění klávesy L.

```
if (!keys['L'])
{
    lp=FALSE;
}
```

Nyní ošetříme stisk F. Když se stiskne, dojde ke zvýšení filter. Pokud bude větší než 2, nastavíme ho zpět na 0. K ošetření delšího stisku klávesy použijeme stejný způsob jako u světla.

```
if (keys['F'] && !fp) // Klávesa F - změna texturového filtru
{
    fp=TRUE;
    filter+=1;

    if (filter>2)
    {
        filter=0;
    }
}

if (!keys['F']) // Uvolnění F
{
    fp=FALSE;
}
```

Otestují stisk klávesy Page Up. Když bude stisknuto, snížíme proměnnou z. To způsobí vzdalování kostky v příkazu glTranslatef(0.0f,0.0f,z).

```
if (keys[VK_PRIOR]) // Klávesa Page Up - zvýší zanoření do obrazovky
{
    z-=0.02f;
}
```

Otestují stisk klávesy Page Down. Když bude stisknuta, zvýšíme proměnnou z. To způsobí přibližování kostky v příkazu glTranslatef(0.0f,0.0f,z).

```
if (keys[VK_NEXT]) // Klávesa Page Down - sníží zanoření do obrazovky
{
    z+=0.02f;
}
```

Dále zkontrolujeme kurzorové klávesy. Bude-li stisknuto vlevo/vpravo, proměnná xspeed se bude zvyšovat/snižovat. Bude-li stisknuto nahoru/dolů, proměnná yspeed se bude zvyšovat/snižovat. Jestli si vzpomínáte, výše jsem psal, že vysoké hodnoty způsobí rychlou rotaci. Dlouhý stisk nějaké klávesy způsobí právě rychlou rotaci kostky.

```
if (keys[VK_UP]) // Šipka nahoru
{
    xspeed-=0.01f;
}

if (keys[VK_DOWN]) // Šipka dolů
{
    xspeed+=0.01f;
}

if (keys[VK_RIGHT]) // Šipka vpravo
{
    yspeed+=0.01f;
}

if (keys[VK_LEFT]) // Šipka vlevo
{
    yspeed-=0.01f;
}
```

Nyní byste měli vědět jak vytvořit vysoce kvalitní, realisticky vypadající, texturovaný objekt. Také jsme se něco dozvěděli o třech různých filtrech. Stiskem určitých kláves můžete pohybovat objektem na obrazovce, a nakonec víme jak aplikovat jednoduché světlo. Zkuste experimentovat s jeho pozicí a barvou.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>

přeložil: Jiří Rajský - RAJSOFT junior <predator.jr (zavináč) seznam.cz>

Lekce 8 - Blending

Další typ speciálního efektu v OpenGL je blending, neboli průhlednost. Kombinace pixelů je určena alfa hodnotou barvy a použitou funkcí. Nabývá-li alfa 0.0f, materiál zprůhlední, hodnota 1.0f přináší pravý opak.

Rovnice blendingu

Nemáte rádi matematiku a chcete vidět, jak se používá průhlednost prakticky, pak přeskočte tuto část. Pro někoho může být nepochopitelná.

```
(Rs Sr + Rd Dr, Gs Sg + Gd Dg, Bs Sb + Bd Db, As Sa + Ad Da)
```

OpenGL vypočítá výsledek blendingu dvou pixelů z předchozí rovnice. 's' a 'r' představují zdrojový a cílový pixel. 'S' a 'D' jsou činitele blendingu. Tyto hodnoty určují jak moc budou pixely průhledné. Většina obyčejných hodnot pro S a D jsou (As, As, As, As) (AKA zdrojová alfa) pro S a (1, 1, 1, 1) - (As, As, As, As) (AKA jedna minus zdrojová alfa) pro D. Rovnice bude vypadat takto:

```
(Rs As + Rd (1 - As), Gs As + Gd (1 - As), Bs As + Bs (1 - As), As As + Ad (1 - As))
```

Nyní už se budeme věnovat praktickému kódu. Použijeme kód z lekce 7.

```
#include <windows.h> // Hlavičkový soubor pro Windows
#include <stdio.h> // Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h> // Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h> // Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h> // Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL; // Privátní GDI Device Context
HGLRC hRC = NULL; // Trvalý Rendering Context
HWND hWnd = NULL; // Obsahuje Handle našeho okna
HINSTANCE hInstance; // Obsahuje instanci aplikace

bool keys[256]; // Pole pro ukládání vstupu z klávesnice
bool active = TRUE; // Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE; // Ponese informaci o tom, zda je program ve fullscreenu
```

Blending se v OpenGL zapíná stejně, jako všechno ostatní. Nastavíme jeho parametry a vypneme depth buffer, jinak by se objekty za průhlednými polygony nevykreslily. To sice není správná cesta k blendingu, ale ve většině jednoduchých projektů bude fungovat. Správná cesta k vykreslení průhledných (alfa < 1.0) polygonů je obrácení depth bufferu (nejvzdálenější objekty se vykreslují první). Např.: uvažujme polygon 1 jako vzdálenější od pozorovatele. Správně by měl být tedy vykreslen polygon 2 a až po něm polygon 1. Když se na to podíváte, jako ve skutečnosti, všechno světlo se mísí za těmito dvěma polygony (jsou-li průhledné), musí se vykreslit polygon 2 první a potom polygon 1. Raději řadte průhledné objekty podle hloubky a kreslete je až po vykreslení celé scény se zapnutým depth bufferem, jinak můžete dostat špatné výsledky. Víím že je to těžké, ale je to jediná správná cesta.

```
bool light; // Světlo ON/OFF
bool blend; // Blending OFF/ON
bool lp; // Stisknuto L?
bool fp; // Stisknuto F?
bool bp; // Stisknuto B?

GLfloat xrot; // X Rotace
GLfloat yrot; // Y Rotace
GLfloat xspeed; // Rychlost x rotace
GLfloat yspeed; // Rychlost y rotace
GLfloat z=-5.0f; // Hloubka v obrazovce

GLfloat LightAmbient[] = { 0.5f, 0.5f, 0.5f, 1.0f }; // Okolní světlo
GLfloat LightDiffuse[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // Přímé světlo
GLfloat LightPosition[] = { 0.0f, 0.0f, 2.0f, 1.0f }; // Pozice světla

GLuint filter; // Použitý texturový filtr
GLuint texture[3]; // Ukládá 3 textury
```

Posuneme se dolů na LoadGLTextures() a změníme jméno textury.

```
// Funkce LoadGLTextures()
    if(TextureImage[0]=LoadBMP("Data/Glass.bmp"))// Loading bitmapy
```

Další řádky přidáme do InitGL(). Nastavení jimi objekty na plný jas a 50% alfu (průhlednost). To znamená, že když bude blending zapnut, objekt bude z 50% průhledný. Alfa hodnota 0.0 je úplná průhlednost, 1.0 je opak. Druhý řádek nastaví typ blendingu.

```
// Funkce InitGL()
    glColor4f(1.0f,1.0f,1.0f,0.5f);// Plný jas, 50% alfa
    glBlendFunc(GL_SRC_ALPHA,GL_ONE);// Funkce blendingu pro průsvitnost založená na
    hodnotě alfa
```

Podívejte se na následující kód, je umístěn na konci lekce ve funkci WinMain().

```
// Funkce WinMain()
    if (keys['B'] && !bp)// Klávesa B - zapne blending
    {
        bp=TRUE;
        blend = !blend;

        if(blend)
        {
            glEnable(GL_BLEND);// Zapne blending
            glDisable(GL_DEPTH_TEST);// Vypne hloubkové testování
        }
        else
        {
            glDisable(GL_BLEND);// Vypne blending
            glEnable(GL_DEPTH_TEST);// Zapne hloubkové testování
        }
    }

    if (!keys['B'])// Uvolnění B
    {
        bp=FALSE;
    }
```

Není to jednoduché? Stačí zapnout blending, vypnout hloubkové testování a zavolat funkci glColor4f(r,g,b,a) - vše, co nakreslíme bude průhledné.

Jak ale nastavíme barvu použitou v textuře? Jednoduše, v modulated texture modu, každý pixel mapovaný texturou je násobkem aktuální barvy. Když je kreslená barva (0.5, 0.6, 0.4), násobíme barvou a předáme (0.5, 0.6, 0.4, 0.2) (alfa se rovná 1.0, není-li určena).

Alfa z textury

Alfa hodnota použitá pro průhlednost může být přečtena z textury pouze jako barva. To se dělá tak, že se předá alfa do obrázku při nahrávání a poté se ve funkci glTexImage2D() použije pro barevný formát GL_RGBA.

napsal: Tom Stanis <stanis (zavináč) cs.wisc.edu>
přeložil: Jiří Rajský - RAJSOFT junior <predator.jr (zavináč) seznam.cz>

Lekce 9 - Pohyb bitmap ve 3D prostoru

Tento tutoriál vás naučí pohyb objektů ve 3D prostoru a kreslení bitmap bez černých míst, zakrývajících objekty za nimi. Jednoduchou animaci a rozšířené použití blendingu. Teď byste už měli rozumět OpenGL velmi dobře. Naučili jste se vše od nastavení OpenGL okna, po mapování textur a použití světel a blendingu. To byl první tutoriál pro středně pokročilé. A pokračujeme dále...

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Twinkle určuje, zda se používá třpytivý efekt a tp indikuje stisk klávesy T.

```
bool twinkle;// Třpytivý efekt
bool tp;// Stisknuto T?
```

Num určuje kolik hvězd bude zobrazeno na obrazovce. Je definováno jako konstanta, takže ho můžete měnit libovolně, ale jen v tomto řádku. Nezkoušejte měnit hodnotu num později v kódu, pokud nechcete přivodit katastrofu.

```
const num=50;// Počet zobrazovaných hvězd
```

Deklarujeme strukturu, v níž budeme uchovávat informace o jednotlivých hvězdách.

```
typedef struct// Struktura hvězdy
{
    int r, g, b;// Barva
    GLfloat dist;// Vzdálenost od středu
    angle;// Úhel natočení
} stars;// Jméno struktury je stars
```

Každá položka v poli star obsahuje objekt struktury stars, tj. pět hodnot popisujících hvězdu.

```
stars star[num];// Pole hvězd o velikosti num
```

Dále vytvoříme proměnné pro nastavení vzdálenosti pozorovatele (zoom) a úhlu pozorování (tilt). Deklarujeme proměnnou spin natáčející hvězdy okolo osy z, což bude vypadat jako by se otáčely okolo své současné pozice. Loop je řídicí proměnná cyklu, který použijeme pro nakreslení všech padesáti hvězd. Texture[1] ukládá jednu černobílou texturu.

```
GLfloat zoom=-15.0f;// Hloubka v obrazovce
GLfloat tilt=90.0f;// Úhel pohledu
GLfloat spin;// Natočení hvězd
GLuint loop;// Řídicí proměnná cyklu
GLuint texture[1];// Ukládá texturu
```

Hned po předcházejícím kódu přidáme kód pro nahrání textury. Nebudu jej znovu opisovat. Je to ten samý jako v lekci 6, 7 a 8. Bitmapa, kterou tentokrát nahrajeme je nazvána star.bmp. Textura bude používat lineární filtrování.

```
if(TextureImage[0]=LoadBMP("Data/Tim.bmp"))// Loading bitmapy
```

V tomto projektu nebudeme používat hloubkové testování, takže pokud používáte kód z lekce 1, ujistěte se, že jste odstranili volání `glDepthFunc(GL_EQUAL)`; a `glEnable(GL_DEPTH_TEST)`; jinak získáte velmi špatné výsledky. Nicméně v tomto kódu používáme mapování textur, takže se ujistěte, že jste přidali řádky, které nejsou v lekci 1. Všimněte si že povolujeme mapování textur a blending.

```
int InitGL(GLvoid)// Všechna nastavení OpenGL
{
```

```

if (!LoadGLTextures())// Nahraje textury
{
    return FALSE;
}

glEnable(GL_TEXTURE_2D);// Zapne texturové mapování
glShadeModel(GL_SMOOTH);// Povolí jemné stínování
glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Černé pozadí
glClearDepth(1.0f);// Nastavení hloubkového bufferu
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);// Nejlepší perspektivní korekce
glBlendFunc(GL_SRC_ALPHA, GL_ONE);// Typ blendingu pro průhlednost
glEnable(GL_BLEND);// Zapne blending

```

Následující kód je nový. Nastaví počáteční úhel, vzdálenost a barvu každé hvězdy. Všimněte si jak je jednoduché změnit hodnoty ve struktuře. Smyčka projde všech 50 hvězd.

```

for (loop=0; loop<num; loop++)// Inicializuje hvězdy
{
    star[loop].angle=0.0f;// Všechny mají na začátku nulový úhel

```

Počítám vzdálenost pomocí aktuální hvězdy (hodnoty proměnné loop), kterou dělím maximálním počtem hvězd. Poté násobím výsledek pěti. V podstatě to posune každou hvězdu o trochu dále než tu předcházející. Když je loop 50 (poslední hvězda), loop děleno num je 1.0f. Příčina proč násobím pěti je, že $1 \cdot 5 = 5$ a to je okraj obrazovky. Nechci aby hvězdy nebyly zobrazené takže 5.0f je perfektní. Pokud nastavíte hodnotu proměnné zoom hlouběji do obrazovky, můžete použít hodnotu větší než 5.0f, ale hvězdy budou menší (z důvodu perspektivy). Všimněte si, že barva každé hvězdy je tvořena pomocí náhodných hodnot od 0 do 255. Můžete se divit jak můžeme použít tak velké hodnoty, když normálně jsou hodnoty barev od 0.0f do 1.0f. Když nastavujeme barvu, použijeme funkci glColor4ub namísto glColor4f. ub znamená unsigned byte, který může nabývat hodnot od 0 do 255. V tomto programu je jednodušší použít byty než generovat desetinné hodnoty.

```

        star[loop].dist=(float(loop)/num)*5.0f;// Vzdálenost od středu
        star[loop].r=rand()%256;// Barva
        star[loop].g=rand()%256;// Barva
        star[loop].b=rand()%256;// Barva
    }
return TRUE;
}

```

Na řadu přichází vykreslování.

```

int DrawGLScene(GLvoid)// Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smaže obrazovku a hloubkový
    buffer
    glBindTexture(GL_TEXTURE_2D, texture[0]);// Výběr textury

    for (loop=0; loop<num; loop++)// Prochází jednotlivé hvězdy
    {
        glLoadIdentity();// Reset matice

        glTranslatef(0.0f,0.0f,zoom);// Přesun do obrazovky o zoom
        glRotatef(tilt,1.0f,0.0f,0.0f);// Naklopení pohledu

```

Ted' pohneme hvězdou. První věc kterou uděláme je pootočení okolo osy y. Další řádek kódu posune hvězdu na ose x. Normálně to znamená posun na pravou stranu obrazovky, ale protože jsme pootočili výhled okolo osy y, kladná hodnota osy x může být kdekoli.

```

        glRotatef(star[loop].angle,0.0f,1.0f,0.0f);// Rotace o úhel konkrétní hvězdy
        glTranslatef(star[loop].dist,0.0f,0.0f);// Přesun vpřed na ose x

```

Hvězda je ve skutečnosti plochá textura. Pokud nakreslíte plochý čtyřúhelník a namapujete na něj texturu, bude to vypadat dobře. Bude čelem k vám, jak má. Ale když scénu pootočíte o 90 stupňů okolo osy y, textura bude čelem k levé nebo pravé straně obrazovky a vy uvidíte pouze tenkou linku, což nechceme. Chceme aby hvězdy byly pořád čelem k nám nezávisle na natočení a naklopení. Uděláme to zrušením všech rotací v opačném pořadí těsně předtím než vykreslíme hvězdu. Pootočíme zpět zadáním invertovaného úhlu pro rotaci a poté zrušíme naklopení opět pomocí záporného úhlu. Protože jsme dříve posunuli počátek, tak je na pozici ve které jsme ji chtěli. Změnili jsme její polohu, ale texturu stále vidíme správně zepředu.

```

        glRotatef(-star[loop].angle,0.0f,1.0f,0.0f);// Zrušení pootočení
        glRotatef(-tilt,1.0f,0.0f,0.0f);// Zrušení naklopení

```

Jestliže je twinkle TRUE nakreslíme na obrazovku nerotující hvězdu. Pro získání rozdílných barev vezmeme maximální

počet hvězd (num) a odečteme číslo aktuální hvězdy (loop), poté odečteme 1, protože loop nabývá hodnot od 0 do num-1. Tímto způsobem získáme hvězdy rozdílných barev. Není to právě nejlepší způsob, ale je efektivní. Poslední hodnota je alfa hodnota. Čím je nižší, tím je hvězda průhlednější. Pokud projde kód podmínkou, bude každá hvězda nakreslena dvakrát. To zpomalí program. O kolik závisí na vašem počítači, ale výsledek bude stát za to - smísí se barvy dvou hvězd. Protože se nenatáčí, budou vypadat, jako by byly animované. Všimněte si jak je jednoduché přidat barvu do textury. Třebaže je textura černobílá, dostaneme takovou barvu, jakou zvolíme před vykreslením.

```

if (twinkle)// Pokud je zapnutý třpytivý efekt
{
    glColor4ub(star[(num-loop)-1].r,star[(num-loop)-1].g,star[(num-loop)-1].b,255);

    glBegin(GL_QUADS);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
    glEnd();
}

```

Teď vykreslíme hlavní hvězdu. Jediný rozdíl od předcházejícího kódu je, že tato hvězda je natočena okolo osy z a má jinou barvu (viz. indexy).

```

glRotatef(spin,0.0f,0.0f,1.0f);
glColor4ub(star[loop].r,star[loop].g,star[loop].b,255);

glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
glEnd();

```

Pootočíme hvězdu zvětšením hodnoty proměnné spin. Poté změním úhel každé hvězdy o loop/num. To znamená, že vzdálenější hvězdy se otáčejí rychleji. Nakonec snížíme vzdálenost hvězdy od středu, takže to vypadá, že jsou nasávány doprostřed.

```

spin+=0.01f;// Pootočení hvězd
star[loop].angle+=float(loop)/num;// Zvýšení úhlu hvězdy
star[loop].dist-=0.01f;// Změna vzdálenosti hvězdy od středu

```

Zkontrolujeme zda hvězda dosáhla středu. Pokud se tak stane, dostane novou barvu a je posunuta o 5 jednotek od středu, takže může opět začít svou cestu jako nová hvězda.

```

if (star[loop].dist<0.0f)// Dosáhla středu
{
    star[loop].dist+=5.0f;// Nová pozice
    star[loop].r=rand()%256;// Nová barva
    star[loop].g=rand()%256;// Nová barva
    star[loop].b=rand()%256;// Nová barva
}
}
return TRUE;
}

```

Přidáme kód zjišťující stisk klávesy T. Přejděte k funkci WinMain(). Najděte řádek SwapBuffers(hDC). Píšeme za něj.

```

// Funkce WinMain()
SwapBuffers(hDC);// Prohození bufferů

if (keys['T'] && !tp)// T - třpytivý efekt
{
    tp=TRUE;
    twinkle=!twinkle;
}

if (!keys['T'])// Uvolnění T
{
    tp=FALSE;
}

if (keys[VK_UP])// Šipka nahoru - nakloní obraz

```

```
{
    tilt-=0.5f;
}
if (keys[VK_DOWN])// Šipka dolu - nakloní obraz
{
    tilt+=0.5f;
}
if (keys[VK_PRIOR])// PageUp - zvětší hloubku
{
    zoom-=0.2f;
}
if (keys[VK_NEXT])// PageDown - zmenší hloubku
{
    zoom+=0.2f;
}
```

A máme hotovo. Naučili jste se jednoduchou, ale celkem efektní animaci.

napsal: **Jeff Molofee - NeHe** <nehe (zavináč) connect.ab.ca>
přeložil: **Milan Turek** <nalim.kerut (zavináč) email.cz>

Lekce 10 - Vytvoření 3D světa a pohyb v něm

Do současnosti jsme programovali otáčející se kostku nebo pár hvězd. Máte (měli byste mít :-) základní pojem o 3D. Ale rotující krychle asi nejsou to nejlepší k tvorbě dobrých deathmatchových protivníků! Nečekejte a začněte s Quakem IV ještě dnes! Tyto dny potřebujete k velkému, komplikovanému a dynamickému 3D světu s pohybem do všech směrů, skvělými efekty zrcadel, portálů, deformacemi a třeba také vysokým frameratem. Tato lekce vám vysvětlí základní strukturu 3D světa a pohybu v něm.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <math.h>// Hlavičkový soubor pro matematickou knihovnu

#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu

bool blend;// Blending ON/OFF
bool bp;// B stisknuto? (blending)
bool fp;// F stisknuto? (texturové filtry)

const float piover180 = 0.0174532925f;// Zjednoduší převod mezi stupni a radiány
float heading;// Pomocná pro přepočítávání xpos a zpos při pohybu
float xpos;// Určuje x-ové souřadnice na podlaze
float zpos;// Určuje z-ové souřadnice na podlaze

GLfloat yrot;// Y rotace (natočení scény doleva/doprava - směr pohledu)
GLfloat walkbias = 0;// Houpání scény při pohybu (simulace kroků)
GLfloat walkbiasangle = 0;// Pomocná pro vypočítání walkbias
GLfloat lookupdown = 0.0f;// Určuje úhel natočení pohledu nahoru/dolů
GLfloat z=0.0f;// Hloubka v obrazovce

GLuint filter;// Použitý texturový filtr
GLuint texture[3];// Ukládá textury
```

Během definování 3D světa stylem dlouhých sérií čísel se stává stále obtížnějším udržet složitý kód přehledný. Musíme třídit data do jednoduchého a především funkčního tvaru. Pro zpřehlednění vytvoříme celkem tři struktury.

Body obsahují skutečná data, která zajímají OGL. Každý bod definujeme pozicí v prostoru (x,y,z) a koordinátami textury (u,v).

```
typedef struct tagVERTEX// Struktura bodu
{
    float x, y, z;// Souřadnice v prostoru
    float u, v;// Texturové koordináty
} VERTEX;
```

Všechno se skládá z ploch. Protože trojúhelníky jsou nejjednodušší, využijeme právě je.

```
typedef struct tagTRIANGLE// Struktura trojúhelníku
{
    VERTEX vertex[3];// Pole tří bodů
} TRIANGLE;
```

Na počátku všeho je sektor. Každý 3D svět je v základě celý ze sektorů. Může jím být místnost, kostka či jakýkoli jiný větší útvar.

```
typedef struct tagSECTOR// Struktura sektoru
```

```

{
    int numtriangles;// Počet trojúhelníků v sektoru
    TRIANGLE* triangle;// Ukazatel na dynamické pole trojúhelníků
} SECTOR;

```

```
SECTOR sector1;// Bude obsahovat všechna data 3D světa
```

Abychom program ještě více zpřehlednili, ve zdrojovém kódu, který se kompiluje, nebudou žádné číselné souřadnice. K exe souboru - výsledku naší práce - přiložíme textový soubor. V něm nadefinujeme všechny body 3D prostoru a k nim odpovídající texturové koordináty. Z důvodu větší přehlednosti přidáme komentáře. Bez nich by byl totální zmatek. Obsah souboru se může kdykoli změnit. Hodit se to bude především při vytváření prostředí - metoda pokusů a omylů, kdy nemusíte pokaždé rekompilovat program. Upravovat může i uživatel a tím si vytvořit vlastní prostředí. Nemusíte mu poskytovat nic navíc, nekuli zdrojové kódy. Tento soubor by přece stejně dostal. Ze začátku bude lepší používat textové soubory (snadná editace, méně kódu), binární odložíme na později.

První řádka NUMPOLLIIES xx určuje celkový počet trojúhelníků. Text za zpětnými lomítky značí komentář. V každém následujícím řádku je definován jeden bod v prostoru a texturové koordináty. Tři řádky určí trojúhelník, celý soubor sektor.

```
NUMPOLLIIES 36
```

```

// Floor 1
-3.0 0.0 -3.0 0.0 6.0
-3.0 0.0 3.0 0.0 0.0
 3.0 0.0 3.0 6.0 0.0
-3.0 0.0 -3.0 0.0 6.0
 3.0 0.0 -3.0 6.0 6.0
 3.0 0.0 3.0 6.0 0.0

```

```

// Ceiling 1
-3.0 1.0 -3.0 0.0 6.0
-3.0 1.0 3.0 0.0 0.0
 3.0 1.0 3.0 6.0 0.0
-3.0 1.0 -3.0 0.0 6.0
 3.0 1.0 -3.0 6.0 6.0
 3.0 1.0 3.0 6.0 0.0

```

... atd. Data jednoho trojúhelníku tedy obecně vypadají takto:

```

x1 y1 z1 u1 v1
x2 y2 z2 u2 v2
x3 y3 z3 u3 v3

```

Otázkou je, jak tyto data vyjmeme ze souboru. Vytvoříme funkci readstr(), která načte jeden **použitelný** řádek.

```

void readstr(FILE *f,char *string)// Načte jeden použitelný řádek ze souboru
{
    do
    {
        fgets(string, 255, f);// Načti řádek
    } while ((string[0] == '/') || (string[0] == '\n'));// Pokud není použitelný načti
    další

    return;
}

```

Tuto funkci budeme volat v SetupWorld(). Nadefinujeme náš soubor jako filein a otevřeme ho pouze pro čtení. Na konci ho samozřejmě zavřeme.

```

void SetupWorld()// Načti 3D svět ze souboru
{
    float x, y, z, u, v;// body v prostoru a koordináty textur
    int numtriangles;// Počet trojúhelníků
    FILE *filein;// Ukazatel na soubor
    char oneline[255];// Znakový buffer
    filein = fopen("data/world.txt", "rt");// Otevření souboru pro čtení

```

Přečteme data sektoru. Tato lekce bude počítat pouze s jedním sektorem, ale není těžké provést malou úpravu. Program potřebuje znát počet trojúhelníků v sektoru, aby věděl, kolik informací má přečíst. Tato hodnota může být definována jako konstanta přímo v programu, ale určitě uděláme lépe, když ji uložíme přímo do souboru (program se přizpůsobí).

```
readstr(filein, oneline); // Načtení prvního použitelného řádku
sscanf(oneline, "NUMPOLLIIES %d\n", &numtriangles); // Vyjmeme počet trojúhelníků
```

Alokujeme potřebnou paměť pro všechny trojúhelníky a uložíme jejich počet do položky struktury.

```
sector1.triangle = new TRIANGLE[numtriangles]; // Alokace potřebné paměti
sector1.numtriangles = numtriangles; // Uložení počtu trojúhelníků
```

Po alokaci paměti můžeme přistoupit k inicializaci všech datových složek sektoru.

```
for (int loop = 0; loop < numtriangles; loop++) // Prochází trojúhelníky
{
    for (int vert = 0; vert < 3; vert++) // Prochází vrcholy trojúhelníků
    {
```

Načteme řádek, do pomocných proměnných uložíme jednotlivé hodnoty a ty znovu uložíme do položek struktury. S mezikrokem je kód mnohem přehlednější.

```
readstr(filein, oneline); // Načte řádek
sscanf(oneline, "%f %f %f %f %f", &x, &y, &z, &u, &v); // Načtení do
pomocných proměnných
```

```
// Inicializuje jednotlivé položky struktury
sector1.triangle[loop].vertex[vert].x = x;
sector1.triangle[loop].vertex[vert].y = y;
sector1.triangle[loop].vertex[vert].z = z;
sector1.triangle[loop].vertex[vert].u = u;
sector1.triangle[loop].vertex[vert].v = v;
```

```
    }
}

fclose(filein); // Zavře soubor
return;
}
```

Právě napsanou funkci zavoláme při inicializaci programu.

```
int InitGL(GLvoid) // Všechna nastavení OpenGL
{
    if (!LoadGLTextures()) // Nahraje texturu
    {
        return FALSE;
    }

    glEnable(GL_TEXTURE_2D); // Zapne mapování textur
    glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Nastavení blendingu pro průhlednost
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Černé pozadí
    glClearDepth(1.0); // Nastavení hloubkového bufferu
    glDepthFunc(GL_LESS); // Typ hloubkového testování
    glEnable(GL_DEPTH_TEST); // Zapne hloubkové testování
    glShadeModel(GL_SMOOTH); // Povolíme jemné stínování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce

    SetupWorld(); // Loading 3D světa

    return TRUE;
}
```

Teď když máme sektor načtený do paměti, potřebujeme ho zobrazit. Už dlouho známe nějaké ty rotace a pohyb, ale kamera vždy směřovala do středu (0,0,0). Každý dobrý 3D engine umožňuje chodit kolem a objevovat svět. Jedna možnost, jak k tomu dospět je točit kamerou a kreslit 3D prostředí relativně k pozici kamery - funkce glLookAt(). Protože tohle ještě neznáme budeme kameru simulovat takto:

- Uživatel stiskne šípku
- Vlevo/vpravo - otočíme svět okolo středu v opačném směru než je rotace kamery - glRotatef()
- Dopředu/dozadu - posuneme svět v opačném směru než je pohyb kamery - glTranslatef()

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Vymaže obrazovku a hloubkový
buffer
```

```

glLoadIdentity();// Reset matice

GLfloat x_m, y_m, z_m, u_m, v_m;// Pomocné souřadnice a koordináty textury
GLfloat xtrans = -xpos;// Pro pohyb na ose x
GLfloat ztrans = -zpos;// Pro pohyb na ose z
GLfloat ytrans = -walkbias-0.25f;// Poskakování kamery (simulace kroků)
GLfloat sceneroty = 360.0f - yrot;// Úhel směru pohledu

int numtriangles;// Počet trojúhelníků

glRotatef(lookupdown, 1.0f,0.0f,0.0f);// Rotace na ose x - pohled nahoru/dolů
glRotatef(sceneroty, 0.0f,1.0f,0.0f);// Rotace na ose y - otočení doleva/doprava
glTranslatef(xtrans, ytrans, ztrans);// Posun na pozici ve scéně

glBindTexture(GL_TEXTURE_2D, texture[filter]);// Výběr textury podle filtru

numtriangles = sector1.numtriangles;// Počet trojúhelníků - pro přehlednost

// Projde a vykreslí všechny trojúhelníky
for (int loop_m = 0; loop_m < numtriangles; loop_m++)
{
    glBegin(GL_TRIANGLES);// Začátek kreslení trojúhelníků
        glNormal3f(0.0f, 0.0f, 1.0f);// Normála ukazuje dopředu - světlo

        x_m = sector1.triangle[loop_m].vertex[0].x;// První vrchol
        y_m = sector1.triangle[loop_m].vertex[0].y;
        z_m = sector1.triangle[loop_m].vertex[0].z;
        u_m = sector1.triangle[loop_m].vertex[0].u;
        v_m = sector1.triangle[loop_m].vertex[0].v;
        glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);// Vykreslení

        x_m = sector1.triangle[loop_m].vertex[1].x;// Druhý vrchol
        y_m = sector1.triangle[loop_m].vertex[1].y;
        z_m = sector1.triangle[loop_m].vertex[1].z;
        u_m = sector1.triangle[loop_m].vertex[1].u;
        v_m = sector1.triangle[loop_m].vertex[1].v;
        glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);// Vykreslení

        x_m = sector1.triangle[loop_m].vertex[2].x;// Třetí vrchol
        y_m = sector1.triangle[loop_m].vertex[2].y;
        z_m = sector1.triangle[loop_m].vertex[2].z;
        u_m = sector1.triangle[loop_m].vertex[2].u;
        v_m = sector1.triangle[loop_m].vertex[2].v;
        glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m);// Vykreslení

        glEnd();// Konec kreslení trojúhelníků
    }
    return TRUE;
}

```

Přejdeme do funkce WinMain() na ovládání klávesnicí. Když je stisknuta šipka vlevo/vpravo, proměnná yrot je zvýšena/snížena, tudíž se natočí výhled. Když je stisknuta šipka dopředu/dozadu, spočítá se nová pozice pro kameru s použitím sinu a kosinu - vyžaduje trochu znalostí trigonometrie. Piover180 je pouze číslo pro konverzi mezi stupni a radiány. Walkbias je offset vytvářející houpání scény při simulaci kroků. Jednoduše upraví y pozici kamery podle sinové vlny. Jako jednoduchý pohyb vpřed a vzad nevypadá špatně.

```

// Funkce WinMain()
    if (keys['B'] && !bp)// Klávesa B - zapne/vypne blending
    {
        bp=TRUE;
        blend=!blend;
        if (!blend)
        {
            glDisable(GL_BLEND);
            glEnable(GL_DEPTH_TEST);
        }
        else
        {
            glEnable(GL_BLEND);
            glDisable(GL_DEPTH_TEST);
        }
    }

```



```

}
if (!keys['B'])
{
    bp=FALSE;
}

if (keys['F'] && !fp)// Klávesa F - cyklování mezi texturovými filtry
{
    fp=TRUE;
    filter+=1;
    if (filter>2)
    {
        filter=0;
    }
}
if (!keys['F'])
{
    fp=FALSE;
}

if (keys[VK_UP])// Šipka nahoru - pohyb dopředu
{
    xpos -= (float)sin(heading*piover180) * 0.05f;// Pohyb na ose x
    zpos -= (float)cos(heading*piover180) * 0.05f;// Pohyb na ose z
    if (walkbiasangle >= 359.0f)
    {
        walkbiasangle = 0.0f;
    }
    else
    {
        walkbiasangle+= 10;
    }
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f;// Simulace
    kroků
}

if (keys[VK_DOWN])// Šipka dolů - pohyb dozadu
{
    xpos += (float)sin(heading*piover180) * 0.05f;// Pohyb na ose x
    zpos += (float)cos(heading*piover180) * 0.05f;// Pohyb na ose z
    if (walkbiasangle <= 1.0f)
    {
        walkbiasangle = 359.0f;
    }
    else
    {
        walkbiasangle-= 10;
    }
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f;// Simulace
    kroků
}

if (keys[VK_RIGHT])// Šipka doprava
{
    heading -= 1.0f;// Natočení scény
    yrot = heading;
}

if (keys[VK_LEFT])// Šipka doleva
{
    heading += 1.0f;// Natočení scény
    yrot = heading;
}

if (keys[VK_PRIOR])// Page Up
{
    lookupdown-= 1.0f;// Natočení scény
}

```

```

if (keys[VK_NEXT])// Page Down
{
    lookupdown+= 1.0f;// Natočení scény
}

```

Vytvořili jsme první 3D svět. Nevypadá sice jako v Quake-ovi, ale my také nejsme Carmack nebo Abrash. Zkuste tlačítka F - texturový filtr a B - blending. PgUp/PgDown nahnýlí kameru nahoru/dolů. Pohyb šipkami vás doufám napadne.

Teď asi přemýšlíte co dál. Možná použijete tento kód na plnohodnotný 3D engine, měli byste být schopni ho vytvořit. Pravděpodobně budete mít ve hře více než jeden sektor, zvláště při použití vchodů.

Tato implementace kódu umožňuje nahrávání mnohonásobných sektorů a má zpětné vykreslování /backface culling/ (nekreslí polygony od kamery). Hodně štěstí v dalších pokusech.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Jiří Rajský - RAJSOFT junior <predator.jr (zavináč) seznam.cz>
kompletně přepsal: Michal Turek - Woq <WOQ (zavináč) email.cz>

Pozn.: Tuto lekci nepsal NeHe, ale Lionel Brits. Jak sám autor uvádí, je to jeho první tutoriál - a bohužel bylo to vidět. Pokud se podíváte do anglické verze, tak zjistíte, že bez zdrojových kódů nemáte absolutní šanci něco pochopit. Někdy je dokonce velmi těžké identifikovat, která část kódu patří ke které funkci. Aby byl text kratší používal vynechávky (někdy i u hodně důležitého kódu - třeba načítání pozic ze souboru), ap. Překlad Jiřího Rajského byl, dá se říct, přesný a to v tomto případě, byla možná chyba. Proto jsem se rozhodl větší část lekce přepsat. Víím, že ani teď to není nijak zvláště slavné, ale snažil jsem se. Kód jsem samozřejmě neupravoval (i když by si to také zasloužil).

Chyby v kódu: Když jsem přepisoval tuto lekci, musel jsem ji pochopit ze zdrojových kódů a při tom jsem našel několik chyb. Je mi to tak trochu blbý, protože bych kód asi sám nedokázal napsat, ale na druhou stranu byste o tom měli vědět.

Zbytečná deklarace proměnné z. Tuto proměnnou autor pravděpodobně používal ze začátku a pak ji nahradil jinou. Svědčí o tom i dvojité testování PageUp/PageDown (do lekce nevypisováno). Nikde jinde ji nenajdete.

Neuvolnění dynamicky alokované paměti. Ve funkci SetupWorld() jsme pomocí operátoru new alokovali paměť pro trojúhelníky. Nikdy v programu, ale není její uvolnění. I když by měl operační systém po skončení programu rušit všechny systémové zdroje, nelze se na to spoléhat. Tuto chybu odstraní například takto:

```

// Přidat na konec funkce KillGLWindow()
delete [] sector1.triangle;// Uvolnění dynamicky alokované paměti

```

Lekce 11 - Efekt vlnící se vlajky

Naučíme se jak pomocí sinusové funkce animovat obrázky. Pokud znáte standartní šetřič Windows "Létající 3D objekty" (i on by měl být programovaný v OpenGL), tak budeme dělat něco podobného.

Budeme vycházet z šesté lekce. Neopisuji celý zdrojový kód, takže možná bude lepší, když budete mít někde po ruce i zdrojový kód ze zmiňované lekce. První věc, kterou musíte udělat je vložit hlavičkový soubor matematické knihovny. Nebudeme pracovat s moc složitou matematikou, nebojte se, použijete pouze siny a kosiny.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <math.h>// Hlavičkový soubor pro matematickou knihovnu
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hdc = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Deklarujte trojrozměrné pole bodů k uložení souřadnic v jednotlivých osách. `Wiggle_count` se použije k nastavení a následnému zjišťování, jak rychle se bude textura vlnit. Proměnná `hold` zajistí plynulé vlnění textury.

```
float points[45][45][3];// Pole pro body v mřížce vlny
int wiggle_count = 0;// Rychlost vlnění

GLfloat xrot;// Rotace na ose x
GLfloat yrot;// Rotace na ose y
GLfloat zrot;// Rotace na ose z

GLfloat hold;// Pomocná, k zajištění plynulosti pohybu
GLuint texture[1];// Ukládá texturu
```

Přesuňte se dolů k funkci `LoadGLTexture()`. Budete používat novou texturu s názvem `Tim.bmp`, takže najdete funkci `LoadBMP("Data/NeHe.bmp")` a přepište ji tak, aby nahrávala nový obrázek.

```
if(TextureImage[0]=LoadBMP("Data/Tim.bmp"))// Loading bitmapy
```

Teď přidejte následující kód na konec funkce `InitGL()`. Výsledek uvidíte na první pohled. Přední strana textury bude normálně vybarvená, ale jak se po chvíli obrázek natočí, zjistíte, že ze zadní strany zbyl drátěný model. `GL_FILL` určuje klasické kreslení polygony, `GL_LINES` vykresluje pouze okrajové linky, při `GL_POINTS` by šlo vidět pouze vrcholové body. Která strana polygonu je přední a která zadní nelze určit jednoznačně, stačí rotace a už je to naopak. Proto vznikla konvence, že mnohoúhelníky, u kterých byly při vykreslování zadány vrcholy proti směru hodinových ručiček jsou převrácené.

```
// Konec funkce InitGL()
glPolygonMode(GL_BACK, GL_FILL);// Přední strana vyplněná polygony
glPolygonMode(GL_FRONT, GL_LINE);// Zadní strana vyplněná mřížkou
```

Následující dva cykly inicializují naši síť. Abychom dostali správný index musíme dělit řídicí proměnou smyčky pěti (tzn. $45/9=5$). Odčítám 4,4 od každé souřadnice, aby se vlna vycentrovala na počátku souřadnic. Stejného efektu může být dosaženo s pomocí posunutí, ale já mám radši tuto metodu. Hodnota `points[x][y][2]` je tvořená hodnotou `sinu`. Funkce `sin()` potřebuje radiány, tudíž vezmeme hodnotu ve stupních, což je naše $x/5$ násobené čtyřiceti a pomocí vzorce ($\text{radiány}=2 \cdot \pi \cdot \text{stupně}/360$) ji přepočítáme.

```
for (int x=0; x<45; x++)// Inicializace vlny
{
    for (int y=0; y<45; y++)
    {
        points[x][y][0]=float((x/5.0f)-4.5f);
        points[x][y][1]=float((y/5.0f)-4.5f);
        points[x][y][2]=float(sin(((x/5.0f)*40.0f)/360.0f)*3.141592654*2.0f));
    }
}
```

```

    }
}
return TRUE;
}

```

Na začátku vykreslovací funkce deklarujeme proměnné. Jsou použity jako řídicí v cyklu. Uvidíte je v kódu níže, ale většina z nich neslouží k něčemu jinému než, že kontrolují cykly a ukládají dočasné hodnoty

```

int DrawGLScene(GLvoid) // Vykreslování
{
    int x, y;
    float float_x, float_y, float_xb, float_yb;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f, 0.0f, -12.0f); // Posunutí do obrazovky
    glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Rotace na ose x
    glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Rotace na ose y
    glRotatef(zrot, 0.0f, 0.0f, 1.0f); // Rotace na ose z

    glBindTexture(GL_TEXTURE_2D, texture[0]); // Výběr textury

```

Všimněte si, že čtverce jsou kresleny **po** směru hodinových ručiček. Z toho plyne, že čelní plocha, kterou vidíte bude vyplněná a zezadu bude drátěný model. Pokud bychom čtverce vykreslovali **proti** směru hodinových ručiček drátěný model by byl na přední straně.

```

glBegin(GL_QUADS); // Začátek kreslení čtverců
    for( x = 0; x < 44; x++ ) // Cykly procházejí pole
    {
        for( y = 0; y < 44; y++ )
        {

```

Každý z polygonů (čtverce v síti) má 1/44x1/44 textury. Cyklus určuje levý dolní bod (první 2 řádky). Poté spočítáme pravý horní (další 2 řádky). Takže máme dva body na úhlopříčce čtverce a kombinací hodnot jejich souřadnic získáme zbylé dva body na textuře.

```

        // Vypočítání texturových koordinátů
        float_x = float(x)/44.0f;
        float_y = float(y)/44.0f;
        float_xb = float(x+1)/44.0f;
        float_yb = float(y+1)/44.0f;

        // Zadání jednotlivých bodů
        glTexCoord2f(float_x, float_y);
        glVertex3f(points[x][y][0], points[x][y][1], points[x][y][2]);

        glTexCoord2f(float_x, float_yb);
        glVertex3f(points[x][y+1][0], points[x][y+1][1], points[x][y+1][2]);

        glTexCoord2f(float_xb, float_yb);
        glVertex3f(points[x+1][y+1][0], points[x+1][y+1][1], points[x+1][y+1][2]);

        glTexCoord2f(float_xb, float_y);
        glVertex3f(points[x+1][y][0], points[x+1][y][1], points[x+1][y][2]);
    }
}

glEnd(); // Konec kreslení čtverců

```

Při sudém vykreslení v pořadí přesuneme souřadnice v poli do sousedních souřadnic a tím přesuneme i vlnu o kousek vedle. Celý první sloupec (vnější cyklus) postupně ukládáme do pomocné proměnné. Potom o kousek přesuneme vlnu jednoduchým přiřazením každého prvku do sousedního a nakonec přiřadíme uloženou hodnotu okraje na opačný konec obrázku. Tím vzniká dojem, že když mizí jedna vlna, okamžitě začíná vznikat nová, ale programově je to konec té staré :-] Zjednodušeně řečeno máme jen jednu vlnu, která se po opuštění obrázku přesouvá na začátek. Nakonec vynulujeme wiggles_count, abychom udrželi animaci v chodu.

```

if (wiggles_count == 2) // Pro snížení rychlosti pohybu
{

```

```

for (y = 0; y < 45; y++)// Prochází hodnoty na y
{
    hold=points[0][y][2];// Uloží kraj vlny
    for (x = 0; x < 44; x++)// Prochází hodnoty na x
    {
        points[x][y][2] = points[x+1][y][2];// Přiřazení do sousedního prvku
    }
    points[44][y][2]=hold;// Uložený kraj bude na druhé straně
}
wiggle_count = 0;// Nulování počítadla vykreslování
}
wiggle_count++;// Inkrementace počítadla

```

Aktualizujeme rotaci a ukončíme funkci.

```

xrot+=0.3f;
yrot+=0.2f;
zrot+=0.4f;

return TRUE;
}

```

Zkompilujte a spusťte program. Z přední strany byste měli vidět hezkou vlnící se bitmapu a po následném natočení z ní zůstane pouze drátěný model.

napsal: Bosco <bosco4 (zavináč) home.com>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 12 - Display list

Chcete vědět, jak urychlit vaše programy v OpenGL? Jste unaveni z nesmyslného opisování již napsaného kódu? Nejde to nějak jednodušeji? Nešlo by například jedním příkazem vykreslit otexturovanou krychli? Samozřejmě, že jde. Tento tutoriál je určený speciálně pro vás. Předvytvořené objekty a jejich vykreslování jedním řádkem kódu. Jak snadné...

Řekněme, že programujete hru "Asteroidy". Každý level začíná alespoň se dvěma. No, takže se v klidu posadíte a přijdete na to, jak vytvořit 3d asteroid. Jistě bude z polygonů, jak jinak. Třeba osmistěnný. Pokud byste chtěli pracovat elegantně, vytvoříte cyklus a v něm můžete vše vykreslovat. Skončíte s osmnácti nebo více řádky. V klidu. Ale pozor! Pokud tento cyklus proběhne vícekrát, znatelně zpomalí vykreslování. Jednou, až budete vytvářet mnohem komplexnější objekty a scény, pochopíte, co mám na mysli.

Takže, jaké je řešení? Display list, neboli předvytvořené objekty! Tímto způsobem vytváříte vše pouze jednou. Namapovat textury, barvy, cokoli, co chcete. A samozřejmě musíte tento display list pojmenovat. Jelikož vytváříme asteroidy nazveme display list "asteroid". Ve chvíli, kdy budete chtít vykreslit texturovaný/obarvený asteroid na monitor, všechno, co uděláte je zavolání funkce `glCallList(asteroid)`. Předvytvořený asteroid se okamžitě zobrazí. Protože je jednou vytvořený v paměti (display listu), OpenGL nemusí vše znovu přepočítávat. Odstranili jsme velké zatížení procesoru a umožnili programu běžet o mnoho rychleji.

Připraveni? Vytvoříme scénu skládající se z patnácti krychlí. Tyto krychle jsou vytvořeny z krabice a víka - celkem dva display listy. Víko bude vybarveno na tmavší odstín. Kód vychází z šesté lekce. Přepíšeme většinu programu, aby bylo snazší najít změny.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Deklarujeme proměnné. Napřed místo pro texturu. Další dvě proměnné budou vystupovat jako pointery na místo do paměti RAM, kde jsou uloženy display listy.

```
GLuint texture[1];// Ukládá texturu
GLuint box;// Ukládá display list krabice
GLuint top;// Ukládá display list víka

GLuint xloop;// Pozice na ose x
GLuint yloop;// Pozice na ose y
GLfloat xrot;// Rotace na ose x
GLfloat yrot;// Rotace na ose y
```

Vytvoříme dvě pole barev. První ukládá světlé barvy. Hodnoty ve složených závorkách reprezentují červené, zelené a modré složky. Druhé pole určuje tmavší barvy, které použijeme ke kreslení víka krychle. Chceme, aby bylo tmavší než ostatní stěny.

```
static GLfloat boxcol[5][3]// Pole pro barvy stěn krychle
{
    // Světlé: červená, oranžová, žlutá, zelená, modrá
    {1.0f,0.0f,0.0f},{1.0f,0.5f,0.0f},{1.0f,1.0f,0.0f},{0.0f,1.0f,0.0f},{0.0f,1.0f,1.0f}
};

static GLfloat topcol[5][3]// Pole pro barvy víka krychle
{
    // Tmavé: červená, oranžová, žlutá, zelená, modrá
    {0.5f,0.0f,0.0f},{0.5f,0.25f,0.0f},{0.5f,0.5f,0.0f},{0.0f,0.5f,0.0f},
    {0.0f,0.5f,0.5f}
}
```

```
};
```

Následující funkce generuje display listy.

```
GLvoid BuildLists(// Generuje display listy)
{
```

Začneme oznámením OpenGL, že chceme vytvořit dva listy. `glGenList(2)` pro ně alokuje místo v paměti a vrátí pointer na první z nich.

```
    box=glGenLists(2); // 2 listy
```

Vytvoříme první list. Už jsme zabrali místo pro dva listy a víme, že `box` ukazuje na začátek připravené paměti. Použijeme příkaz `glNewList()`. První parametr `box` řekne, že chceme uložit list do paměti, kam ukazuje. Druhý parametr `GL_COMPILE` říká, že chceme předvytvořit list v paměti tak, aby se nemuselo při každém vykreslování znovu všechno generovat a přepočítávat. `GL_COMPILE` je stejné jako programování. Pokud napíšete program a nahrajete ho do vašeho překladače (kompilery), musíte ho zkompileovat vždy, když ho chcete spustit. Ale pokud bude zkompileován do .exe souboru, všechno, co se musí pro spuštění vykonat je kliknout myší na tento .exe soubor a spustit ho. Samozřejmě bez kompilace. Cokoli OpenGL zkompileje v display listu je možno použít bez jakékoli další potřeby přepočítávání. Urychlí se vykreslování.

```
    glNewList(box, GL_COMPILE); // Nový kompilovaný display list - krabice
    glBegin(GL_QUADS);
        // Spodní stěna
        glNormal3f( 0.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
        // Přední stěna
        glNormal3f( 0.0f, 0.0f, 1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
        // Zadní stěna
        glNormal3f( 0.0f, 0.0f, -1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        // Pravá stěna
        glNormal3f( 1.0f, 0.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
        // Levá stěna
        glNormal3f(-1.0f, 0.0f, 0.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
    glEnd();
    glEndList();
```

Příkazem `glEndList()` oznámíme, že končíme vytváření listu. Cokoli je mezi `glNewList()` a `glEndList()` je součástí display listu a naopak, pokud je něco před nebo za už k němu nepatří. Abychom zjistili, kam ho uložíme druhý display list, vezmeme hodnotu již vytvořeného a přičteme k němu jedničku (na začátku funkce jsme řekli, že děláme 2 display listy, takže je to v pořádku).

```
    top=box+1; // Do top vložíme adresu druhého display listu
```

```
    glNewList(top, GL_COMPILE); // Kompilovaný display list - víko
    glBegin(GL_QUADS);
        // Horní stěna
        glNormal3f( 0.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
```

```

        glEnd();
    glEndList();
}

```

Vytvořili jsme oba display listy. Nahrávání textur je stejné, jako v minulých lekcích. Rozhodl jsem se použít mimapping, protože nemám rád, když vidím pixely. Použijeme obrázek cube.bmp uložený v adresáři data. Najdíte funkci LoadBMP() a upravte řádek se jménem bitmapy.

```

    if (TextureImage[0]=LoadBMP("Data/Cube.bmp"))// Loading texture

```

V inicializační funkci je jen několik změn. Přidáme řádek BuildList(). Všimněte si, že jsme ho umístili až za LoadGLTextures(). Display list by se zkompiloval bez textur.

```

int InitGL(GLvoid)// Všechna nastavení OpenGL
{
    if (!LoadGLTextures())// Nahraje texturu
    {
        return FALSE;
    }
    BuildLists();// Vytvoří display listy
    glEnable(GL_TEXTURE_2D);// Zapne texturové mapování
    glShadeModel(GL_SMOOTH);// Jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Černé pozadí
    glClearDepth(1.0f);// Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST);// Povolí hloubkové testování
    glDepthFunc(GL_LEQUAL);// Typ hloubkového testování

```

Následující tři řádky zapínají rychlé a špinavé osvětlení (quick and dirty lighting). Light0 je předdefinováno na většině video karet, takže zamezí nepříjemnostem při nastavení světel. Po light0 nastavíme osvětlení. Pokud vaše karta nepodporuje light0, uvidíte černý monitor - musíte vypnout světla. Poslední řádka přidává barvu do mapování textur. Nezapneme-li vybarvování materiálu, textura bude mít vždy originální barvu. glColor3f(r,g,b) nebude mít žádný efekt (ve vykreslovací funkci).

```

    glEnable(GL_LIGHT0);// Zapne implicitní světlo
    glEnable(GL_LIGHTING);// Zapne světla
    glEnable(GL_COLOR_MATERIAL);// Zapne vybarvování materiálů

```

Nakonec nastavíme perspektivní korekce, aby obraz vypadal lépe. Vrácením true oznámíme programu, že inicializace proběhla v pořádku.

```

    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);// Nejlepší perspektivní korekce
    return TRUE;
}

```

Přichází na řadu vykreslovací funkce. Jako obvykle, přidáme pár šíleností s matematikou. Tentokrát, ale nebudou žádné siny a kosiny. Začneme vymazáním obrazovky a depth bufferu. Potom namapujeme texturu na krychli. Mohl bych tento příkaz přidat do kódu display listu, Ale teď kdykoli mohu vyměnit aktuální texturu za jinou. Doufám, že už rozumíte, že cokoli je v display listu, tak se nemůže změnit.

```

int DrawGLScene(GLvoid)// Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smaže obrazovku a hloubkový
    buffer

    glBindTexture(GL_TEXTURE_2D, texture[0]);// Výběr textury

```

Máme cyklus s řídicí proměnnou yloop. Tato smyčka je použita k určení pozice krychli na ose y. Vykreslujeme pět řádků, proto kód proběhne pětkrát.

```

    for (yloop=1;yloop<6;yloop++)// Prochází řádky
    {

```

Dále máme vnořený cyklus s proměnnou xloop. Je použitý pro pozici krychli na ose x. Jejich počet závisí na tom, ve kterém řádku se nacházíme. Pokud se nacházíme v horním řádku vykreslíme jednu, ve druhém dvě, atd.

```

        for (xloop=0;xloop<yloop;xloop++)// Prochází sloupce
        {
            glLoadIdentity();// Reset matice

```

Následující řádek přesune počátek souřadnic na daný bod obrazovky. První pohled je to trochu matoucí.

Na ose x: Posuneme se doprava o 1,4 jednotky, takže pyramida je na středu obrazovky. Potom násobíme proměnnou xloop hodnotou 2,8 a přičteme 1,4. (Násobíme hodnotou 2,8, takže krychle nejsou jedna nad druhou. 2,8 je přibližně

jejich šířka, když jsou pootočený o 45 stupňů.) Nakonec odečteme $yloop * 1,4$. To je posune doleva v závislosti na tom, ve které řadě jsme. Pokud bychom je nepřesunuli, seřadí se na levé straně. (A nevypadají jako pyramida.)

Na ose y: Odečteme proměnnou $yloop$ od šesti jinak by pyramida byla vytvořena vzhůru nohama. Poté násobíme výsledek hodnotou 2,4. Jinak krychle budou jedna na vrcholu druhé na ose Y. (2,4 se přibližně rovná výšce krychle). Poté odečteme 7, takže pyramida začíná na spodku obrazovky a je sestavována ze zdola nahoru.

Na ose z: Posuneme 20 jednotek dovnitř. Takže se pyramida vejde akorát na obrazovku.

```
// Pozice krychle na obrazovce
glTranslatef(1.4f+(float(xloop)*2.8f)-(float(yloop)*1.4f), ((6.0f-float
(yloop))*2.4f)-7.0f,-20.0f);
```

Nakloníme krychle o 45 stupňů k pohledu a odečteme $2 * yloop$. Perspektivní mód nachýlí krychle automaticky, takže odečítáme, abychom vykompenzovali naklonění. Není to nejlepší cesta, ale pracuje to. Potom přičteme $xrot$. To nám dává možnost ovládat úhel klávesnicí. Také použijeme rotaci o 45 stupňů na ose y. Přičteme $yrot$ kvůli ovládnutí klávesnicí.

```
// Rotace
glRotatef(45.0f-(2.0f*yloop)+xrot,1.0f,0.0f,0.0f);
glRotatef(45.0f+yrot,0.0f,1.0f,0.0f);
```

Vybereme barvu krabice (světlou). Všimněte si, že používáme `glColor3fv()`. Tato funkce vybírá najednou všechny tři hodnoty (červená, zelená, modrá) najednou a tím nastaví barvu. V tomto případě ji najdeme v poli `boxcol` s indexem $yloop-1$. Tím zajistíme rozlišnou barvu, pro každý řádek pyramidy. Kdybychom použili $xloop-1$, dostali bychom stejné barvy pro každý sloupec.

```
glColor3fv(boxcol[yloop-1]); // Barva
```

Po nastavení barvy zbývá jediné - vykreslit krabici. Pro vykreslení zavoláme pouze funkci `glCallList(box)`. Parametr řekne OpenGL, který display list máme na mysli. Krabice bude vybarvená dříve vybranou barvou, bude posunutá a taky natočená.

```
glCallList(box); // Vykreslení
```

Barvu víka vybíráme úplně stejně, jako před chvílí, akorát z pole tmavších barev. Potom ho vykreslíme.

```
glColor3fv(topcol[yloop-1]); // Barva;
glCallList(top); // Vykreslení
```

```
    }
}
return TRUE;
}
```

Poslední zbytek změn uděláme ve funkci `WinMain()`. Kód přidáme za příkaz `SwapBuffers(hDC)`. Ověříme, zda jsou stisknuty šipky a podle výsledku pohybujeme krychlemi.

```
// Funkce WinMain()
SwapBuffers(hDC); // Výměna bufferů

if (keys[VK_LEFT]) // Šipka vlevo
{
    yrot-=0.2f;
}

if (keys[VK_RIGHT]) // Šipka vpravo
{
    yrot+=0.2f;
}

if (keys[VK_UP]) // Šipka nahoru
{
    xrot-=0.2f;
}

if (keys[VK_DOWN]) // Šipka dolů
{
    xrot+=0.2f;
}
```

Po dočtení této lekce, byste měli rozumět, jak display list pracuje, jak ho vytvořit a jak ho vykreslit. Jsou velkým přínosem. Nejen, že zjednoduší psaní složitějších projektů, ale také přidají trochu na rychlosti celého programu.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 13 - Bitmapové fonty

Často kladená otázka týkající se OpenGL zní: "Jak zobrazit text?". Vždycky jde namapovat texturu textu. Bohužel nad ním máte velmi malou kontrolu. A pokud nejste dobří v blendingu, většinou skončíte smíxováním s ostatními obrázky. Pokud byste chtěli znát lehčí cestu k výstupu textu na jakékoli místo s libovolnou barvou nebo fontem, potom je tato lekce určitě pro vás. Bitmapové fonty jsou 2D písma, které nemohou být rotovány. Vždy je uvidíte zepředu.

Možná si řeknete co je tak těžkého na výstupu textu. Můžete spustit grafický editor, vepsat text do obrázku, nahrát ho jako texturu, zapnout blending a poté namapovat na polygon. Ale tím uberete čas procesoru. V závislosti na typu filtrování může výsledek vypadat rozmazaně nebo jako poskládaný z kostiček. Pokud by měl alfa kanál, skončí smíchaný s objekty na obrazovce. Jistě víte kolik různých fontů je dostupných v systému. V tomto tutoriálu se naučíte jak je používat. Nejen, že bitmapové fonty vypadají stokrát lépe než text na textuře, ale můžete je jednoduše měnit za běhu programu. Není třeba dělat texturu pro každé slovo nebo nápis, který chcete vypsát. Stačí jen jeden příkaz. Snažil jsem se vytvořit tuto funkci co nejjednodušší. Všechno co musíte udělat je napsat `glPrint("Hello, world!")`. Podle dlouhého úvodu můžete říci, že jsem s tímto tutoriálem dost spokojený. Trvalo mi přibližně hodinu a půl napsat tento program. Proč tak dlouho? Protože ve skutečnosti nejsou dostupné žádné informace o používání bitmapových fontů, pokud samozřejmě nemáte rádi MFC. Abych udržel vše jednoduché, rozhodl jsem se, že by bylo pěkné napsat jej v k pochopení jednoduchém C kódu.

Malá poznámka: Tento kód je specifický pro Windows. Používá `wgl` funkce Windows pro vytvoření fontu. Apple má pravděpodobně `agl` podporu, která by měla dělat tu samou věc a X má `glx`. Naneštěstí nemohu zaručit, že tento kód je přenositelný. Pokud má někdo na platformě nezávislý kód pro kreslení fontů na obrazovku, pošlete mi jej a já napíši jiný tutoriál o fontech:

Začneme typickým kódem z lekce 1. Přidáme hlavičkový soubor `stdio.h` pro vstupně výstupní operace, `stdarg.h` pro rozbor textu a konvertování proměnných do textu a konečně `math.h`, takže můžeme pohybovat textem po obrazovce s použitím funkcí `SIN` a `COS`.

```
#include <windows.h>// Hlavičkový soubor pro Windows

#include <math.h>// Hlavičkový soubor pro matematickou knihovnu
#include <stdio.h>// Hlavičkový soubor pro standartní vstup/výstup
#include <stdarg.h>// Hlavičkový soubor pro funkce s proměnným počtem parametrů

#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Přidáme 3 nové proměnné. V `base` uložíme číslo prvního vytvořeného `display` listu. Každý znak bude potřebovat vlastní, takže jich bude relativně dost. Znaků 'A' přiřadíme číslo 65, 'B' 66, 'C' 67 atd. Lehce usoudíte, že 'A' bude uloženo v `base + 65` ('A' je 65 znak `Ascii` tabulky). Dále přidáme 2 čítače, které použijeme k pohybu textu po obrazovce s použitím `sinů` a `kosinů`. Budou sloužit i ke generování barvy znaků (více dále).

```
GLuint base;// Číslo základního display listu znaků
GLfloat cnt1;// Pro pohyb a barvu textu
GLfloat cnt2;// Pro pohyb a barvu textu
```

Následující funkce vytvoří font - asi nejtěžší část. "HFONT font" řekne Windows, že budeme manipulovat s fonty Windows. Vytvořením 96 `display` listů definujeme `base`. Po skončení této operace v ní bude uloženo číslo prvního listu.

```
GLvoid BuildFont(GLvoid)// Vytvoření fontu
{
    HFONT font;// Proměnná fontu
    base = glGenLists(96);// 96 znaků
```

Vytvoříme font. První parametr specifikuje velikost. Všimněte si, že je to záporné číslo. Vložením znaménka mínus

řekneme Windows, aby našlo písmo podle výšky ZNAKU. Pokud bychom použili kladné, hledalo by se podle výšky BUŇKY.

```
font = CreateFont(-24, // Výška
```

Určíme šířku buňky. Zadááním nuly Windows použije implicitní hodnotu. Konkrétní hodnotou vytvoříme font širší.

```
0, // Šířka
```

Úhel escapement natočí font. Není to zrovna nejlepší vlastnost. Kdybyste nepoužili 0, 90, 180 nebo 270 stupňů, font by se pravděpodobně ořezal rámečkem.

```
0, // Úhel escapement  
0, // Úhel orientace
```

Tučnost fontu je užitečný parametr. Lze použít čísla 0 až 1000 nebo některou z předdefinovaných hodnot. FW_DONTCARE (0), FW_NORMAL (400), FW_BOLD (700) a FW_BLACK (900). Je jich samozřejmě více, ale myslím si, že tyto čtyři bohatě stačí (popř. použijte nápovědu MSDN). Čím větší hodnotu použijete, tím bude tučnější.

```
FW_BOLD, // Tučnost  
FALSE, // Kurzíva  
FALSE, // Podtržení  
FALSE, // Přeškrtnutí
```

Znaková sada popisuje typ znaků, které chcete použít. Např. CHINESEBIG5_CHARSET, GREEK_CHARSET, RUSSIAN_CHARSET, DEFAULT_CHARSET atd. ANSI je jediná, kterou používám, nicméně DEFAULT by konečkonců mohlo pracovat také. (Pokud rádi používáte fonty Webdings nebo Wingdings použijte SYMBOL_CHARSET.).

```
ANSI_CHARSET, // Znaková sada
```

Přesnost výstupu říká Windows jakou znakovou sadu použít, mají-li dvě stejná jména. Je-li více možných fontů OUT_TT_PRECIS vybere TRUETYPE verzi, která vypadá mnohem lépe - především, když se zvětší. Zadat můžete také OUT_TT_ONLY_PRECIS, která vždy použije TrueType font.

```
OUT_TT_PRECIS, // Přesnost výstupu (TrueType)
```

Přesnost ořezání je typ ořezání, který se použije, když se font dostane ven z ořezávacího regionu.

```
CLIP_DEFAULT_PRECIS, // Přesnost ořezání
```

Do výstupní kvality můžete zadat PROOF, DRAFT, NONANTIALIASED, DEFAULT nebo ANTIALIASED (méně hranaté).

```
ANTIALIASED_QUALITY, // Výstupní kvalita
```

Nastavíme rodinu a pitch. Do pitch lze zadat DEFAULT_PITCH, FIXED_PITCH a VARIABLE_PITCH. Do rodiny FF_DECORATIVE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS, FF_DONTCARE. Zkuste si s nimi pohrát.

```
FF_DONTCARE | DEFAULT_PITCH, // Rodina a pitch
```

Nakonec zadáme jméno fontu. Spusťte MS Word nebo jiný textový editor a najděte si jméno písma, které se vám líbí.

```
"Courier New"); // Jméno fontu
```

Vybereme font do DC (device context - kontext zařízení) a vytvoříme 96 display listů počínající 32 (v Ascii tabulce jsou před 32 netižitelné znaky, 32 - mezera). Můžete sestavit všech 256 zadáním čísla 256 do glGenList() (výše - na začátku této funkce). Ujistěte se, že smažete všech 256 listů po skončení programu (funkce KillFont(GLvoid)) a samozřejmě musíte napsat v následujícím příkazu místo 32 -> 0 a místo 96 -> 255 (viz. další lekce o fontech).

```
SelectObject(hDC, font); // Výběr fontu do DC  
wglUseFontBitmaps(hDC, 32, 96, base); // Vytvoří 96 znaků, počínaje 32 v Ascii  
}
```

Následující kód je krásně jednoduchý. Smaže 96 vytvořených display listů z paměti počínaje prvním, který je určen v "base". Nejsem si jistý, jestli by to Windows udělali automaticky. Jeden řádek za jistotu stojí.

```
GLvoid KillFont(GLvoid) // Smaže font  
{  
    glDeleteLists(base, 96); // Smaže všech 96 znaků (display listů)  
}
```

A teď přichází na řadu funkce, kvůli níž je napsána tato lekce. Volá se úplně stejně jako klasická printf("Hello, world!"); s tím rozdílem, že na začátek přidáte gl a před závorkou uberete f :-]

```
GLvoid glPrint(const char *fmt, ...) // Klon printf() pro OpenGL
{
```

První řádek alokuje paměť pro 256 znaků. Jakýsi řetězec, který nakonec vypíšeme na obrazovku. Druhou proměnnou tvoří ukazatel do argumentů funkce, který jsme při volání zadali s řetězcem kód této lekce. (printf("%d %d", i, j) - to znáte, ne?)

```
    chartext[256]; // Ukládá řetězec
    va_listap; // Pointer do argumentů funkce
```

Další dva řádky zkoušejí, jestli byl zadán text. Pokud ne fmt ukazuje na nic (NULL) a tudíž se nic nevypíše.

```
    if (fmt == NULL) // Byl předán text?
        return; // Konec
```

Následující kód konvertuje veškeré symboly (%d, %f...) v řetězci na konkrétní hodnoty. Po úpravě bude vše uloženo v text.

```
    va_start(ap, fmt); // Rozbor řetězce
    vsprintf(text, fmt, ap); // Zamění symboly za konkrétní čísla
    va_end(ap); // Výsledek je uložen v text
```

Příkaz glListBase(base-32) je na vysvětlení trochu obtížnější. Řekněme, že vykreslujeme znak 'A', který je reprezentován 65 (v Ascii). Bez glListBase(base-32) OpenGL neví, kde má najít tento znak. Mohlo by vyhledat display list 65, ale pokud by se base rovnalo 1000, tak by 'A' bylo uloženo v display listu 1065. Takže nastavením base na počáteční bod, OpenGL bude vědět, odkud vzít ten správný display list. Odečítáme 32, protože jsme nevytvořili prvních 32 display listů, tudíž je přeskočíme.

```
    glPushAttrib(GL_LIST_BIT); // Uloží současný stav display listů
    glListBase(base - 32); // Nastaví základní znak na 32
```

Zavoláme funkci glCallLists(), která najednou zobrazuje více display listů. strlen(text) vrátí počet znaků v řetězci a tím i počet k zobrazení. Dále potřebujeme znát typ předávaného parametru (poslední). Ani teď nebudeme vkládat více než 256 znaků, takže použijeme GL_UNSIGNED_BYTE (byte může nabývat hodnot 0-255, což je přesně to, co potřebujeme). V posledním parametru předáme text. Každý display list ví, kde je pravá hrana toho předchozího, čímž zamezíme nakupení znaků na sebe, na jedno místo. Před začátkem kreslení následující znaku se přesune o tuto hodnotu doprava (glTranslatef()). Nakonec nastavíme GL_LIST_BIT zpět na hodnotu mající před voláním glListBase().

```
    glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // Vykreslí display listy
    glPopAttrib(); // Obnoví původní stav display listů
```

```
}
```

Jediná změna v inicializačním kódu je příkaz volající BuildFont().

```
int InitGL(GLvoid) // Všechna nastavení OpenGL
{
    glShadeModel(GL_SMOOTH); // Povolí jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST); // Povolí hloubkové testování
    glDepthFunc(GL_EQUAL); // Typ hloubkového testování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce

    BuildFont(); // Vytvoří font

    return TRUE;
}
```

Přejdeme k vykreslování. Po obvyklých inicializacích se přesuneme o 1 jednotku do obrazovky. Bitmapové fonty pracují lépe při použití kolmé (ortho) projekce než při perspektivní. Nicméně kolmá vypadá hůře, tudíž provedeme translaci do obrazovky. Po přesunu o 1 jednotku dovnitř, budeme moci umístit text kamkoli od -0.5 do +0.5 na ose x. Po přesunu o 10 bychom mohli vykreslovat na pozice od -5.0 do +5.0. Nikdy neměňte velikost textu a naprosto nikdy nepoužívejte změnu měřítka glScale(x,y,z). Chcete-li mít font větší či menší musíte na to myslet při vytváření.

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f, 0.0f, -1.0f); // Přesun o 1 do obrazovky
```

Dále nastavíme barvu textu. V tomto případě používám dva čítače. Červená složka se určuje podle kosinu prvního

čítače. Hodnoty se mění od -1.0 do +1.0. Zelená složku vypočítáme podle sinu druhého čítače. Rozsahy jsou stejné jako v předchozím případě. K modré barvě jsou použity oba čítače s kosinem. Hodnoty náležejí od 0.5 do 1.5, tedy výsledek operace nebude nikdy 0 a text bude vždy viditelný.

```
// Pulzování barev závislé na pozici
glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos
(cnt1+cnt2)));
```

K určení pozice použijeme nový příkaz. Střed zůstal na 0.0, ale asi jste si všimli, že schází pozice osy z. Po přesunu o jednotku do obrazovky je levý nejvzdálenější viditelný bod -0.5 a pravý +0.5. Protože se vždy text vykresluje zleva doprava, přesuneme o 0.45 doleva. Tím bude vycentrován na střed. Použitá matematika vykonává stejnou funkci jako při nastavování barvy. Na ose x se text pohybuje od -0.5 do -0.4 (odečetli jsme 0.45). Tím udržíme text vždy viditelný. Na ose y se hranice nacházejí na -0.35 a +0.35.

```
glRasterPos2f(-0.45f+0.05f*float(cos(cnt1)), 0.32f*float(sin(cnt2))); // Pozice textu
```

Vypíšeme text. Tuto funkci jsem navrhl jako super snadnou a uživatelsky příjemnou. Vypadá jako volání printf() ze stdio.h zkřížené s OpenGL. Text se vykreslí na pozici, kam jsme přesunuli před chvílí. Podřetězec %7.2f oznamuje vypisování obsahu proměnné. Sedmička určuje, maximální délku čísla a dvojka upřesňuje počet desetinných míst. f značí desetinné číslo (float). Je mi samozřejmě jasné, že pokud ovládáte jazyk C, tak je to pro vás hračka. Konvence jsou stejné jako u klasického printf(). Pokud to bude nutné můžete se podívat do nápovědy MSDN.

```
glPrint("Active OpenGL Text With NeHe - %7.2f", cnt1); // Výpis textu
```

Nakonec zbývá inkrementování čítače, aby se měnila pozice a barva.

```
cnt1+=0.051f;
cnt2+=0.005f;
return TRUE;
}
```

Poslední kód, který se provede před opuštěním programu je smazání fontu voláním KillFont().

```
//Konec funkce KillGLWindow(GLvoid)
if(!UnregisterClass("OpenGL",hInstance))
{
    MessageBox(NULL,"Could Not Unregister Class.,"SHUTDOWN ERROR",MB_OK |
    MB_ICONINFORMATION);
    hInstance=NULL;
}

KillFont(); //Smazání fontu
}
```

Hotovo... Na internetu jsem hledal podobný tutoriál, ale nic jsem nenašel. Možná jsem první, který píše na podobné téma. Vše je možné. Užijte si výpis textu a šťastné kódování.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 14 - Outline fonty

Bitmapové fonty nestačí? Potřebujete kontrolovat pozici textu i na ose z? Chtěli byste fonty s hloubkou? Pokud zní vaše odpověď ano, pak jsou 3D fonty nejlepší řešení. Můžete s nimi pohybovat na ose z a tím měnit jejich velikost, otáčet je, prostě dělat vše, co nemůžete s obyčejnými. Jsou nejlepší volbou ke hrám a demům.

Tato lekce je volným pokračováním té minulé (13). Tehdy jsme se naučili používat bitmapové fonty. 3D písma se vytvářejí velmi podobně. Nicméně... vypadají stokrát lépe. Můžete je zvětšovat, pohybovat s nimi ve 3D, mají hloubku. Při osvětlení vypadají opravdu efektně. Stejně jako v minulé lekci je kód specifický pro Windows. Pokud by měl někdo na platformě nezávislý kód, sem s ním a já napíšu nový tutoriál. Rozšíříme typický kód první lekce.

```
#include <windows.h>// Hlavičkový soubor pro Windows

#include <math.h>// Hlavičkový soubor pro matematickou knihovnu
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <stdarg.h>// Hlavičkový soubor pro funkce s proměnným počtem parametrů

#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace
```

Base si pamatujete z 13. lekce jako ukazatel na první z display listů ascii znaků, rot slouží k pohybu, rotaci a vybarvování textu.

```
GLuint base;// Číslo základního display listu znaků
GLfloat rot;// Pro pohyb, rotaci a barvu textu

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

GLYPHMETRICSFLOAT gmf[256] ukládá informace o velikosti a orientaci každého z 256 display listů fontu. Dále v lekci vám ukážu, jak zjistit šířku jednotlivých znaků a tím velmi snadno a přesně vycentrovat text na obrazovce.

```
GLYPHMETRICSFLOAT gmf[256];// Ukládá informace o fontu
```

Skoro celý kód následující funkce byl použit již ve 13. lekci, takže pokud mu moc nerozumíte, víte, kde hledat informace.

```
GLvoid BuildFont(GLvoid)// Vytvoření fontu
{
    HFONT font;// Proměnná fontu
    base = glGenLists(256);// 256 znaků

    font = CreateFont(-24,// Výška
        0,// Šířka
        0,// Úhel escapement
        0,// Úhel orientace
        FW_BOLD,// Tučnost
        FALSE,// Kurzíva
        FALSE,// Podtržení
        FALSE,// Přeškrtnutí
        ANSI_CHARSET,// Znaková sada
        OUT_TT_PRECIS,// Přesnost výstupu (TrueType)
        CLIP_DEFAULT_PRECIS,// Přesnost ořezání
        ANTIALIASED_QUALITY,// Výstupní kvalita
        FF_DONTCARE|DEFAULT_PITCH,// Rodina a pitch
        "Courier New");// Jméno fontu

    SelectObject(hDC, font);// Výběr fontu do DC
```

Pomocí funkce wglUseFontOutlines() vytvoříme 3D font. V parametrech předáme DC, první znak, počet display listů,

které se budou vytvářet a ukazatel na paměť, kam se budou vytvořené display listy ukládat.

```
wglUseFontOutlines(hdc, // Vybere DC
0, // Počáteční znak
255, // Koncový znak
base, // Adresa prvního znaku
```

Nastavíme úroveň odchylek, která určuje jak hranatě bude vypadat. Potom určíme šířku nebo spíše hloubku na ose z. 0.0f by byl plochý 2D font. Čím větší číslo přiřadíme, tím bude hlubší. Parametr WGL_FONT_POLYGONS říká, že má OpenGL vytvořit pevné (celistvé) znaky s použitím polygonů. Při použití WGL_FONT_LINES se vytvoří z linek (podobné drátěnému modelu). Je důležité poznamenat, že by se v tomto případě negenerovaly normálové vektory, takže světlo nebude vypadat dobře. Poslední parametr ukazuje na buffer pro uložení informací o display listech.

```
0.0f, // Hranatost
0.2f, // Hloubka v ose z
WGL_FONT_POLYGONS, // Polygony ne drátěný model
gmf); // Adresa bufferu pro uložení informací.
}
```

V následující funkci se maže 256 display listů fontu počínaje prvním, který je definován v base. Nejsem si jistý, jestli by to Windows udělaly automaticky. Jeden řádek za jistotu stojí. Funkce se volá při skončení programu.

```
GLvoid KillFont(GLvoid) // Smaže font
{
    glDeleteLists(base, 256); // Smaže všech 256 znaků
}
```

Tento kód zavoláte vždy, když budete potřebovat vypsát nějaký text. Řetězec je uložen ve "fmt".

```
GLvoid glPrint(const char *fmt, ...) // Klon printf() pro OpenGL
{
```

Proměnnou "length" použijeme ke zjištění délky textu. Pole "text" ukládá konečný řetězec pro vykreslení. Třetí proměnná je ukazatel do parametrů funkce (pokud bychom zavolali funkci s nějakou proměnnou, "ap" na ni bude ukazovat.

```
float length=0; // Délka znaku
char text[256]; // Konečný řetězec
va_list ap; // Ukazatel do argumentů funkce

if (fmt == NULL) // Pokud nebyl předán řetězec
    return; // Konec
```

Následující kód konvertuje veškeré symboly v řetězci (%d, %f ap.) na znaky, které reprezentují číselné hodnoty v proměnných. Poupravovaný text se uloží do řetězce text.

```
va_start(ap, fmt); // Rozbor řetězce pro proměnné
vsprintf(text, fmt, ap); // Zamění symboly za čísla
va_end(ap); // Výsledek je nyní uložen v text
```

Text by šel vycentrovat manuálně, ale následující metoda je určitě lepší. V každém průchodu cyklem přičteme k délce řetězce šířku aktuálního znaku, kterou najdeme v gmf[text[loop]].gmfCellIncX. gmf ukládá informace o každém znaku (display listu), tedy například i výšku znaku, uloženou pod gmfCellIncY. Tuto techniku lze použít při vertikálním vykreslování.

```
for (unsigned int loop=0; loop<(strlen(text)); loop++) // Zjistí počet znaků textu
{
    length+=gmf[text[loop]].gmfCellIncX; // Inkrementace o šířku znaku
}
```

K vycentrování textu posuneme počátek doleva o polovinu délky řetězce.

```
glTranslatef(-length/2, 0.0f, 0.0f); // Zarovnání na střed
```

Nastavíme GL_LIST_BIT a tím zamezíme působení jiných display listů, použitých v programu na glListBase(). Především příkazem určíme, kde má OpenGL hledat správné display listy jednotlivých znaků.

```
glPushAttrib(GL_LIST_BIT); // Uloží současný stav display listů
glListBase(base); // Nastaví první display list na base
```

Zavoláme funkci glCallLists(), která najednou zobrazuje více display listů. strlen(text) vrátí počet znaků v řetězci a tím i počet k zobrazení. Dále potřebujeme znát typ předávaného parametru (poslední). Ani teď nebudeme vkládat více než 256 znaků, takže použijeme GL_UNSIGNED_BYTE (byte může nabyvat hodnot 0-255, což je přesně to, co

potřebujeme). V posledním parametru předáme text. Každý display list ví, kde je pravá hrana toho předchozího, čímž zamezíme nakupení znaků na sebe, na jedno místo. Před začátkem kreslení následující znaku se přesune o tuto hodnotu doprava (glTranslatef()). Nakonec nastavíme GL_LIST_BIT zpět na hodnotu mající před voláním glListBase().

```
    glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // Vykreslí display listy
    glPopAttrib(); // Obnoví původní stav display listů
}
```

Provedeme pár drobných změn v inicializačním kódu. Řádka BuildFont() ze 13. lekce zůstala na stejném místě, ale přibyl nový kód pro použití světel. Light0 je předdefinován na většině grafických karet. Také jsem přidal glEnable(GL_COLOR_MATERIAL). Ke změně barvy písma potřebujeme zapnout vybarvování materiálů, protože i znaky jsou 3D objekty. Pokud vykreslujete vlastní objekty a nějaký text, musíte před funkcí glPrint() zavolat glEnable(GL_COLOR_MATERIAL) a po vykreslení textu glDisable(GL_COLOR_MATERIAL), jinak by se změnila barva i vámi vykreslovaného objektu.

```
int InitGL(GLvoid) // Všechna nastavení OpenGL
{
    glShadeModel(GL_SMOOTH); // Povolí jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST); // Povolí hloubkové testování
    glDepthFunc(GL_LEQUAL); // Typ hloubkového testování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce
    glEnable(GL_LIGHT0); // Zapne implicitní světlo
    glEnable(GL_LIGHTING); // Zapne světla
    glEnable(GL_COLOR_MATERIAL); // Zapne vybarvování materiálů
    BuildFont(); // Vytvoří font
    return TRUE;
}
```

Přesuneme se 10 jednotek do obrazovky. Outline fonty vypadají skvěle v perspektivním módu. Když jsou umístěny hlouběji, zmenšují se. Pomocí funkce glScalef(x,y,z) můžeme také měnit měřítka os. Pokud bychom například chtěli vykreslit font dvakrát vyšší, použijeme glScalef(1.0f,2.0f,1.0f).

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f,0.0f,-10.0f); // Přesun do obrazovky
    glRotatef(rot,1.0f,0.0f,0.0f); // Rotace na ose x
    glRotatef(rot*1.5f,0.0f,1.0f,0.0f); // Rotace na ose y
    glRotatef(rot*1.4f,0.0f,0.0f,1.0f); // Rotace na ose z
}
```

Jako obvykle jsem použil pro změnu barev "jednoduché" matematiky. (Pozn. překladatele: tahle věta se mi povedla :)

```
    // Pulzování barev závislé na pozici a rotaci
    glColor3f(1.0f*float(cos(rot/20.0f)),1.0f*float(sin(rot/25.0f)),1.0f-0.5f*float(cos
    (rot/17.0f)));

    glPrint("NeHe - %3.2f",rot/50); // Výpis textu

    rot+=0.5f; // Inkrementace čítače
    return TRUE;
}
```

Poslední kód, který se provede před opuštěním programu je smazání fontu voláním KillFont().

```
//Konec funkce KillGLWindow(GLvoid)
    if(!UnregisterClass("OpenGL",hInstance))
    {
        MessageBox(NULL,"Could Not Unregister Class.,"SHUTDOWN ERROR",MB_OK |
        MB_ICONINFORMATION);
        hInstance=NULL;
    }

    KillFont(); //Smazání fontu
}
```

Po dočtení této lekce byste měli být schopni používat 3D fonty. Stejně jako jsem psal ve 13. lekci, ani tentokrát jsem na internetu nenašel podobný článek. Možná jsem opravdu první, kdo píše o tomto tématu.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 15 - Mapování textur na fonty

Po vysvětlení bitmapových a 3D fontů v předchozích dvou lekcích jsem se rozhodl napsat lekci o mapování textur na fonty. Jedná se o tzv. automatické generování koordinátů textur. Po dočtení této lekce budete umět namapovat texturu opravdu na cokoli - zcela snadno a jednoduše.

Stejně jako v minulé a předminulé lekci je kód specifický pro Windows. Pokud by měl někdo na platformě nezávislý kód sem s ním a já napíšu nový tutoriál o fontech.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <math.h>// Hlavičkový soubor pro matematickou knihovnu
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup

#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu

GLuint base;// Ukazatel na první z display listů pro font
GLuint texture[1];// Ukládá texturu
GLuint rot;// Pro pohyb, rotaci a barvu textu
```

Při psaní funkce nahrávající font jsem udělal malou změnu. Pokud jste si spustili program, asi jste na první pohled nenašli ten font - ale byl tam. Všimli jste si poletující pirátské lebky se zkříženými kostmi. Právě ona je jeden znak z písma Wingdings, které patří mezi tzv. symbolové fonty.

```
GLvoid BuildFont(GLvoid)// Vytvoření fontu
{
    GLYPHMETRICSFLOAT gmf[256];// Ukládá informace o fontu
    HFONT font;// Proměnná fontu

    base = glGenLists(256);// 256 znaků

    font = CreateFont(-12,// Výška
        0,// Šířka
        0,// Úhel escapement
        0,// Úhel orientace
        FW_BOLD,// Tučnost
        FALSE,// Kurzíva
        FALSE,// Podtržení
        FALSE,// Přeškrtnutí
```

Místo ANSI_CHARSET podle stylu lekce 14, použijeme SYMBOL_CHARSET. Tím řekneme Windowsům, že vytvářený font není typickým písmem tvořeným znaky, ale že obsahuje malé obrázky (symboly). Pokud byste zapomněli změnit tuto řádku, písma typu Wingdings, webdings a další, která zkoušíte použít, nebudou vykreslovat symboly (lebka ...), ale normální znaky (A, B ...).

```
        SYMBOL_CHARSET,// Znaková sada
        OUT_TT_PRECIS,// Přesnost výstupu (TrueType)
        CLIP_DEFAULT_PRECIS,// Přesnost ořezání
        ANTIALIASED_QUALITY,// Výstupní kvalita
        FF_DONTCARE|DEFAULT_PITCH,// Rodina a pitch
        "Wingdings");// Jméno fontu

    SelectObject(hDC, font);// Výběr fontu do DC

    wglUseFontOutlines(hDC,// Vybere DC
        0,// Počáteční znak
        255,// Koncový znak
```

```
base, // Adresa prvního znaku
```

Počítám s větší hranatostí. To znamená, že se OpenGL nebude držet obrysů fontu tak těsně. Pokud zde předáte 0.0f, všimnete si problémů spojených s mapováním textur na zakřivené roviny. Povolíte-li jistou hranatost, většina problémů zmizí. (Já (překladač) jsem žádné problémy s 0.0f neměl, dokonce to vypadalo o dost lépe.)

```
0.1f, // Hranatost
0.2f, // Hloubka v ose z
WGL_FONT_POLYGONS, // Polygony ne drátěný model
gmf); // Adresa bufferu pro uložení informací.
}
```

K nahrání textur přidáme kód, který už znáte z předchozích tutoriálů. Vytvoříme mipmapovanou texturu, protože vypadá lépe.

```
int LoadGLTextures() // Vytvoří texturu
{
    int Status=FALSE; // Indikuje chyby

    AUX_RGBImageRec *TextureImage[1]; // Místo pro obrázek
    memset(TextureImage,0,sizeof(void *)*1); // Nastaví ukazatel na NULL

    if (TextureImage[0]=LoadBMP("Data/Lights.bmp")) // Nahraje bitmapu
    {
        Status=TRUE;

        glGenTextures(1, &texture[0]); // Generuje texturu

        // Vytvoří lineárně mipmapovanou texturu
        glBindTexture(GL_TEXTURE_2D, texture[0]);
        gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[0]-
        >sizeY, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
    }
}
```

Další řádky umožňují použít automatické generování koordinátů textur na jakékoli zobrazované objekty. Příkaz `glTexGen` je velmi silný a komplexní. Popsat všechny vlastnosti, které zahrnuje, by byl tutoriál sám o sobě. Nicméně, vše, co potřebujete vědět, je že, `GL_S` a `GL_T` jsou texturové koordináty. Implicitně jsou nastaveny tak, aby vzaly pozice `x` a `y` na obrazovce a přišly s bodem textury. Všimněte si, že objekty nejsou texturovány na ose `z`. Přední i zadní část ploch je otexturovaná a to je to, na čem záleží. `x` (`GL_S`) mapuje textury zleva doprava a `y` (`GL_T`) nahoru a dolů. `GL_TEXTURE_GEN_MODE` použijeme při výběru texturového mapování `S` i `T`. Jsou celkem tři možnosti v dalším parametru:

`GL_EYE_LINEAR` - textura je namapovaná na všechny stěny stejně
`GL_OBJECT_LINEAR` - textura je fixovaná na přední stěnu, do hloubky se protáhne
`GL_SPHERE_MAP` - textura kovově odrážející světlo

Je důležité poznamenat, že jsem vypustil spoustu kódu. Správně bychom měli určit také `GL_OBJECT_PLANE`, ale implicitní nastavení nám stačí. Pokud byste se chtěli dozvědět více, tak si kupte nějakou dobrou knihu nebo zkuste náповědu MSDN.

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
}

if (TextureImage[0]) // Pokud bitmapa existuje
{
    if (TextureImage[0]->data) // Pokud existují data bitmapy
    {
        free(TextureImage[0]->data); // Smaže data bitmapy
    }

    free(TextureImage[0]); // Smaže strukturu bitmapy
}

return Status;
}
```

Uděláme také několik změn v inicializačním kódu. `BuildFont()` přesuneme pod loading textur. Pokud byste chtěli měnit barvy textur použitím `glColor3f(R,G,B)`, přidejte `glEnable(GL_COLOR_MATERIAL)`.

```

int InitGL(GLvoid) // Všechna nastavení OpenGL
{
    if (!LoadGLTextures()) // Nahraje texturu
    {
        return FALSE;
    }

    BuildFont(); // Vytvoří font

    glShadeModel(GL_SMOOTH); // Jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST); // Povolí hloubkové testování
    glDepthFunc(GL_LEQUAL); // Typ hloubkového testování
    glEnable(GL_LIGHT0); // Zapne implicitní světlo
    glEnable(GL_LIGHTING); // Zapne světla
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce

```

Zapneme automatické mapování textur. texture[0] se teď namapuje na jakýkoli 3D objekt kreslený na obrazovku. Pokud byste potřebovali více kontroly můžete automatické mapování při kreslení ručně zapínat a vypínat.

```

    glEnable(GL_TEXTURE_2D); // Zapne texturové mapování
    glBindTexture(GL_TEXTURE_2D, texture[0]); // Zvolí texturu
    return TRUE;
}

```

Namísto udržování objektu uprostřed obrazovky budeme v této lekci "létat" dokola po celém monitoru. Přesuneme se o 3 jednotky dovnitř. Hodnota pro osu x se bude měnit od -1.1 do +1.1. Krajní meze na ose y jsou -0.8 a +0.8. K výpočtu použijeme proměnnou "rot". Jako vždy budeme rotovat okolo os.

```

int DrawGLScene(GLvoid) // Vykreslování
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    bufferu
    glLoadIdentity(); // Reset matice

    glTranslatef(1.1f*float(cos(rot/16.0f)), 0.8f*float(sin(rot/20.0f)), -3.0f);
    glRotatef(rot, 1.0f, 0.0f, 0.0f); // Rotace na x
    glRotatef(rot*1.2f, 0.0f, 1.0f, 0.0f); // Rotace na y
    glRotatef(rot*1.4f, 0.0f, 0.0f, 1.0f); // Rotace na z

```

Přesuneme se trochu doleva, dolů a dopředu k vycentrování symbolu na každé ose, abychom simulovali také otáčení kolem vlastního centra (-0.35 je číslo, které pracuje ;) S tímto přesunem jsem si musel trochu pohrát, protože si nejsem jistý, jak je font široký, každé písmeno se víceméně liší. Nejsem si jistý, proč se fonty nevytvářejí kolem centrálního bodu.

```

    glTranslatef(-0.35f, -0.35f, 0.1f); // Vycentrování

```

Nakonec nakreslíme lebku a zkřížené kosti. Nechápete-li proč právě "N", tak si pusťte MS Word vyberte písmo Wingdings a napište "N" - odpovídá mu tento symbol. Aby se lebka pohybovala každým překreslením inkrementujeme rot.

```

    glPrint("N"); // Vykreslí lebku a zkřížené kosti
    rot+=0.1f; // Inkrementace rotace a pohybu

    return TRUE;
}

```

I když jsem nepopsal probíranou látku do žádných extrémních detailů, měli byste pochopit, jak požádat OpenGL o automatické generování texturových koordinátů. Neměli byste mít žádné problémy s otexturováním jakýchkoli objektů. Změnou pouhých dvou řádků kódu (viz. GL_SPHERE_MAP u vytváření textur), dosáhnete perfektního efektu sférického mapování (kovové odlesky světla).

**napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>**

Lekce 16 - Mlha

Tato lekce rozšiřuje použitím mlhy lekci 7. Naučíte se používat tři různých filtrů, měnit barvu a nastavit oblast působení mlhy (v hloubce). Velmi jednoduchý a "efektní" efekt.

Na začátek programu, za všechna #include, přidáme deklarace nových proměnných. Gp pro zjištění stisku klávesy G, ve filte" najdeme číslo 0 až 2, specifikující právě používaný texturový filtr. V poli fogMode[] ukládáme tři různé typy mlhy. Fogfilter určuje právě používanou mlhu. Ve fogColor je uložena šedá barva.

```
bool gp;// G stisknuto?
GLuint filter;// Určuje texturový filtr
GLuint fogMode[]={ GL_EXP, GL_EXP2, GL_LINEAR };// Tři typy mlhy
GLuint fogfilter= 0;// Která mlha se používá
GLfloat fogColor[4]= {0.5f, 0.5f, 0.5f, 1.0f};// Barva mlhy
```

Přesuneme se do funkce InitGL(). glClearColor(r,g,b,a) jsme vždy používali pro nastavení černého pozadí. Tentokrát uděláme malou změnu - použijeme šedé pozadí (barvu mlhy), protože vypadá lépe.

```
// Funkce InitGL()
glClearColor(0.5f,0.5f,0.5f,1.0f);// Šedá barva pozadí (stejná, jako má mlha)
```

Příkaz glFogi(GL_FOG_MODE, fogMode[fogfilter]) vybere typ filtru. Pro nás bude zatím nejjednodušší všechny možnosti vložit do pole a pak je voláním použít. Co tedy znamenají: **GL_EXP** - základní renderovaná mlha, která zahalí celou obrazovku; neposkytuje zrovna perfektní výsledek, ale odvádí slušnou práci na starších počítačích. **GL_EXP2** - další vývojový krok GL_EXP; opět zamlží celý monitor, ale tentokrát do větší hloubky. **GL_LINEAR** - nejlepší renderovací mód; objekty se mnohem lépe ztrácejí a vynořují

```
glFogi(GL_FOG_MODE, fogMode[fogfilter]);// Mód mlhy
glFogfv(GL_FOG_COLOR, fogColor);// Barva mlhy
glFogf(GL_FOG_DENSITY, 0.35f);// Hustota mlhy
```

O kvalitu mlhy se starat nebudeme, nicméně lze také použít GL_NICEST nebo GL_FASTEST. Nebudu je dále rozebírat - názvy mluví sami za sebe.

```
glHint(GL_FOG_HINT, GL_DONT_CARE);// Kvalita mlhy
glFogf(GL_FOG_START, 1.0f);// Začátek mlhy - v hloubce (osa z)
glFogf(GL_FOG_END, 5.0f);// Konec mlhy - v hloubce (osa z)
glEnable(GL_FOG);// Zapne mlhu
```

Ošetříme stisk klávesy 'G', kterou můžeme při běhu cyklovat mezi různými módy mlhy.

```
// Funkce WinMain()
if(keys['G'] && !gp)// Je stisknuto 'G'?
{
    gp=TRUE;
    fogfilter+=1;// Inkrementace fogfilter

    if(fogfilter>2)// Hlídá přetečení
    {
        fogfilter=0;
    }

    glFogi (GL_FOG_MODE, fogMode[fogfilter]);// Nastavení módu mlhy
}

if(!keys['G'])// Bylo uvolněno 'G'?
{
    gp=FALSE;
}
```

Hodně zajímavý, ale především totálně jednoduchý efekt. Celkem bezbolestně jsme se naučili používat mlhu v OpenGL programech.

napsal: Christopher Aliotta - Precursor <chris (zavináč) incinerated.com>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 17 - 2D fonty z textur

V této lekci se naučíte, jak vykreslit font pomocí texturou omapovaného obdélníku. Dozvíte se také, jak používat pixely místo jednotek. I když nemáte rádi mapování 2D znaků, najdete zde spoustu nových informací o OpenGL.

Tuším, že už vás asi fonty unavují. Textové lekce vás, nicméně nenaučili jenom "něco vypsát na monitor", naučili jste se také 3D fonty, mapování textur na cokoli a spoustu dalších věcí. Nicméně, co se stane pokud budete kompilovat projekt pro platformu, která nepodporuje fonty? Podíváte se do lekce 17... Pokud si pamatujete na první lekci o fontech (13), tak jsem tam vysvětloval používání textur pro vykreslování znaků na obrazovku. Obvykle, když používáte textury ke kreslení textu na obrazovku, spustíte grafický program, zvolíte font, napíšete znaky, uložíte bitmapu a "loadujete" ji do svého programu. Tento postup není zrovna efektivní pro program, ve kterém používáte hodně textů nebo texty, které se neustále mění. Ale jak to udělat lépe? Program v této lekci používá pouze JEDNU! texturu. Každý znak na tomto obrázku bude zabírat 16x16 pixelů. Bitmapa tedy celkem zabírá čtverec o straně 256 bodů (16*16=256) - standardní velikost. Takže... pojďme vytvořit 2D font z textury. Jako obvykle, i tentokrát rozvíjíme první lekci.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu

GLuint base;// Ukazatel na první z display listů pro font
GLuint texture[2];// Ukládá textury
GLuint loop;// Pomocná pro cykly

GLfloat cnt1;// Čítač 1 pro pohyb a barvu textu
GLfloat cnt2;// Čítač 2 pro pohyb a barvu textu
```

Následující kód je trochu odlišný, od toho z předchozích lekcí. Všimněte si, že TextureImage[] ukládá dva záznamy o obrázcích. Je velmi důležité zdvojit paměťové místo a loading. Jedno špatné číslo by mohlo zplodit přetečení paměti nebo totální error.

```
int LoadGLTextures()// Nahraje bitmapu a konvertuje na texturu
{
    int Status=FALSE;// Indikuje chyby
    AUX_RGBImageRec *TextureImage[2];// Alokuje místo pro bitmapy
```

Pokud byste zaměnili číslo 2 za jakékoli jiné, budou se dít věci. Vždy se musí rovnat číslu z předchozí řádky (tedy v TextureImage[]). Textury, které chceme nahrát se jmenují font.bmp a bumps.bmp. Tu druhou můžete zaměnit - není až tak podstatná.

```
    memset(TextureImage,0,sizeof(void *)*2);// Nastaví ukazatel na NULL

    if ((TextureImage[0]=LoadBMP("Data/Font.bmp")) && (TextureImage[1]=LoadBMP
("Data/Bumps.bmp")))
    {
        Status=TRUE;// Nastaví status na TRUE
```

Nebudu vám ani říkat kolik emailů jsem obdržel od lidí ptajících se: "Proč vidím jenom jednu texturu?" nebo "Proč jsou všechny moje textury bílé?!". Většinou bývá problém v tomto řádku. Opět pokud přepíšete 2 na 1, bude vidět jenom jedna textura (druhá bude bílá). A naopak, zaměníte-li 2 za 3, program se zhroutí. Příkaz glGenTextures() by se měl volat jenom jednou a tímto jedním voláním vytvořit najednou všechny textury, které hodláte použít. Už jsem viděl lidi, kteří tvořili každou texturu zvlášť. Je dobré, si vždy na začátku rozmyslet, kolik jich budete používat.

```
    glGenTextures(2, &texture[0]);// 2 textury
    for (loop=0; loop<2; loop++)
    {
```

```

        glBindTexture(GL_TEXTURE_2D, texture[loop]);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage
        [loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
    }
}

```

Na konci funkce uvolníme všechnu paměť, kterou jsme alokovali pro vytvoření textur. I zde si všimněte uvolňování dvou záznamů.

```

for (loop=0; loop<2; loop++)
{
    if (TextureImage[loop])// Pokud obrázek existuje
    {
        if (TextureImage[loop]->data)// Pokud existují data obrázku
        {
            free(TextureImage[loop]->data);// Uvolní paměť obrázku
        }
        free(TextureImage[loop]);// Uvolní strukturu obrázku
    }
}
return Status;
}

```

Teď vytvoříme font. Protože použijeme trochu matematiky, zaběhneme trochu do detailů.

```

GLvoid BuildFont(GLvoid)// Vytvoření display listů fontu
{

```

Jak už plyne z názvu, budou proměnné použity k určení pozice, každého znaku na textuře fontu.

```

    float cx;// Koordináty x
    float cy;// Koordináty y

```

Dále řekneme OpenGL, že chceme vytvořit 256 display listů. "base" ukazuje na první display list. Potom vybereme texturu.

```

    base=glGenLists(256);// 256 display listů
    glBindTexture(GL_TEXTURE_2D, texture[0]);// Výběr textury

```

Začneme cyklus generující všech 256 znaků.

```

    for (loop=0; loop<256; loop++)// Vytváří 256 display listů
    {

```

První řádka může vypadat trochu nejasně. Symbol % vyjadřuje celočíselný zbytek po dělení 16. Pomocí cx se budeme přesunovat na textuře po řádcích (zleva doprava), cy zajišťuje pohyb ve sloupcích (od shora dolů). Další operace $\%16.0f$ konvertuje výsledek do koordinátů textury. Pokud bude loop rovno 16 - cx bude rovno zbytku z $16/16$ tedy nule ($16/16=1$ zbytek 0). Ale cy bude výsledkem "normálního" dělení - $16/16=1$. Dále bychom se tedy měli na textuře přesunout na dalších řádek, dolů o výšku jednoho znaku a přesunovat se opět zleva doprava. loop se tedy rovná 17, $cx=17/16=1,0625$. Desetinná část (0,0625) je vlastně rovna jedné šestnáctině. Z toho plyne, že jsme se přesunuly o jeden znak doprava. cy je stále jedna (viz. dále). $18/16$ udává posun o 2 znaky doprava a jeden znak dolů. Analogicky se dostaneme k loop=32. cx bude rovno 0 ($32/16=2$ zbytek 0). cy=2, tím se na textuře posuneme o dva znaky dolů. Dává to smysl? (Pozn. překladatele: Já bych asi použil vnořený cyklus - vnějším jít po sloupcích a vnitřním po řádcích. Bylo by to pochopitelnější (...a snadnější na překlad :-))



```
cx=float(loop%16)/16.0f;// X pozice aktuálního znaku
cy=float(loop/16)/16.0f;// Y pozice aktuálního znaku
```

Ted, po troše matematického vysvětlování, začneme vytvářet 2D font. Pomocí cx a cy vyjmeme každý znak z textury fondu. Přičteme loop k hodnotě base - aby se znaky nepřepisovaly ukládáním vždy do prvního. Každý znak se uloží do vlastního display listu.

```
glNewList(base+loop, GL_COMPILE); // Vytvoření display listu
```

Po zvolení display listu do něj nakreslíme obdélník otexturovaný znakem.

```
glBegin(GL_QUADS); // Pro každý znak jeden obdélník
```

Cx a cy jsou schopny uložit velmi malou desetinnou hodnotu. Pokud cx a zároveň cy budou 0, tak bude příkaz vypadat takto: `glTexCoord2f(0.0f,1-0.0f-0.0625f)`; Pamatujte si, že 0,0625 je přesně 1/16 naší textury nebo šířka/výška jednoho znaku. Koordináty mohou ukazovat na levý dolní roh naší textury. Všimněte si, že používáme `glVertex2i(x,y)` namísto `glVertex3f(x,y,z)`. Nebudeme potřebovat hodnotu z, protože pracujeme s 2D fontem. Protože používáme kolnou projekci (ortho), nemusíme se přesunout do hloubky - stačí tedy pouze x, y. Okno má velikost 0-639 a 0-479 (640x480) pixelů, tudíž nemusíme používat desetinné nebo dokonce záporné hodnoty. Cesta jak nastavit ortho obraz je určit 0, 0 jako levý dolní roh a 640, 480 jako pravý horní roh. Zjednodušeně řečeno: zbavili jsme se záporných koordinát. Užitečná věc, pro lidi, kteří se nechtějí starat o perspektivu, a kteří více preferují práci s pixely než s jednotkami :)

```
glTexCoord2f(cx,1-cy-0.0625f); glVertex2i(0,0); // Levý dolní
```

Druhý koordinát je teď posunut o 1/16 doprava (šířka znaku) - přičteme k x-ové hodnotě 0,0625f.

```
glTexCoord2f(cx+0.0625f,1-cy-0.0625f); glVertex2i(16,0); // Pravý dolní
```

Třetí koordinát zůstává vpravo, ale přesunul se nahoru (o výšku znaku).

```
glTexCoord2f(cx+0.0625f,1-cy); glVertex2i(16,16); // Pravý horní
```

Určíme levý horní roh znaku.

```
glTexCoord2f(cx,1-cy); glVertex2i(0,16); // Levý horní
```

```
glEnd(); // Konec znaku
```

Přesuneme se o 10 pixelů doprava, tím se umístíme doprava od právě nakreslené textury. Pokud bychom se nepřesunuli, všechny znaky by se nakupily na jedno místo. Protože je font trošku "hubenější" (užší), nepřesuneme se o celých 16 pixelů (šířku znaku), ale pouze o 10. Mezi jednotlivými písmeny by byly velké mezery.

```
glTranslated(10,0,0); // Přesun na pravou stranu znaku
```

```
glEndList(); // Konec kreslení display listu
```

```
} // Cyklus pokračuje dokud se nevytvoří všech 256 znaků
```

```
}
```

Opět přidáme kód pro uvolnění všech 256 display listů znaku. Provede se při ukončování programu.

```
GLvoid KillFont(GLvoid) // Uvolní paměť fondu
{
    glDeleteLists(base,256); // Smaže 256 display listů
```

```
}
```

V následující funkci se provádí výstup textu. Všechno je pro vás nové, tudíž vysvětlím každou řádku hodně podrobně. Do tohoto kódu by mohla být přidána spousta dalších funkcí, jako je podpora proměnných, zvětšování znaků, rozestupy ap. Funkci `glPrint()` předáváme tři parametry. První a druhý je pozice textu v okně (u Y je nula dole!), třetí je žádaný řetězec a poslední je znaková sada. Podívejte se na bitmapu fontu. Jsou tam dvě rozdílené znakové sady (v tomto případě je první obyčejná - 0, druhá kurzívou - cokoli jiného).

```
GLvoid glPrint(GLint x, GLint y, char *string, int set)// Provádí výpis textu
{
```

Napřed se ujistíme, zda je `set` buď 1 nebo 0. Pokud je větší než 1, přiřadíme jí 0. (Pozn. překladatele: Autor asi zapomněl na častou obranu uživatelů při zhroutil programu: "Ne určitě jsem tam nezadal záporné číslo!" :-)

```
    if (set>1)
    {
        set=1;
    }
}
```

Protože je možné, že máme před spuštěním funkce vybranou (na tomto místě) "randomovou" texturu, zvolíme tu "fontovou".

```
    glBindTexture(GL_TEXTURE_2D, texture[0]);// Výběr textury
```

Vypneme hloubkové textování - blending vypadá lépe (text by mohl skončit za nějakým objektem, nemusí vypadat správně...). Okolí textu vám nemusí vadit, když používáte černé pozadí.

```
    glDisable(GL_DEPTH_TEST);// Vypne hloubkové testování
```

Hodně důležitá věc! Zvolíme projekční matici (Projection Matrix) a příkazem `glPushMatrix()` ji uložíme (něco jako paměť na kalkulačce). Do původního stavu ji můžeme obnovit voláním `glPopMatrix()` (viz. dále).

```
    glMatrixMode(GL_PROJECTION);// Vybere projekční matici
    glPushMatrix();// Uloží projekční matici
```

Poté, co byla projekční matice uložena, resetujeme matici a nastavíme ji pro kolmou projekci (Ortho screen). Parametry mají význam ořezávacích rovin (v pořadí): levá, pravá, dolní, horní, nejbližší, nejvzdálenější. Levou stranu bychom mohli určit na -640, ale proč pracovat se zápornými čísly? Je moudré nastavit tyto hodnoty, abyste si zvolili meze (rozlišení), ve kterých právě pracujete.

```
    glLoadIdentity();// Reset matice
    glOrtho(0,640,0,480,-1,1);// Nastavení kolmé projekce
```

Teď určíme matici modelview a opět voláním `glPushMatrix()` uložíme stávající nastavení. Poté resetujeme matici modelview, takže budeme moci pracovat s kolmou projekcí.

```
    glMatrixMode(GL_MODELVIEW);// Výběr matice
    glPushMatrix();// Uložení matice
    glLoadIdentity();// Reset matice
```

S uloženými nastaveními pro perspektivu a kolmou projekci, můžeme začít vykreslovat text. Začneme translací na místo, kam ho chceme vykreslit. Místo `glTranslatef()` použijeme `glTranslated()`, protože není důležitá desetinná hodnota. Nelze určit půlku pixelu :- (Pozn. překladatele: Tady bude asi jeden totálně velký error, jelikož `glTranslated()` pracuje v přesnosti double, tedy ještě ve větší - nicméně stane se. (Alespoň, že víme o co jde :-). Jo, ten smajlík u půlky pixelu byl i v původní verzi.)

```
    glTranslated(x,y,0);// Pozice textu (0,0 - levá dolní)
```

Řádek níže určí znakovou sadu. Při použití druhé přičteme 128 k display listu base (128 je polovina z 256 znaků). Přičtením 128 "přeskočíme" prvních 128 znaků.

```
    glListBase(base-32+(128*set));// Zvolí znakovou sadu (0 nebo 1)
```

Zbývá vykreslení. Jako pokaždé v minulých lekcích to provedeme i zde voláním `glCallLists()`. `strlen(string)` je délka řetězce (ve znacích), `GL_BYTE` znamená, že každý znak je reprezentován bytem (hodnoty 0 až 255). Nakonec, ve string předáváme konkrétní text pro vykreslení.

```
    glCallLists(strlen(string),GL_BYTE,string);// Vykreslení textu na obrazovku
```

Obnovíme perspektivní pohled. Zvolíme projekční matici a použijeme `glPopMatrix()` k odvolání se na dříve uložená (`glPushMatrix()`) nastavení.

```
    glMatrixMode(GL_PROJECTION);// Výběr projekční matice
    glPopMatrix();// Obnovení uložené projekční matice
```

Zvolíme matice modelview a uděláme to samé jako před chvílí.

```
glMatrixMode(GL_MODELVIEW); // Výběr matice modelview
glPopMatrix(); // Obnovení uložené modelview matice
```

Povolíme hloubkové testování. Pokud jste ho na začátku nevyprávěli, tak tuto řádku nepotřebujete.

```
glEnable(GL_DEPTH_TEST); // Zapne hloubkové testování
}
```

Vytvoříme textury a display listy. Pokud se něco nepovede vrátíme false. Tím program zjistí, že vznikl error a ukončí se.

```
int InitGL(GLvoid) // Všechno nastavení OpenGL
{
    if (!LoadGLTextures()) // Nahraje textury
    {
        return FALSE;
    }

    BuildFont(); // Vytvoří font
}
```

Následují obvyklé nastavení OpenGL.

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Černé pozadí
glClearDepth(1.0); // Nastavení hloubkového bufferu
glDepthFunc(GL_LEQUAL); // Typ hloubkového testování
glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Vybere typ blendingu
glShadeModel(GL_SMOOTH); // Povolí jemné stínování
glEnable(GL_TEXTURE_2D); // Zapne mapování 2D textur
return TRUE;
}
```

Začneme kreslit scénu - na začátku stvoříme 3D objekt a až potom text. Důvod proč jsem se rozhodl přidat 3D objekt je prostý: chci demonstrovat současné použití perspektivní i kolmé projekce v jednom programu.

```
int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice
}
```

Zvolíme texturu vytvořenou z bumps.bmp, přesuneme se o pět jednotek dovnitř a provedeme rotaci o 45° na ose Z. Toto pootočení po směru hodinových ručiček vyvolá dojem diamantu a ne dvou čtverců.

```
glBindTexture(GL_TEXTURE_2D, texture[1]); // Výběr textury
glTranslatef(0.0f, 0.0f, -5.0f); // Přesun o pět do obrazovky
glRotatef(45.0f, 0.0f, 0.0f, 1.0f); // Rotace o 45° po směru hodinových ručiček na ose
z
```

Provedeme další rotaci na osách X a Y, která je závislá na proměnné cnt1*30. Má za následek otáčení objektu dokola, stejně jako se otáčí diamant na jednom místě.

```
glRotatef(cnt1*30.0f, 1.0f, 1.0f, 0.0f); // Rotace na osách x a y
```

Protože chceme aby se jevil jako pevný, vypneme blending a nastavíme bílou barvu. Vykreslíme texturou namapovaný čtyřúhelník.

```
glDisable(GL_BLEND); // Vypnutí blendingu
glColor3f(1.0f, 1.0f, 1.0f); // Bílá barva

glBegin(GL_QUADS); // Kreslení obdélníku
    glTexCoord2d(0.0f, 0.0f);
    glVertex2f(-1.0f, 1.0f);
    glTexCoord2d(1.0f, 0.0f);
    glVertex2f( 1.0f, 1.0f);
    glTexCoord2d(1.0f, 1.0f);
    glVertex2f( 1.0f, -1.0f);
    glTexCoord2d(0.0f, 1.0f);
    glVertex2f(-1.0f, -1.0f);
glEnd(); // Konec obdélníku
```

Dále provedeme rotaci o 90° na osách X a Y. Opět vykreslíme čtyřúhelník. Tento nový uprostřed protíná prvně kreslený a je na něj kolmý (90°). Hezký souměrný tvar.

```

glRotatef(90.0f,1.0f,1.0f,0.0f); // Rotace na osách X a Y o 90°

glBegin(GL_QUADS); // Kreslení obdélníku
    glTexCoord2d(0.0f,0.0f);
    glVertex2f(-1.0f, 1.0f);
    glTexCoord2d(1.0f,0.0f);
    glVertex2f( 1.0f, 1.0f);
    glTexCoord2d(1.0f,1.0f);
    glVertex2f( 1.0f,-1.0f);
    glTexCoord2d(0.0f,1.0f);
    glVertex2f(-1.0f,-1.0f);
glEnd(); // Konec obdélníku

```

Zapneme blending a začneme vypisovat text. Použijeme stejné pulzování barev jako v některých minulých lekcích.

```

glEnable(GL_BLEND); // Zapnutí blendingu
glLoadIdentity(); // Reset matice

// Změna barvy založená na pozici textu
glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos
(cnt1+cnt2)));

```

Pro vykreslení stále využíváme funkci `glPrint()`. Prvními parametry jsou x-ová a Y-ová souřadnice, třetí atribut, "NeHe", bude výstupem a poslední určuje znakovou sadu (0-normální, 1-kurzíva). Asi jste si domysleli, že textem pohybujeme pomocí sinů a kosinů. Pokud jste tak trochu "v pasti", vraťte se do minulých lekcí, ale není podmínkou tomu až tak rozumět.

```

glPrint(int((280+250*cos(cnt1))),int(235+200*sin(cnt2)),"NeHe",0); // Vypíše text

glColor3f(1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt2)),1.0f*float(cos
(cnt1)));

glPrint(int((280+230*cos(cnt2))),int(235+200*sin(cnt1)),"OpenGL",1); // Vypíše text

```

Nastavíme barvu na modrou a na spodní část okna napíšeme jméno autora této lekce. Celé to zopakujeme s bílou barvou a posunutím o dva pixely doprava - jednoduchý stín (není-li zapnutý blending nebude to fungovat).

```

glColor3f(0.0f,0.0f,1.0f); // Modrá barva
glPrint(int(240+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0); // Vypíše text

glColor3f(1.0f,1.0f,1.0f); // Bílá barva
glPrint(int(242+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0); // Vypíše text

```

Inkrementujeme čítače - text se bude pohybovat a objekt rotovat.

```

    cnt1+=0.01f;
    cnt2+=0.0081f;
    return TRUE;
}

```

Myslím, že teď mohu oficiálně prohlásit, že moje tutoriály nyní vysvětlují všechny možné cesty k vykreslení textu. Kód z této lekce může být použit na jakékoli platformě, na které funguje OpenGL, je snadný k používání. Vykreslování tímto způsobem "užirá" velmi málo procesorového času. Rád bych poděkoval Guiseppu D'Agatovi za originální verzi této lekce. Hodně jsem ji upravil a konvertoval na nový základní kód, ale bez něj bych to asi nesvedl. Jeho verze má trochu více možností, jako vzdálenost znaků apod., ale já jsem zase stvořil "extrémně skvělý 3D objekt".

**napsal: Giuseppe D'Agata <waveform (zavináč) tiscalinet.it>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>**

Lekce 18 - Quadratics

Představuje se vám báječný svět quadraticů. Jedním řádkem kódu snadno vytváříte komplexní objekty typu koule, disku, válce ap. Pomocí matematiky a trochy plánování lze snadno morphovat jeden do druhého.

Quadratic (neznám český ekvivalent slova, takže zůstanu u původní verze) je jednoduchou cestou k vykreslení komplexních objektů. Na pozadí pracují na několika cyklech for a troše trigonometrie. Rozvineme kód z lekce 7, přidáme pár proměnných a aby byla také nějaká změna, použijeme jinou texturu

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu

bool light;// Světlo ON/OFF
bool lp;// L stisknuté?
bool fp;// F stisknuté?
bool sp;// Stisknutý mezerník?

int part1;// Začátek disku
int part2;// Konec disku
int p1=0;// Přírůstek 1
int p2=1;// Přírůstek 2

GLfloat xrot;// X rotace
GLfloat yrot;// Y rotace
GLfloat xspeed;// Rychlost x rotace
GLfloat yspeed;// Rychlost y rotace
GLfloat z=-5.0f;// Hloubka v obrazovce

GLUQuadricObj *quadratic;// Bude ukládat quadratic objekt

GLfloat LightAmbient[]={ 0.5f, 0.5f, 0.5f, 1.0f };// Okolní světlo
GLfloat LightDiffuse[]={ 1.0f, 1.0f, 1.0f, 1.0f };// Přímé světlo
GLfloat LightPosition[]={ 0.0f, 0.0f, 2.0f, 1.0f };// Pozice světla

GLuint filter;// Typ filtru
GLuint texture[3];// Místo pro 3 textury
GLuint object=0;// Určuje aktuálně vykreslovaný objekt
```

Přesuneme se do funkce InitGL(), kde inicializujeme quadratic. Na konec, ale před return, přidáme následující kód této lekce. (V prvním řádku vytvoříme nový quadratic (funkce na něj vrátí ukazatel, při chybě nulu). Aby světlo vypadalo opravdu perfektně nastavíme normálové vektory na GLU_SMOOTH (další možné hodnoty GLU_NONE a GLU_FLAT). Nakonec zapneme texturové mapování. Je celkem "neohrabané", protože nemůžeme naplánovat, co kam namapujeme - všechno se generuje automaticky.

```
quadratic=gluNewQuadric();// Vrátí ukazatel na nový quadratic
gluQuadricNormals(quadratic, GLU_SMOOTH);// Vygeneruje normálové vektory (hladké)
gluQuadricTexture(quadratic, GL_TRUE);// Vygeneruje texturové koordináty
```

Rozhodl jsem se, že původní krychli z lekce 7 nesmažu, ale že ji zde ponechám. Měli byste si uvědomit, že stejně jako mapujeme textury na námi vytvořený objekt, tak se úplně stejně mapují na quadratic objekty.

```
GLvoid glDrawCube();// Vykreslí krychli
{
    glBegin(GL_QUADS);
```

```

// Přední stěna
glNormal3f( 0.0f, 0.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
// Zadní stěna
glNormal3f( 0.0f, 0.0f,-1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Vrchní stěna
glNormal3f( 0.0f, 1.0f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
// Spodní stěna
glNormal3f( 0.0f,-1.0f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
// Pravá stěna
glNormal3f( 1.0f, 0.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// Levá stěna
glNormal3f(-1.0f, 0.0f, 0.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
}

```

Ve funkci DrawGLScene() se program větví podle druhu objektu, který chceme kreslit (kužel, válec, koule...). Do všech funkcí zajišťujících vykreslování (kromě naší krychle) se přidává parametr "quadratic".

```

int DrawGLScene (GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f,0.0f,z);
    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);

    glBindTexture(GL_TEXTURE_2D, texture[filter]); // Vybere texturu

    switch(object) // Vybere, co se bude kreslit
    {

```



```

    case 0:
        glDrawCube(); // Krychle
        break;

```

Dalším objektem bude válec. Prvním parametrem je spodní poloměr. Druhý určuje horní poloměr. Předáním rozdílných hodnot se vykreslí jiný tvar (zuzující trubka, popř. kužel). Třetí parametr specifikuje výšku/délku (vzdálenost základen). Čtvrtá hodnota značí množství polygonů "kolem" osy Z a pátá počet polygonů "na" ose Z. Například použitím 5 místo první 32 nevykreslíte válec, ale hranatou trubku, jejíž podstava je tvořena pravidelným pětiúhelníkem. Naopak rozdíl při záměně druhé 32 snad ani nepoznáte. Čím je těchto polygonů více, tím se zvětší kvalita (počet detailů) výstupu. Musím ale podtrhnout, že se program zpomalí. Snažte se vždy najít nějakou rozumnou hodnotu.



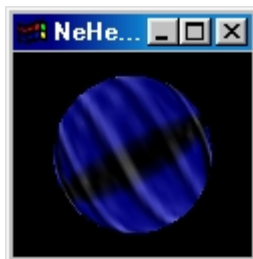
```
case 1:
    glTranslatef(0.0f,0.0f,-1.5f); // Vycentrování válce
    gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32); // Válec
    break;
```

Třetím vytvářeným objektem bude disk tvaru CD. První parametr určuje vnitřní poloměr - pokud zadáte nulu vykreslí se celistvý (bez středového kruhu). Druhu hodnotou je vnější poloměr (zadá-li se o málo větší než vnitřní vytvoříte prsten). Dejte si pozor, abyste nezadali vnější menší než vnitřní. Nespadne vám sice program, ale nic nevidíte. Třetím parametrem je počet plátků, jako když se krájí pizza. Čím jich bude více, tím budou okraje méně zubaté (např. zadáním 5 vykreslíte pravidelný pětiúhelník). Posledním předávaném číslo značí počet kružnic - analogie spirále na CD nebo gramofonové desce. Opět nemá moc velký vliv na kvalitu.



```
case 2:
    gluDisk(quadratic,0.5f,1.5f,32,32); // Disk ve tvaru CD
    break;
```

Následuje objekt, o kterém přemýšlíte v dlouhých bezesných nocích... koule. Stačí jedna funkce. Nejdůležitějším parametrem je poloměr - netřeba vysvětlovat. Pokud byste ale chtěli jít ještě dál, změňte před vykreslením měřítko jednotlivých os (`glScalef(x,y,z)`). Vytvoříte zaoblený tvar, který mi v první chvíli připomínal ozdobu na stromeček (šiška - zploštělá koule). Popř. zkuste zmenšit první 32 na 5. Vytvoříte hranatou (krychloidní :-o) kouli. Jak to popsat... kdybyste ji přes střed rozdělili rovinou, řezem bude pětiúhelník, ale druhým řezem kolmým na první bude stále koule.



```
case 3:
    // glScalef(1.0f,0.5f,1.0f); // Překl.: Změna měřítka
    gluSphere(quadratic,1.3f,32,32); // Koule
    // glScalef(1.0f,2.0f,1.0f); // Překl.: Obnovení měřítka
    break;
```

Už jsem trochu nakousl u válce, že kužel se vytváří téměř stejně. Předáte jeden poloměr rovný nule, tudíž se na jednom konci objeví špička.



```
case 4:
    glTranslatef(0.0f,0.0f,-1.5f); // Vycentrování kužele
    gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32); // Kužel
    break;
```

Šestý tvar vytvoříme příkazem `gluPartialDisk()`. Tento disk bude skoro stejný jako disk výše, nicméně další dva parametry funkce zajistí, že se nebude vykreslovat celý. Parametr `part1` specifikuje počáteční úhel, od kterého chceme kreslit a asi si domyslíte, že ten druhý určuje úhel, za kterým se už nic nevykreslí. Je vztažen k tomu prvnímu, takže pokud první nastavíme na 30° a druhý na 90° přestane se kreslit na $30^\circ + 90^\circ = 120^\circ$. My se rovnou pokusíme o "level 2" - zkusíme přidat jednoduchou animaci, kdy se disk bude překreslovat (po směru hodinových ručiček). Nejdříve zvyšujeme přírůstkový úhel. Jakmile dosáhne 360° (jeden oběh), začneme zvyšovat počáteční úhel - opět do 360° atd.



```
case 5:
    part1+=p1; // Inkrementace počátečního úhlu
    part2+=p2; // Inkrementace přírůstkového úhlu

    if(part1>359) // 360°
    {
        p1=0; // Zastaví zvětšování počátečního úhlu (part1+=0;)
        part1=0; // Vynulování počátečního úhlu
        p2=1; // Začne zvětšovat přírůstkový úhel
        part2=0; // Vynulování přírůstkového úhlu
    }

    if(part2>359) // 360°
    {
        p1=1; // Začne zvětšovat počáteční úhel
        p2=0; // Přestane zvětšovat přírůstkový úhel
    }

    gluPartialDisk(quadratic,0.5f,1.5f,32,32,part1,part2-part1); // Neúplný disk
    break;
};

xrot+=xspeed; // Inkrementace rotace
yrot+=yspeed; // Inkrementace rotace
return TRUE;
}
```

Přidáme ovládání klávesnic - pokud stisknete mezerník objekt se změní na následující v pořadí.

```
// Funkce WinMain()
if(keys[' '] && !sp) // Stisknutý mezerník?
{
    sp=TRUE;
    object++; // Cyklování objekty
    if(object>5) // Ošetření přetečení
        object=0;
}
if(!keys[' ']) // Uvolnění mezerníku?
```



```
{  
    sp=FALSE;  
}
```

Takže to je vše. Měli byste umět v OpenGL vykreslovat jakýkoli quadratic objekt. Pomocí morphingu a quadraticů se dá dosáhnout zajímavých efektů. Příkladem budiž námi animovaný disk.

napsal: GB Schmick - TipTup
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 19 - Částicové systémy

Chtěli jste už někdy naprogramovat exploze, vodní fontány, planoucí hvězdy a jiné skvělé efekty, nicméně kódování částicových systémů bylo buď příliš těžké nebo jste vůbec nevěděli, jak na to? V této lekci zjistíte, jak vytvořit jednoduchý, ale dobře vypadající částicový systém. Extra přidáme duhové barvy a ovládání klávesnicí. Také se dozvíte, jak pomocí triangle stripu jednoduše vykreslovat velké množství trojúhelníků.

V této lekci vytvoříme téměř komplexní částicový systém. Jakmile jednou pochopíte, jak pracují zvládnete cokoli.

Předem upozorňuji, že dodneška jsem nikdy nic podobného vytvářel. Vždy jsem si myslel, že ty slavné a "komerční" částicové systémy jsou hodně komplexním kusem kódu.

Možná nebudete věřit, když píš, že tento kód je 100% původní. Neměl jsem před sebou žádné technické dokumentace. Onehdy jsem prostě přemýšlel a náhle se mi v hlavě vygenerovala spousta nápadů. Namísto uvažování o částici jako o pixelu přesunujícím se z bodu A do bodu B a dělajícím to či ono jsem každé přiřadil vlastní objekt (strukturu) reagující na prostředí kolem. Zapouzdřuje život, stárnutí, barvu, rychlost, gravitační závislosti a další vlastnosti.

Takže ačkoli program, podle mě, vypadá perfektně a pracuje přesně, jak jsem chtěl, možná není tou správnou cestou k vytváření částicových systémů. Osobně jsem se nestaral, jak dobře pracuje, ale ve svých projektech jsem ho mohl bez problémů používat. Jestliže jste typem lidí, "šťouralů", kteří potřebují poznat správnou cestu, zkuste strávit hodiny prohledáváním internetu. Toto bylo varování.

Použijeme kód z lekce 1. Symbolická konstanta definuje počet vytvářených částic. Rainbow zapíná/vypíná cyklování mezi duhovými barvami. Sp a rp předcházejí opakování kódu při delším stisku mezerníku a enteru.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

#define MAX_PARTICLES 1000// Počet vytvářených částic

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu

bool rainbow = true;// Duhový efekt?
bool sp;// Stisknutý mezerník?
bool rp;// Stisknutý enter?
```

Následují pomocné proměnné. Slowdown kontroluje rychlost pohybu částic (čím vyšší číslo, tím pomaleji se pohybují). Xspeed a yspeed ovlivňují rychlost na jednotlivých osách. Jsou pouze jedním faktorem implementovaným kvůli ovládání klávesnicí. Zoom používáme pro přesun do/ze scény.

```
float slowdown=2.0f;// Zpomalení částic
float xspeed;// Základní rychlost na ose x
float yspeed;// Základní rychlost na ose y
float zoom=-40.0f;// Zoom
```

Loop využíváme především jako proměnnou cyklu, ve kterých inicializujeme a vykreslujeme částice. Col vychází ze slova color a značí barvu. Pomocí časovače delay při zapnutém duhovém módu cyklujeme mezi barvami.

Poslední proměnná je klasická textura. Rozhodl jsem se pro ni, protože vypadají mnohem lépe než jednobarevné body. Také si můžete vytvořit texturu ohně, sněhu, jakéhokoli objektu.

```
GLuint loop;// Řídící proměnná cyklů
GLuint col;// Vybraná barva
GLuint delay;// Zpoždění pro duhový efekt
GLuint texture[1];// Ukládá texturu
```

Následuje struktura definující vlastnosti částic. Obsahuje spoustu atributů, takže si je popíšeme. Pokud bude active rovno true částice bude aktivní a false logicky značí neaktivnost. V tomto programu se tato vlastnost nepoužívá, ale někdy jindy by mohla být užitečná. Life a fade definují, jak dlouho a jak jasně bude částice zobrazena. Od života (life) budeme odečítat stárnutí (fade). Na začátku je inicializujeme na random, stejně jako téměř všechny ostatní vlastnosti.

```
typedef struct// Vytvoří strukturu pro částici
{
    bool active;// Aktivní?
    float life;// Život
    float fade;// Rychlost stárnutí

    float r;// Červená složka barvy
    float g;// Zelená složka barvy
    float b;// Modrá složka barvy

    float x;// X Pozice
    float y;// Y Pozice
    float z;// Z Pozice

    float xi;// X směr a rychlost
    float yi;// Y X směr a rychlost
    float zi;// Z X směr a rychlost
}
```

Následující proměnné určují působení gravitace (každá ve své ose). Kladná xg značí působení doprava, záporná doleva. Směry jsou analogické ke směrům souřadnicových os.

```
float xg;// X gravitace
float yg;// Y gravitace
float zg;// Z gravitace

} particles;// Struktura částice
```

Dále deklarujeme pole datového typu particles (naše struktura) o velikosti MAX_PARTICLES a jménu particle.

```
particles particle[MAX_PARTICLES];// Pole částic
```

Inicializací pole barev si vytvoříme barevnou paletu. Každá z dvanácti položek obsahuje 3 RGB složky v rozmezí od červené do fialové.

```
static GLfloat colors[12][3]// Barevná paleta
{
    {1.0f,0.5f,0.5f},{1.0f,0.75f,0.5f},{1.0f,1.0f,0.5f},{0.75f,1.0f,0.5f},
    {0.5f,1.0f,0.5f},{0.5f,1.0f,0.75f},{0.5f,1.0f,1.0f},{0.5f,0.75f,1.0f},
    {0.5f,0.5f,1.0f},{0.75f,0.5f,1.0f},{1.0f,0.5f,1.0f},{1.0f,0.5f,0.75f}
};
```

Do inicializačního kódu jsem oproti kódu z první lekce přidal loading textury, nastavení blendingu a zadání počátečních hodnot částic.

```
int InitGL(GLvoid)// Všechna nastavení OpenGL
{
    if (!LoadGLTextures())// Nahraje textury
    {
        return FALSE;
    }

    glShadeModel(GL_SMOOTH);// Povolíme jemné stínování
    glClearColor(0.0f,0.0f,0.0f,0.0f);// Černé pozadí
    glClearDepth(1.0f);// Nastavení hloubkového bufferu
    glDisable(GL_DEPTH_TEST);// Vypne hloubkové testování
    glEnable(GL_BLEND);// Zapne blending
    glBlendFunc(GL_SRC_ALPHA,GL_ONE);// Typ blendingu
    glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);// Perspektiva
    glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);// Jemnost bodů
    glEnable(GL_TEXTURE_2D);// Zapne mapování textur
    glBindTexture(GL_TEXTURE_2D,texture[0]);// Vybere texturu
}
```

Inicializujeme jednotlivé částice. Začneme aktivováním. Pamatujte si, že naktivní nezobrazujeme a neaktualizujeme. Potom je oživíme. Nebyl jsem si jistý, zda je zhasínání (zprůhledňování) částic závislé na zkracování života, správnou cestou. Nicméně pracuje skvěle, tak co :-). Maximální život 1.0f dává nejjasnější vykreslení (viz. blending).

```
for (loop=0;loop<MAX_PARTICLES;loop++)// Inicializace částic
```

```

{
    particle[loop].active=true;// Aktivace
    particle[loop].life=1.0f;// Oživení

```

Na randomovou hodnotu nastavíme rychlost stárnutí a postupného zhasínání. Každým vykreslením se život (life) zkracuje o stárnutí (fade). Hodnotu 0 až 99 vydělíme 1000 a tím získáme velmi malé číslo. Aby rychlost stárnutí nikdy nebyla nulová, přičteme 0,003.

```

    particle[loop].fade=float(rand()%100)/1000.0f+0.003f;// Rychlost stárnutí

```

Nastavíme barvu částic na některou z výše vytvořené palety. Matematika je jednoduchá: vezmeme řídicí proměnnou cyklu a vynásobíme ji podílem počtu barev s celkovým počtem částic. Například při prvním průchodu bude loop = 0, po dosažení a výpočtu získáme $0 \cdot (12/1000) = 0$. Při posledním průchodu (loop = počet částic - 1 = 999) vyjde $999 \cdot (12/1000) = 11,988$. Protože předáváme int, výsledek se ořeže na 11, což je poslední barva v paletě.

```

    particle[loop].r=colors[loop*(12/MAX_PARTICLES)][0];// Červená
    particle[loop].g=colors[loop*(12/MAX_PARTICLES)][1];// Zelená
    particle[loop].b=colors[loop*(12/MAX_PARTICLES)][2];// Modrá

```

Inicializujeme směr a rychlost pohybu částic. Výpočet provedeme opět randomem, který pro počáteční efekt exploze násobíme deseti. Dostaneme kladná nebo záporná čísla určující směr a rychlost pohybu v jednotlivých osách.

```

    particle[loop].xi=float((rand()%50)-26.0f)*10.0f;// Rychlost a směr pohybu na
    ose x
    particle[loop].yi=float((rand()%50)-25.0f)*10.0f;// Rychlost a směr pohybu na
    ose y
    particle[loop].zi=float((rand()%50)-25.0f)*10.0f;// Rychlost a směr pohybu na
    ose z

```

Nakonec nastavíme gravitační působení. Většinou gravitace strhává věci dolů, ale ta naše bude moci působit všemi směry. Na začátku ovšem klasicky dolů (yg = - 0,8).

```

    particle[loop].xg=0.0f;// Gravitace na ose x
    particle[loop].yg=-0.8f;// Gravitace na ose y
    particle[loop].zg=0.0f;// Gravitace na ose z
}
return TRUE;
}

```

V další funkci se pokusíme o vykreslování, zajistíme působení gravitace ap. Matici ModelView resetujeme pouze jednou a to na začátku. Pozici částic tedy nebudeme určovat složitými posuny a rotacemi, ale pouze souřadnicemi předávanými funkcí glVertex3f().

```

int DrawGLScene(GLvoid)// Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Vymaže obrazovku a hloubkový
    buffer
    glLoadIdentity();// Reset matice
    for (loop=0;loop<MAX_PARTICLES;loop++)// Cyklus prochází každou částici
    {

```

První věcí je zkontrolování, zda je částice aktivní, pokud ne nebudeme ji aktualizovat ani vykreslovat. Nicméně v tomto programu budou aktivní všechny.

```

        if (particle[loop].active)// Pokud je částice aktivní
        {

```

Následující tři proměnné x,y,z jsou spíše pomocné k zpřehlednění kódu. Všimněte si, že k pozici na ose z přičítáme zoom, čímž můžeme jednoduše měnit hloubku v obrazovce.

```

            float x=particle[loop].x;// x pozice
            float y=particle[loop].y;// y pozice
            float z=particle[loop].z+zoom;// z pozice + zoom

```

Dále obarvíme částici její barvou. Jako alfa kanál (průhlednost) s výhodou využijeme život, který nabývá hodnot od 1.0f (plný) do 0.0f (smrt). Postupným stárnutím se tedy stává průhlednější až vybledne docela.

```

            // Barva částice
            glColor4f(particle[loop].r,particle[loop].g,particle[loop].b,particle
            [loop].life);

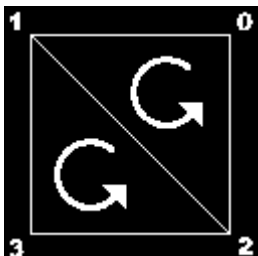
```

Máme pozici i barvu, takže přejdeme k vykreslení. Původně jsem chtěl použít otexturovaný čtverec, ale pak jsem se rozhodl pro otexturovaný "triangle strip". Většina grafických karet trojúhelníky mnohem rychleji než čtverce. Čtýřúhelníky

se často konvertují na dva trojúhelníky. K vykreslení klasickým způsobem bychom potřebovali 6 různých bodů, použitím triangle stripu pouze čtyři. Nejprve tedy požádáme OpenGL o vykreslení triangle stripu.

```
glBegin(GL_TRIANGLE_STRIP); // Vytvoří obdélník pomocí triangle strip
```

Triangle strip vykresluje sérii trojúhelníků užitím bodů V0, V1, V2, potom V2, v1, V3 (všimněte si pořadí), dále V2, V3, V4 atd. Tímto pořadím se zajistí, že všechny se vykreslí se stejnou orientací (viz. pořadí zadávání vrcholů), která je důležitá u některých operací, např. cullingu. Aby se něco vykreslilo musí být zadány alespoň tři body. Pro použití triangle stripu existují dva dobré důvody. První: po inicializaci prvního trojúhelníku stačí pro každý nový trojúhelník jenom jeden bod, který bude skombinován s body toho minulého. Druhý: odstraněním části kódu program poběží rychleji. Zdrojový kód bude kratší a přehlednější. Počet vykreslených trojúhelníků vykreslených na monitor bude o dva menší než počet zadaných bodů.



```
glTexCoord2d(1,1); glVertex3f(x+0.5f,y+0.5f,z); // Horní pravý
glTexCoord2d(0,1); glVertex3f(x-0.5f,y+0.5f,z); // Horní levý
glTexCoord2d(1,0); glVertex3f(x+0.5f,y-0.5f,z); // Dolní pravý
glTexCoord2d(0,0); glVertex3f(x-0.5f,y-0.5f,z); // Dolní levý
```

```
glEnd(); // Ukončí triangle strip
```

Po vykreslení přichází na řadu aktualizace částice. Matematika může vypadat strašně, ale je krásně jednoduchá. Vezmeme pozici na konkrétní ose a přičteme k ní pohyb na této ose vydělený slowdown krát tisíc. Např. pokud bude částice uprostřed obrazovky (0,0,0), pohyb xi 10 a slowdown 1 přesuneme ji o 10/(1*1000) - do bodu 0.01f na ose x. Pokud bychom inkrementovali slowdown na 2 přesuneme se pouze na 0.005f. Toto je také důvod násobení startovní hodnoty desítkou. Body se po spuštění programu pohybují mnohem rychleji, takže vytvoří dojem exploze.

```
particle[loop].x+=particle[loop].xi/(slowdown*1000); // Pohyb na ose x
particle[loop].y+=particle[loop].yi/(slowdown*1000); // Pohyb na ose y
particle[loop].z+=particle[loop].zi/(slowdown*1000); // Pohyb na ose z
```

Po spočítání pohybu aplikujeme gravitační působení. Docílíme toho přičtením "gravitační síly" k rychlosti pohybu. Řekněme, že rychlost pohybu je 10 a gravitace o velikosti 1 působí v opačném směru. Každým překreslením se rychlost pohybu dekrementováním zpomalí. Po deseti překresleních částice změní směr.

```
particle[loop].xi+=particle[loop].xg; // Gravitační působení na ose x
particle[loop].yi+=particle[loop].yg; // Gravitační působení na ose y
particle[loop].zi+=particle[loop].zg; // Gravitační působení na ose z
```

Snížíme hodnotu života o stárnutí. Kdybychom toto nedělali nikdy by částice neshořela. Každá má nastavenou jinou rychlost stárnutí, tudíž nezemřou ve stejný časový okamžik.

```
particle[loop].life-=particle[loop].fade; // Sníží život o stárnutí
```

V této chvíli musíme otestovat, zda je po zestárnutí stále naživu.

```
if (particle[loop].life<0.0f) // Pokud zemřela
{
```

Pokud zemřela "reinkarnujeme" ji nastavením plného života a nové náhodné rychlosti stárnutí.

```
particle[loop].life=1.0f; // Nový život
particle[loop].fade=float(rand()%100)/1000.0f+0.003f; // Náhodné stárnutí
```

Resetujeme její pozici na střed obrazovky.

```
particle[loop].x=0.0f; // Vycentrování doprostřed obrazovky
particle[loop].y=0.0f; // Vycentrování doprostřed obrazovky
particle[loop].z=0.0f; // Vycentrování doprostřed obrazovky
```

Určíme novou rychlost a vlastně i směr. Všimněte si, že jsem zvětšil maximální a minimální rychlost z 50 na 60 oproti funkci InitGL(), ale tentokrát výsledek nenásobím deseti. Už nechceme žádné exploze, ale pomalejší pohyb. Z důvodu ovládání klávesnicí přičítáme k hodnotě i globální rychlost (xspeed, yspeed).

```

particle[loop].xi=xspeed+float((rand()%60)-32.0f);// Nová rychlost a
směr
particle[loop].yi=yspeed+float((rand()%60)-30.0f);// Nová rychlost a
směr
particle[loop].zi=float((rand()%60)-30.0f);// Nová rychlost a směr

```

Částici přiřadíme také novou barvu. Proměnná col ukládá číslo 0 až 11 (12 barev). Pomocí ní vybíráme červenou, zelenou a modrou intenzitu z palety vytvořené na začátku programu.

```

particle[loop].r=colors[col][0];// Vybere barvu z palety
particle[loop].g=colors[col][1];// Vybere barvu z palety
particle[loop].b=colors[col][2];// Vybere barvu z palety
}

```

Následující kód aktualizuje působení gravitace. Stisknutím 8 na klávesnici zvětšíme yg (y gravitaci) a částice bude tažena vzhůru. Tato testování jsou vložena do vykreslování z důvodu zjednodušení. Kdyby bylo umístěno někde jinde museli bychom vytvořit nový cyklus dělající úplně stejnou práci. Podobné postupy poskytují skvělé možnosti. Např. se můžete pokusit o proud vody větrem vystřikující přímo vzhůru. Přidáním gravitace působící dolů vytvoříte fontánu vody.

```

// Pokud je stisknuta 8 a y gravitace je menší než 1.5
if (keys[VK_NUMPAD8] && (particle[loop].yg<1.5f)) particle[loop].yg+=0.01f;
// Pokud je stisknuta 2 a y gravitace je menší než -1.5
if (keys[VK_NUMPAD2] && (particle[loop].yg>-1.5f)) particle[loop].yg-=0.01f;
// Pokud je stisknuta 6 a x gravitace je menší než 1.5
if (keys[VK_NUMPAD6] && (particle[loop].xg<1.5f)) particle[loop].xg+=0.01f;
// Pokud je stisknuta 4 a x gravitace je menší než -1.5
if (keys[VK_NUMPAD4] && (particle[loop].xg>-1.5f)) particle[loop].xg-=0.01f;

```

Pro radost připišeme malou "vychytávku". Můj bratr si myslel, že úvodní výbuch je skvělý efekt. Stisknutím klávesy TAB se všechny částice resetují do centra obrazovky. Rychlost se vynásobí deseti a tím vytvoří explozi.

```

if (keys[VK_TAB])// Způsobí výbuch
{
particle[loop].x=0.0f;// Vycentrování na střed obrazovky
particle[loop].y=0.0f;// Vycentrování na střed obrazovky
particle[loop].z=0.0f;// Vycentrování na střed obrazovky
particle[loop].xi=float((rand()%50)-26.0f)*10.0f;// Náhodná rychlost
particle[loop].yi=float((rand()%50)-25.0f)*10.0f;// Náhodná rychlost
particle[loop].zi=float((rand()%50)-25.0f)*10.0f;// Náhodná rychlost
}
}
return TRUE;// Všechno OK
}

```

Funkci WinMain napíši celou, protože je v ní celkem dost změn.

```

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)
{
MSG msg;
BOOL done=FALSE;

if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
Fullscreen?", MB_YESNO|MB_ICONQUESTION) == IDNO)
{
fullscreen=FALSE;
}

if (!CreateGLWindow("NeHe's Particle Tutorial",640,480,16,fullscreen))
{
return 0;
}
}

```

Toto je první důležitá změna. Při rozhodnutí uživatele použít fullscreen změníme slowdown ze 2.0f na 1.0f. Tato úprava není až tak důležitá - lze ji vypustit. Slouží k urychlení fullscreenu - moje grafická karta pracuje v okně trochu rychleji. Nevím proč.

```

if (fullscreen)
{
slowdown=1.0f;
}

```

```

}
while(!done)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        if (msg.message==WM_QUIT)
        {
            done=TRUE;
        }
        else
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    else
    {
        if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
        {
            done=TRUE;
        }
        else
        {
            DrawGLScene();
            SwapBuffers(hdc);
        }
    }
}

```

Ošetříme vstup z klávesnice (+, -, PageUp, PageDown)

```

if (keys[VK_ADD] && (slowdown>1.0f)) slowdown-=0.01f;// Urychlení částic
if (keys[VK_SUBTRACT] && (slowdown<4.0f)) slowdown+=0.01f;// Zpomalení
částic
if (keys[VK_PRIOR])zoom+=0.1f;// Přiblížení pohledu
if (keys[VK_NEXT])zoom-=0.1f;// Oddálení pohledu

```

V následujících řádcích testujeme stisk enteru, abychom zapnuli cyklování barvami.

```

if (keys[VK_RETURN] && !rp)// Stisk enteru
{
    rp=true;// Nastaví příznak
    rainbow = !rainbow;// Zapne/vypne duhový efekt
}

if (!keys[VK_RETURN]) rp=false;// Po uvolnění vypne příznak

```

Operace při tisku mezerníku mohou být trochu matoucí. Stejně jako při enteru otestujeme, zda je zapnut duhový efekt. Pokud je, podíváme se jestli je hodnota počítadla counter větší než 25. Používá se ke změně barvy celých skupin částic. Pokud by se změnila barva při každém framu všechny částice by se obarvily jinak. Vytvořením zpoždění stihneme obarvit stejnou barvou více částic.

```

if ((keys[' '] && !sp) || (rainbow && (delay>25)))// Mezerník nebo
duhový efekt
{

```

Pokud je stisknut mezerník vypne se duhový efekt. Kdybychom ho nedeaktivovali, tak by se dokola měnily barvy dokud by nebyl stisknut enter. Dává smysl, že pokud člověk bouchá do mezerníku namísto do enteru, tak chce barvami procházet sám.

```

if (keys[' '])rainbow=false;// Pokud je stisknut vypne se duhový
mód

```

Pokud je mezerník stisknut nebo je zapnut duhový mód a zpoždění je větší než 25, přiřazením true do sp oznámíme počítači, že byl stisknut. Poté nastavíme delay na nulu, takže se může znovu počítat do 25. Nakonec inkrementujeme barvu na další v paletě.

```

sp=true;// Oznámí programu, že byl stisknut mezerník
delay=0;// Resetuje zpoždění duhových barev
col++;// Změní barvu částice

```

Protože máme pouze 12 barev musíme zamezit přetečení pole a následné zhroucení programu.

```

if (col>11) col=0;// Proti přetečení pole

```

```

    }

    if (!keys[' ']) sp=false;// Uvolnění mezerníku

```

Definujeme ovládání částic. Na začátku programu jsme deklarovali dvě proměnné rychlosti (xspeed, yspeed). Když částice vyhoří (zemře) přiřadíme jí novou rychlost závisející na těchto proměnných. Můžeme ovlivňovat jejich směr. Řádek dole testuje stisk šipky nahoru. V takovém případě yspeed inkrementujeme. Částice se bude pohybovat nahoru. Max rychlost je omezena na 200, větší už nevypadá dobře. Analogickým principem pracuje i ovládání ostatními šipkami.

```

    if (keys[VK_UP] && (yspeed<200)) yspeed+=1.0f;// Šipka nahoru
    if (keys[VK_DOWN] && (yspeed>-200)) yspeed-=1.0f;// Šipka dolů
    if (keys[VK_RIGHT] && (xspeed<200)) xspeed+=1.0f;// Šipka doprava
    if (keys[VK_LEFT] && (xspeed>-200)) xspeed-=1.0f;// Šipka doleva

```

Zbývá inkrementovat zpoždění delay, použité pro rychlost změn barev. Ostatní kód znáte z minulých lekcí.

```

    delay++;// Inkrementace zpoždění duhového efektu

    if (keys[VK_F1])
    {
        keys[VK_F1]=FALSE;
        KillGLWindow();
        fullscreen = !fullscreen;
        if (!CreateGLWindow("NeHe's Particle
        Tutorial",640,480,16,fullscreen))
        {
            return 0;
        }
    }
}

KillGLWindow();
return (msg.wParam);
}

```

V této lekci jsem se pokoušel o vysvětlení jednoduchého, ale působivého částicového systému. Jeho nejvýhodnější použití spočívá ve vytvoření efektů typu ohně, vody, sněhu, explozí, hvězd a spousty dalších. Jednoduchým modifikováním kódu lze snadno naprogramovat zcela nový efekt.

Děkuji Richardu Nutmanovi za upozornění, že by bylo výhodnější umisťovat částice použitím glVertex3f() namísto resetováním matice a složitými translacemi. Obě metody vypadají stejně, ale jeho verze snižuje zatížení počítače. Program běží rychleji.

**napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>**

Lekce 20 - Maskování

Černé okraje obrázků jsme dosud ořezávali blendingem. Ačkoli je tato metoda efektivní, ne vždy transparentní objekty vypadají dobře. Modelová situace: vytváříme hru a potřebujeme celistvý text nebo zakřivený ovládací panel, ale při blendingu scéna prosvítá. Nejlepším řešením je maskování obrázků.

Bitmapový formát obrázku je podporován každým počítačem a každým operačním systémem. Nejen, že se s nimi snadno pracuje, ale velmi snadno se nahrávají a konvertují na textury. K ořezání černých okrajů textu a obrázků jsme s výhodou používali blending, ale ne vždy výsledek vypadal dobře. Při spritové animaci ve hře nechcete, aby postavou prosvítalo pozadí. Podobně i text by měl být pevný a snadno čitelný. V takových situacích se s výhodou využívá maskování. Má dvě fáze. V první do scény umístíme černobílou texturu, ve druhé na stejné místo vykreslíme hlavní texturu. Použitý typ blendingu zajistí, že tam, kde se v masce (první obrázek) vyskytovala bílá barva zůstane původní scéna. Textura se neprůhledně vykreslí na černou barvu.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Masking ukládá příznak zapnutého/vypnutého maskování a podle scene se rozhodujeme, zda vykreslujeme první nebo druhou verzi scény. Loop je řídicí proměnná cyklů, roll použijeme pro rolování textur a rotaci objektu při zapnuté druhé scéně.

```
bool masking=TRUE;// Maskování on/off
bool mp;// Stisknuto M?
bool sp;// Stisknut mezerník?
bool scene;// Která scéna se má kreslit

GLuint texture[5];// Ukládá 5 textur
GLuint loop;// Řídicí proměnná cyklů

GLfloat roll;// Rolování textur
```

Generování textur je ve svém principu úplně stejné jako ve všech minulých lekcích, ale velmi přehledně demonstruje nahrávání více textur najednou. Téměř vždy jsme používali pouze jednu. Deklarujeme pole ukazatelů na pět bitmap, vynulujeme je a nahrajeme do nich obrázky, které vzápětí změníme na textury.

```
int LoadGLTextures()// Nahraje bitmapu a konvertuje na texturu
{
    int Status=FALSE;
    AUX_RGBImageRec *TextureImage[5];// Alokuje místo pro bitmapy
    memset(TextureImage,0,sizeof(void *)*5);

    if ((TextureImage[0]=LoadBMP("Data/logo.bmp")) &&// Logo
        (TextureImage[1]=LoadBMP("Data/mask1.bmp")) &&// První maska
        (TextureImage[2]=LoadBMP("Data/image1.bmp")) &&// První obrázek
        (TextureImage[3]=LoadBMP("Data/mask2.bmp")) &&// Druhá maska
        (TextureImage[4]=LoadBMP("Data/image2.bmp")))// Druhý obrázek
    {
        Status=TRUE;
        glGenTextures(5, &texture[0]);

        for (loop=0; loop<5; loop++)// Generuje jednotlivé textury
        {
            glBindTexture(GL_TEXTURE_2D, texture[loop]);
```

```

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage
        [loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
    }
}
for (loop=0; loop<5; loop++)
{
    if (TextureImage[loop])
    {
        if (TextureImage[loop]->data)
        {
            free(TextureImage[loop]->data);
        }
        free(TextureImage[loop]);
    }
}
return Status;
}
}

```

Z inicializace zůstala doslova kostra.

```

int InitGL(GLvoid) // Všechno nastavení OpenGL
{
    if (!LoadGLTextures()) // Nahraje textury
    {
        return FALSE;
    }

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Černé pozadí
    glClearDepth(1.0); // Povolí mazání Depth Bufferu
    glEnable(GL_DEPTH_TEST); // Zapne hloubkové testování
    glShadeModel(GL_SMOOTH); // Jemné stínování
    glEnable(GL_TEXTURE_2D); // Zapne mapování textur

    return TRUE;
}

```

Při vykreslování začneme jako obvykle mazáním bufferů, resetem matice a translací do obrazovky.

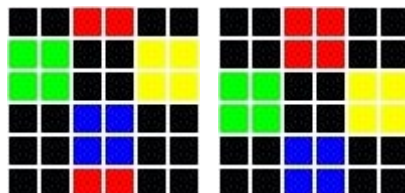
```

int DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f, 0.0f, -2.0f); // Přesun do obrazovky
}

```

Zvolíme texturu loga a namapujeme ji na obdélník. Koordináty vypadají nějak divně. Namísto obvyklých hodnot 0 až 1 tentokrát zadáme čísla 0 a 3. Předáním trojky oznámíme, že chceme namapovat texturu na polygon třikrát. Pro vysvětlení mě napadá vlastnost vedle sebe při umístění malého obrázku na plochu OS. Trojku zadáváme do šířky i do výšky, tudíž se na polygon rovnoměrně namapuje celkem devět stejných obrázků. Ke koordinátům také přičítáme (defakto odečítáme) proměnnou roll, kterou na konci funkce inkrementujeme. Vzniká dojem, že vykreslovaná hladina scény roluje, ale v programu se vlastně mění pouze texturové koordináty. Rolování může být použito pro různé efekty. Například pohybující se mraky nebo text létající po objektu.



```

glBindTexture(GL_TEXTURE_2D, texture[0]); // Výběr textury loga

```



```
glBegin(GL_QUADS); // Kreslení obdélníků
    glTexCoord2f(0.0f, -roll+0.0f); glVertex3f(-1.1f,-1.1f, 0.0f);
    glTexCoord2f(3.0f, -roll+0.0f); glVertex3f( 1.1f,-1.1f, 0.0f);
    glTexCoord2f(3.0f, -roll+3.0f); glVertex3f( 1.1f, 1.1f, 0.0f);
    glTexCoord2f(0.0f, -roll+3.0f); glVertex3f(-1.1f, 1.1f, 0.0f);
glEnd(); // Konec kreslení
```

Zapneme blending. Aby efekt pracoval musíme vypnout testování hloubky. Kdyby se nevypnulo největší pravděpodobností by nic nebylo vidět.

```
glEnable(GL_BLEND); // Zapne blending
glDisable(GL_DEPTH_TEST); // Vypne testování hloubky
```

Podle hodnoty proměnné se rozhodneme, zda budeme obrázek maskovat nebo použijeme mnohokrát vyzkoušený blending. Masky je černobílá kopie textury, kterou chceme vykreslit. Bílé oblasti masky budou průhledné, černé nebudou. Pod bílými sekcemi zůstane scéna nezměněna.

```
if (masking) // Je zapnuté maskování?
{
    glBlendFunc(GL_DST_COLOR, GL_ZERO); // Blending barvy obrazu pomocí nuly (černá)
}
```

Pokud bude scene true vykreslíme duhou, jinak první scénu.

```
if (scene) // Vykreslujeme druhou scénu?
{
```

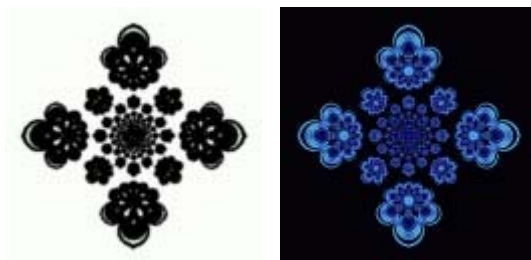
Nechceme objekty příliš velké, takže se přesuneme hlouběji do obrazovky. Provedeme rotaci na ose z o 0° až 360° podle proměnné roll.

```
glTranslatef(0.0f,0.0f,-1.0f); // Přesun o jednotku do obrazovky
glRotatef(roll*360,0.0f,0.0f,1.0f); // Rotace na ose z
```

Pokud je zapnuté maskování, vykreslíme nejdříve masku a potom objekt. Při vypnutém pouze objekt.

```
if (masking) // Je zapnuté maskování?
{
```

Nastavení blendingu pro masku jsme provedli dříve. Zvolíme texturu masky a namapujeme ji na obdélník. Po vykreslení se na scéně objeví černá místa odpovídající masce.



```
glBindTexture(GL_TEXTURE_2D, texture[3]); // Výběr textury druhé masky

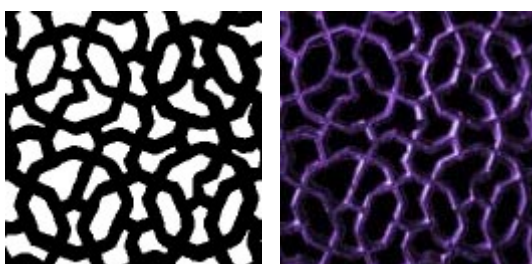
glBegin(GL_QUADS); // Začátek kreslení obdélníků
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f,-1.1f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f,-1.1f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f, 1.1f, 0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f, 1.1f, 0.0f);
glEnd(); // Konec kreslení
}
```

Znovu změníme mód blendingu. Oznáíme tím, že chceme vykreslit všechny části barevné textury, které NEJSOU

černé. Protože je obrázek barevnou kopií masky, tak se vykreslí jen místa nad černými částmi masky. Protože je maska černá, nic ze scény nebude prosvítat skrz textury. Vznikne dojem pevně vypadajícího obrázku. Zvolíme barevnou texturu. Poté ji vykreslíme se stejnými souřadnicemi bodů v prostoru a stejnými texturovými koordinátami jako masku. Kdybychom masku nevykreslili, obrázek by se zkopíroval do scény, ale díky blendingu by byl průhledný. Objekty za ním by prosvítaly.

```
glBlendFunc(GL_ONE, GL_ONE); // Pro druhý barevný obrázek
glBindTexture(GL_TEXTURE_2D, texture[4]); // Zvolí druhý obrázek
glBegin(GL_QUADS); // Začátek kreslení obdélníků
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f,  1.1f, 0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f,  1.1f, 0.0f);
glEnd(); // Konec kreslení
}
```

Při hodnotě FALSE uložené ve scene se vykreslí první scéna. Opět větvíme program podle maskování. Při zapnutém vykreslíme masku pro scénu jedna. Textura roluje zprava doleva (roll přičítáme k horizontálním koordinátám). Chceme, aby textura zaplnila celou scénu, takže neprovádíme translaci do obrazovky.



```
else // Vykreslení první scény
{
    if (masking) // Je zapnuté maskování?
    {
        glBindTexture(GL_TEXTURE_2D, texture[1]); // Výběr textury první masky
        glBegin(GL_QUADS); // Začátek kreslení obdélníků
            glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
            glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
            glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f,  1.1f, 0.0f);
            glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f,  1.1f, 0.0f);
        glEnd(); // Konec kreslení
    }
}
```

Blending nastavíme stejně jako minule. Vybereme texturu scény jedna a vykreslíme ji na stejné místo jako masku.

```
glBlendFunc(GL_ONE, GL_ONE); // Pro první barevný obrázek
glBindTexture(GL_TEXTURE_2D, texture[2]); // Zvolí první obrázek
glBegin(GL_QUADS); // Začátek kreslení obdélníků
    glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
    glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
    glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f,  1.1f, 0.0f);
    glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f,  1.1f, 0.0f);
glEnd(); // Konec kreslení
}
```

Zapneme testování hloubky a vypneme blending. V malém programu je to věc celkem zbytečná, ale u rozsáhlejších projektů někdy nevíte, co zrovna máte zapnuté nebo vypnuté. Tyto chyby se obtížně hledají a kradou čas. Po určité době ztrácíte orientaci, kód se stává složitějším - preventivní opatření.

```
glEnable(GL_DEPTH_TEST); // Zapne testování hloubky
glDisable(GL_BLEND); // Vypne blending
```

Aby se scéna dynamicky pohybovala musíme inkrementovat roll.

```
roll+=0.002f; // Inkrementace roll
if (roll>1.0f) // Je větší než jedna?
```

```

    {
        roll-=1.0f;// Odečte jedna
    }

    return TRUE;
}

```

Ošetříme vstup z klávesnice. Po stisku mezerníku změníme vykreslovanou scénu.

```

// Funkce WinMain()
    if (keys[' '] && !sp)// Mezerník - změna scény
    {
        sp=TRUE;
        scene=!scene;
    }

    if (!keys[' '])// Uvolnění mezerníku
    {
        sp=FALSE;
    }

```

Stiskem klávesy M zapneme, popř. vypneme maskování.

```

    if (keys['M'] && !mp)// Klávesa M - zapne/vypne maskování
    {
        mp=TRUE;
        masking=!masking;
    }

    if (!keys['M'])// Uvolnění klávesy M
    {
        mp=FALSE;
    }

```

Vytvoření masky není příliš těžké. Pokud máte originální obrázek již nakreslený, otevřete ho v nějakém grafickém editoru a transformujte ho do šedé palety barev. Po této operaci zvýšte kontrast, takže se šedé pixely ztmaví na černé. Zkuste také snížit jas ap. Je důležité, aby bílá byla opravdu bílá a černá čistě černá. Máte-li pochyby převedte obrázek do černobílého režimu (2 barvy). Pokud by v masce zůstaly šedé pixely byly by průhledné. Je také důležité, aby barevný obrázek měl černé pozadí a masku bílou. Otestujte si barvy masky kapátkem (většinou bývají chyby na rozhraní). Bílá je v RGB 255 255 255 (FF FF FF), černá 0 0 0.

Lze zjistit barvu pixelů při nahrávání bitmapy. Chcete-li pixel průhledný můžete mu přiřadit alfu rovnou nule. Všem ostatním barvám 255. Tato metoda také pracuje spolehlivě, ale vyžaduje extra kód. Tímto chci poukázat, že k výsledku existuje více cest - všechny mohou být správné.

Naučili jsme se, jak vykreslit část textury bez použití alfa kanálu. Klasický blending, který známe, nevypadal nejlépe a textury s alfa kanálem potřebují obrázky, které alfa kanál podporují. Bitmapy jsou vhodné především díky snadné práci, ale mají již zmíněné omezení. Tento program ukázal, jak obejít nedostatky bitmapových obrázků a vykreslování jedné textury vícekrát na jeden obdélník. Vše jsme rozšířili rolováním textur po scéně.

Děkuji Robu Santovi za ukázkový kód, ve kterém mi poprvé představil trik mapování dvou textur. Nicméně ani tato cesta není úplně dokonalá. Aby efekt pracoval, potřebujete dva průchody - dvakrát vykreslujete jeden objekt. Z toho plyne, že vykreslování tímto způsobem je dvakrát pomalejší. Nicméně... co se dá dělat?

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 21 - Přímky, antialiasing, časování, pravoúhlá projekce, základní zvuky a jednoduchá herní logika

První opravdu rozsáhlý tutoriál - jak už plyne z gigantického názvu. Doufejme, že taková spousta informací a technik dokáže udělat šťastným opravdu každého. Strávil jsem dva dny kódováním a kolem dvou týdnů psaním tohoto HTML souboru. Pokud jste někdy hráli hru Admiar, lekce vás vrátí do vzpomínek. Úkol hry sestává z vyplnění jednotlivých políček mřížky. Samozřejmě se musíte vyhnout všem nepřátelům.

Námět této lekce je vcelku složitý. Víím, že spousta z vás je unavena studiem základů. Každý by zemřel pro zvláštnosti 3D objektů, multitexturingu a podobně. Těmto lidem se omlouvám, protože chci zachovat postupné nabalování znalostí. Po velkém skoku vpřed není u krůčku zpět snadné udržet zájem čtenářů. Já osobně preferuji konstantní tempo. Možná jsem ztratil několik z vás, ale doufám, že ne příliš mnoho. Do dneška se ve všech mých tutoriálech objevovaly polygony, obdélníky a trojúhelníky. Pravděpodobně jste si všimli neúmyslné diskriminace :-)) čar, přímek, linek a podobných jednorozměrných útvarů. O několik hodin později začal vznikat Line Tutoriál. Vypadal v klidu, ale totálně nudný! Linky jsou skvělé, ale v porovnání s některými efekty nic moc. Shrnutí: rozhodl jsem se napsat multi-tutoriál. Na konci lekce bychom měli mít vytvořenu jednoduchou hru typu 'Admiar'. Misí bude vyplnit políčka mřížky. Hráče nesmí chytit nepřátelé - jak jinak. Implementujeme levely, etapy, životy, zvuky a kódy - k průchodu skrz levely, když se věci stanou příliš obtížnými. Ačkoli hru spustíte i na Pentiu 166 s Voodoo 2, rychlejší procesor nebude na škodu.

Rozšíříme standardní kód z lekce jedna. Přidáme potřebné hlavičkové soubory - stdio.h pro souborové operace a stdarg.h kvůli výstupu proměnných (level, obtížnost ap.).

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <stdarg.h>// Hlavičkový soubor pro funkce s proměnným počtem parametrů
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Deklarujeme proměnné. Pole vline ukládá záznamy o 121 vertikálních linkách, které tvoří mřížku. 11 přímek zleva doprava a 11 čas ze shora dolů. Hline ukládá 121 horizontálních přímek. Ap používáme ke zjištění stisku klávesy A. Filled je nastaveno na FALSE, jestliže mřížka není kompletně vyplněná a TRUE pokud je. Gameover ukončuje hru. Pokud se anti rovná TRUE je zapnut antialiasing objektů.

```
bool vline[11][10];// Ukládá záznamy o vertikálních linkách
bool hline[10][11];// Ukládá záznamy o horizontálních linkách
bool ap;// Stisknuto 'A'?
bool filled;// Bylo ukončeno vyplňování mřížky?
bool gameover;// Konec hry?
bool anti = TRUE;// Antialiasing?
```

Přicházejí na řadu celočíselné proměnné. Loop1 a loop2 užíváme k označení bodů v herní mřížce, zjištění zda do nás nepřítel nevrátil a k vygenerování randomové pozice. Zastavení pohybu nepřátel je implementováno čítačem delay. Po dosažení určité hodnoty se začnou znovu hýbat a delay se zpátky vynuluje.

Proměnná adjust je speciální. I když program obsahuje timer, tento timer pouze zjišťuje, zda je počítač (průběh programu) příliš rychlý a v takovém případě ho zpomalíme. Na grafické kartě GeForce hra běží hodně rychle. Po testu s PIII/450 s Voodoo 3500 TV si nelze nevíšimnout extrémní lenosti. Problém spočívá v kódu pro časování, který hru pouze zpomaluje. Zrychlení jím nelze provést. Vytvořil jsem proměnnou adjust, která může nabývat nuly až pěti. Čím vyšší hodnota, tím rychleji se objekty pohybují - podpora starších systémů. Nicméně nezáleží, jak rychlá je hra, absolutní rychlost provádění programu se nikdy nezvýší. Nastavením adjust na trojku vytvoříme kompromis pro pomalé i rychlé systémy. Více o časování dále.

Lives ukládá počet životů, level užíváme k zaznamenávání obtížnosti. Není to level, který se zobrazuje na monitoru. Level2 začíná se stejnou hodnotou, ale může být inkrementován donekonečna - záleží na obratnosti hráče. Pokud

dokáže dosáhnout třetího levelu, proměnná level se přestane zvyšovat. určuje pouze vnitřní obtížnost hry. Stage definuje konkrétní etapu hry.

```
int loop1;// Řídící proměnná cyklů
int loop2;// Řídící proměnná cyklů
int delay;// Doba zastavení nepřátel
int adjust = 3;// Rychlostní kompenzace pro pomalé systémy
int lives = 5;// Počet životů hráče
int level = 1;// Vnitřní obtížnost hry
int level2 = level;// Zobrazovaný level
int stage = 1;// Etapa/fáze hry
```

Definujeme strukturu objektu - hráč, nepřítel ap. Vnitřní proměnné fx a fy ukládají pomocnou polohu pro plynulý pohyb (fx = fine x). X a y definují pozici na mřížce. Mohou nabývat hodnot od nuly do deseti. Kdybychom se s hráčem po scéně pohybovali pomocí těchto dvou proměnných měli bychom jedenáct pozic vodorovně a jedenáct svisle. Hráč by přeskakoval z jednoho místa na druhé. Proto při pohybu používáme upřesňující fx a fy. Poslední proměnnou spin používáme pro otáčení objektů okolo osy z.

```
struct object// Struktura objektu ve hře
{
    int fx, fy;// Pohybová pozice
    int x, y;// Absolutní pozice
    float spin;// Otáčení objektu dokola
};
```

Na základě struktury vytvoříme hráče, devět nepřátel a jeden speciální objekt - skleněné přesýpací hodiny, které se sem tam objeví. Pokud je stihnete sebrat, nepřítel se na chvíli zastaví.

```
struct object player;// Hráč
struct object enemy[9];// Nepřátelé
struct object hourglass;// Skleněné hodiny
```

Abychom proměnné pro časovač měli pohromadě, sloučíme je do struktury. Frekvenci časovače deklarujeme jako 64-bitové celé číslo. Resolution je perioda (obrácená hodnota frekvence). Mm_timer_start a mm_timer_elapsed udržují počáteční a uplynulý čas. Používáme je pouze tehdy, pokud počítač nemá performance counter (v překladu: čítač provedení nebo výkonu, zůstanu u anglického termínu). Logická proměnná performance_timer bude nastavena na TRUE pokud program detekuje, že počítač má performance counter. Pokud ho nenajde budeme pro časování používat méně přesný, ale celkově dostačující multimedialní timer. Poslední dvě proměnné jsou opět 64-bitové integery, které ukládají čas spuštění a uplynulý čas performance counteru. Proměnnou na bázi této struktury pojmenujeme timer.

```
struct // Informace pro časovač
{
    __int64 frequency;// Frekvence
    float resolution;// Perioda
    unsigned long mm_timer_start;// Startovní čas multimedialního timeru
    unsigned long mm_timer_elapsed;// Uplynulý čas multimedialní timeru
    bool performance_timer;// Užíváme Performance Timer?
    __int64 performance_timer_start;// Startovní čas Performance Timeru
    __int64 performance_timer_elapsed;// Uplynulý čas Performance Timeru
} timer;// Struktura se jmenuje timer
```

Následující pole si můžeme představit jako tabulku rychlostí. objekt ve hře se může pohybovat rozdílnými rychlostmi. Vše závisí na proměnné adjust (výše). Pokud se její hodnota rovná nule pohybující se o pixel za určitý čas, pokud pět, rychlost činí dvacet pixelů. Inkrementováním adjust se na pomalých počítačích zvýší rychlost (ale i "trhanost") hry. Počet pixelů kroku je v tabulce. Adjust používáme jako index do tohoto pole.

```
int steps[6]={ 1, 2, 4, 5, 10, 20 };// Krokovací hodnota pro přizpůsobení pomalého videa
```

Deklarujeme pole dvou textur - pozadí a bitmapový font. Base ukazuje na první display list fontu (viz. minulé tutoriály). Funkce pro nahrávání a vytváření textur nebudu opisovat, byly tu už tolikrát, že je musíte znát na z paměť (překladatel).

```
GLuint texture[2];// Dvě textury
GLuint base;// Základní display list pro font
```

Inicializujeme časovač. Začneme vynulováním všech proměnných. Potom zjistíme, zda budeme moci používat performance counter. Pokud ano, uložíme frekvenci do timer.frequency, pokud ne budeme používat multimedialní timer - nastavíme timer.performance_timer na FALSE a načteme do počáteční hodnoty aktuální čas. Timer.resolution definujeme na 0.001 (Překladatel: dělení je celkem zbytečné) a timer.frequency na 1000. Protože ještě neuplynul žádný čas, přiřadíme uplynulému času startovní čas.

```
void TimerInit(void)// Inicializace timeru
{
```

```

memset(&timer, 0, sizeof(timer)); // Vynuluje proměnné struktury

// Zjistí jestli je Performance Counter dostupný a pokud ano, bude načtena jeho
frekvence
if (!QueryPerformanceFrequency((LARGE_INTEGER *) &timer.frequency))
{
    // Performance Counter není dostupný
    timer.performance_timer = FALSE; // Nastaví Performance Timer na FALSE
    timer.mm_timer_start = timeGetTime(); // Získání aktuálního času
    timer.resolution = 1.0f/1000.0f; // Nastavení periody
    timer.frequency = 1000; // Nastavení frekvence
    timer.mm_timer_elapsed = timer.mm_timer_start; // Uplynulý čas = počáteční
}

```

Má-li počítač performance counter projdeme touto větví. Nastavíme počáteční hodnotu a oznámíme, že můžeme používat performance counter. Poté spočítáme periodu pomocí frekvence získané v if() výše. Perioda je převrácená hodnota frekvence. Nakonec nastavíme uplynulý čas na startovní. Všimněte si, že místo sdílení proměnných obou timerů, jsem se rozhodl použít různé. Obě cesty by pracovaly, ale tato je přehlednější.

```

else
{
    // Performance Counter je možné používat
    QueryPerformanceCounter((LARGE_INTEGER *) &timer.performance_timer_start); //
    Počáteční čas
    timer.performance_timer = TRUE; // Nastavení Performance Timer na TRUE
    timer.resolution = (float) (((double)1.0f)/((double)timer.frequency)); //
    Spočítání periody
    timer.performance_timer_elapsed = timer.performance_timer_start; // Nastaví
    uplynulý čas na počáteční
}
}

```

V následující funkci načteme timer a vrátíme uplynulý čas v milisekundách. Deklarujeme 64-bitové celé číslo, do kterého načteme současnou hodnotu čítače. Opět větvíme program podle přítomnosti performance timeru. První řádkou v if() načteme obsah čítače. Dále od něj odečteme počáteční čas, který jsme získali při inicializaci časovače. Získaný rozdíl násobíme periodou čítače. Abychom výsledek v sekundách převedli na milisekundy násobíme ho tisícem. Tuto hodnotu vrátíme. Nepoužíváme-li performance counter, provede se větev else, která dělá analogicky to samé. Načteme současný čas, odečteme od něj počáteční, násobíme periodou a poté tisícem. Opět získáme uplynulý čas v milisekundách a vrátíme ho.

```

float TimerGetTime() // Získá čas v milisekundách
{
    __int64 time; // Čas se ukládá do 64-bitového integeru

    if (timer.performance_timer) // Performance Timer
    {
        QueryPerformanceCounter((LARGE_INTEGER *) &time); // Načte aktuální čas
        // Vrátí uplynulý čas v milisekundách
        return ((float)(time - timer.performance_timer_start) * timer.resolution)
        *1000.0f;
    }
    else // Multimedialní timer
    {
        // Vrátí uplynulý čas v milisekundách
        return ((float)(timeGetTime() - timer.mm_timer_start) * timer.resolution)
        *1000.0f;
    }
}

```

V další funkci se resetuje pozice hráče na levý horní roh a poloha nepřátel na randomové body. Levý horní roh scény má souřadnice [0;0]. Přiřadíme je hráčově x a y. Protože je na začátku linek, nepohybuje se, takže i upřesňující pohybové pozice nastavíme na nulu.

```

void ResetObjects(void) // Reset hráče a nepřátel
{
    player.x = 0; // Hráč bude vlevo nahoře
    player.y = 0; // Hráč bude vlevo nahoře
    player.fx = 0; // Pohybová pozice
    player.fy = 0; // Pohybová pozice
}

```

Přejdeme k inicializaci polohy nepřátel. Jejich aktuální počet (zobrazených) je roven vnitřnímu levelu násobenému jeho

současnou obtížností/etapou. Zapamatujte si, že maximální počet levelů je tři a maximální počet etap v levelu je také tři. Z toho plyne, že můžeme mít nejvíce devět nepřátel. V cyklu nastavíme x pozici každého nepřítele na pět až deset a y pozici na nula až deset. Nechceme, aby se pohybovali ze staré pozice na novou, takže se ujistíme, že se fx a fy budou rovnat x krát délka linky (60) a y krát výška linky (40).

```
for (loop1=0; loop1<(stage*level); loop1++)// Prochází nepřátele
{
    enemy[loop1].x = 5 + rand() % 6;// Nastaví randomovou x pozici
    enemy[loop1].y = rand() % 11;// Nastaví randomovou y pozici
    enemy[loop1].fx = enemy[loop1].x * 60;// Pohybová pozice
    enemy[loop1].fy = enemy[loop1].y * 40;// Pohybová pozice
}
}
```

Funkce glPrint() se moc nezměnila. Narozdíl od minulých tutoriálů jsem přidal možnost výpisu hodnot proměnných. Zapneme mapování textur, resetujeme matici a přesuneme se na určenou pozici. Pokud je zvolena první (nultá) znaková sada, změníme měřítko tak, aby byl font dvakrát vyšší a jeden a půl krát širší. Pomocí této finty budeme moci vypsat titul hry většími písmeny. Na konci vypneme mapování textur.

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...)// Výpis textů
{
    char text[256];// Bude ukládat výsledný řetězec
    va_list ap;// Ukazatel do argumentů funkce

    if (fmt == NULL)// Nebyl předán řetězec
        return;// Konec

    va_start(ap, fmt);// Rozdělí řetězec pro proměnné
    vsprintf(text, fmt, ap);// Konvertuje symboly na čísla
    va_end(ap);// Výsledek je uložen v text

    if (set>1)// Byla předána špatná znaková sada?
    {
        set=1;// Pokud ano, zvolí se kurzíva
    }

    glEnable(GL_TEXTURE_2D);// Zapne texturové mapování
    glLoadIdentity();// Reset matice
    glTranslated(x,y,0);// Přesun na požadovanou pozici
    glListBase(base-32+(128*set));// Zvolí znakovou sadu

    if (set==0)// Pokud je určena první znaková sada font bude větší
    {
        glScalef(1.5f,2.0f,1.0f);// Změna měřítka
    }

    glCallLists(strlen(text),GL_UNSIGNED_BYTE, text);// Výpis textu na monitor
    glDisable(GL_TEXTURE_2D);// Vypne texturové mapování
}
```

Implementace změny velikosti okna je nová. Namísto perspektivní scény použijeme pravoúhlu projekci (ortho view). Její hlavní charakteristikou je, že se při změně vzdálenosti pozorovatele (translace do hloubky) objekty nezmenšují - vypnutá perspektiva. Osa z je méně užitečná, někdy dokonce ztrácí význam. V tomto tutoriálu s ní nebudeme pracovat vůbec.

Začneme nastavením viewportu, úplně stejně, jako při perspektivní scéně. Poté zvolíme projekční matici (analogie filmovému projektoru; obsahuje informace, jak se zobrazí obrázek) a resetujeme ji.

Inicializujeme pravoúhlu projekci. První parametr 0.0f určuje pozici levé hrany scény. Druhá předávaná hodnota označuje polohu pravé hrany. Pokud by mělo okno velikost 640 x 480, tak ve width bude uložena hodnota 640. Scéna by začínala na ose x nulou a končila 640 - přesně jako okno. Třetím parametrem označujeme spodní okraj scény. Bývá záporný, ale protože chceme pracovat s pixely určíme spodek okna rovnu jeho výšce. Nula, čtvrtý parametr, definuje horní okraj. Poslední dvě hodnoty náleží k ose z. V této lekci se o ni nestaráme, takže nastavíme rozmezí od -1.0f do 1.0f. Všechno budeme vykreslovat v hloubce nula, takže uvidíme vše.

Po nastavení pravoúhlé scény, zvolíme matici modelview (informace o objektech, lokacích, atd.) a resetujeme ji.

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)// Inicializace a změna velikosti okna
{
    if (height==0)// Proti dělení nulou
    {
```

```

    height=1;// Výška se rovná jedné
}

glViewport(0,0,width,height);// Reset Viewportu

glMatrixMode(GL_PROJECTION);// Zvolí projekční matici
glLoadIdentity();// Reset projekční matice

glOrtho(0.0f,width,height,0.0f,-1.0f,1.0f);// Vytvoří pravoúhlu scénu

glMatrixMode(GL_MODELVIEW);// Zvolí matici modelview
glLoadIdentity();// Reset matice modelview
}

```

Při inicializaci se vyskytne několik nových příkazů. Začneme klasicky loadingem textur a kontrolou úspěšnosti této akce, poté vygenerujeme znakovou sadu fontu. Zapneme jemné stínování, nastavíme černé pozadí a vyčistíme hloubku jedničkou.

glHint() oznamuje OpenGL, jak má vykreslovat. V tomto případě požadujeme, aby všechny linky byly nejhezčí, jaké OpenGL dokáže vytvořit. Tímto příkazem zapínáme antialiasing. Také zapneme blending a zvolíme jeho mód tak, abychom umožnili, již zmíněný, antialiasing linek. Blending je potřeba, pokud chceme pěkně skombinovat (smíchat, zprůhlednit - blend with) s obrázkem na pozadí. Pokud chcete vidět, jak špatně budou linky vypadat, vypněte blending. Je důležité poukázat na fakt, že antialiasing se nemusí zobrazovat správně(? překl.). Objekty ve hře jsou docela malé, takže si nemusíte všimnout, že něco není v pořádku. Podívejte se pořádně. Všimněte si, jak se linky na nepřátelích zjemní pokud je antialiasing zapnutý. Hráč a hodiny by měli vypadat mnohem lépe.

```

int InitGL(GLvoid)// Nastavení OpenGL
{
    if (!LoadGLTextures())// Loading textur
    {
        return FALSE;
    }

    BuildFont();// Vytvoření fontu

    glShadeModel(GL_SMOOTH);// Zapne jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Černé pozadí
    glClearDepth(1.0f);// Nastavení hloubkového bufferu
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);// Nastavení antialiasingu linek
    glEnable(GL_BLEND);// Zapne blending
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);// Typ blendingu
    return TRUE;
}

```

Na řadu přichází vykreslování. Smažeme obrazovku a hloubkový buffer a zvolíme texturu fontu - texture[0]. Abychom slova "GRID CRAZY" vypsalí purpurovou barvou nastavíme R a G naplno, G s poloviční intenzitou. Nápis vypíšeme na souřadnice [207;24]. Použijeme první (nultou) znakovou sadu, takže bude text velkými písmeny. Poté zaměníme purpurovou barvu za žlutou a vypíšeme "Level" s obsahem proměnné level2. Dvojka v %2i určuje maximální počet číslic. Pomocí i oznamujeme, že se jedná o celočíselnou proměnnou (integer). O trochu níže, tou samou barvou, zobrazíme "Stage" s konkrétní etapou hry.

```

int DrawGLScene(GLvoid)// Všechno kreslení
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smaže obrazovku a hloubkový
    buffer
    glBindTexture(GL_TEXTURE_2D, texture[0]);// Zvolí texturu fontu

    glColor3f(1.0f,0.5f,1.0f);// Purpurová barva
    glPrint(207,24,0,"GRID CRAZY");// Vypíše logo hry

    glColor3f(1.0f,1.0f,0.0f);// Žlutá barva
    glPrint(20,20,1,"Level:%2i",level2);// Vypíše level
    glPrint(20,40,1,"Stage:%2i",stage);// Vypíše etapu
}

```

Zkontrolujeme konec hry. Pokud je gameover rovno TRUE zvolíme náhodnou barvu. Používáme glColor3ub(), protože je mnohem jednodušší vygenerovat číslo od 0 do 255 než od 0.0f do 1.0f. Doprava od titulku hry vypíšeme "GAME OVER" a o řádek níže "PRESS SPACE". Upozorňujeme hráče, že zemřel a že pomocí mezerníku může hru resetovat.

```

if (gameover)// Konec hry?
{
    glColor3ub(rand()%255,rand()%255,rand()%255);// Náhodná barva
    glPrint(472,20,1,"GAME OVER");// Vypíše GAME OVER
}

```

```

    glPrint(456,40,1,"PRESS SPACE");// Vypíše PRESS SPACE
}

```

Pokud mu však nějaké životy zbyly, zobrazíme doprava od titulku hry animované obrázky hráče. Vytvoříme cyklus, který jde od nuly do aktuálního počtu životů mínus jedna. Jedničku odečítáme, protože jeden obrázek se zobrazuje do hracího pole.

```

for (loop1=0; loop1<lives-1; loop1++)// Cyklus vykreslující životy
{

```

Uvnitř cyklu resetujeme matici a provedeme translaci doprava na pozici, kterou získáme výpočtem: 490 plus řídicí proměnná krát 40. Tímto způsobem budeme moci vykreslit každý animovaný život hráče o 40 pixelů doprava od minulého. Poté otáčujeme pohled proti směru hodinových ručiček v závislosti na hodnotě uložené v player.spin. Záporným znaménkem způsobíme, že se budou životy otáčet opačným směrem než hráč.

```

    glLoadIdentity();// Reset matice

    glTranslatef(490+(loop1*40.0f),40.0f,0.0f);// Přesun doprava od titulku
    glRotatef(-player.spin,0.0f,0.0f,1.0f);// Rotace proti směru hodinových ručiček

```

Zvolíme zelenou barvu a začneme zobrazovat. Kreslení linek je úplně stejné, jako kreslení polygonů. Začneme s glBegin(GL_LINES). Tím oznámíme OpenGL, že chceme kreslit přímky. Pro jednu stačí pouze dva body. My zadáváme body pomocí glVertex2d(), protože nepotřebujeme hloubku, ale samozřejmě lze použít i glVertex3f() pro plnohodnotný bod ve 3D prostoru.

```

    glColor3f(0.0f,1.0f,0.0f);// Zelená barva

    glBegin(GL_LINES);// Začátek kreslení životů

        glVertex2d(-5,-5);// Levý horní bod
        glVertex2d( 5, 5);// Pravý dolní bod
        glVertex2d( 5,-5);// Pravý horní bod
        glVertex2d(-5, 5);// Levý dolní bod

    glEnd();// Konec kreslení

```

Po vykreslení X (X - tvar hráče), znovu natočíme scénu, ale tentokrát pouze o polovinu úhlu. Zadáme tmavší zelenou barvu a vykreslíme +, ale trochu větší než X. Protože je + pomalejší a tmavší, X vypadá, jako by se otáčelo na jeho vrcholu.

```

    glRotatef(-player.spin*0.5f,0.0f,0.0f,1.0f);// Rotace proti směru hodinových
ručiček
    glColor3f(0.0f,0.75f,0.0f);// Tmavší zelená barva

    glBegin(GL_LINES);// Pokračování kreslení životů

        glVertex2d(-7, 0);// Levý středový bod
        glVertex2d( 7, 0);// Pravý středový bod
        glVertex2d( 0,-7);// Horní středový bod
        glVertex2d( 0, 7);// Dolní středový bod

    glEnd();// Konec kreslení
}

```

Nakreslíme herní mřížku. Nastavením proměnné filled na TRUE oznámíme programu, že už byla mřížka kompletně vyplněná (více dále). Určíme šířku čáry na 2.0f - linky ztloustnou a mřížka bude opticky více definovaná. Přestože se zhorší kvalita grafického výstupu, vypneme antialiasing. Velmi zatěžuje procesor a pokud nemáte hodně dobrou grafickou kartu, zaznamenáte obrovské zpomalení. Vyzkoušejte si a konejte, jak uznáte za vhodné.

```

filled=TRUE;// Před testem je všechno vyplněné
glLineWidth(2.0f);// Širší čáry
glDisable(GL_LINE_SMOOTH);// Vypne antialiasing
glLoadIdentity();// Reset matice

```

Po resetu matice deklarujeme dva vnořené cykly. Prvním procházíme mřížku zleva doprava a druhým ze shora dolů. Nastavíme barvu na modrou a pokud je právě kreslená linka již přejetá hráčem, přebijeme modrou barvu bílou. Dále zkontrolujeme, zda se nechystáme kreslit příliš vpravo. Pokud ano přeskochíme kreslení.

```

for (loop1=0; loop1<11; loop1++)// Cyklus zleva doprava
{
    for (loop2=0; loop2<11; loop2++)// Cyklus ze shora dolů
    {
        glColor3f(0.0f,0.5f,1.0f);// Modrá barva

```

```

if (hline[loop1][loop2])// Byla už linka přejetá?
{
    glColor3f(1.0f,1.0f,1.0f);// Bílá barva
}

if (loop1<10)// Nekreslit úplně vpravo
{

```

Otestujeme, jestli už byla horizontální linka přejetá. Pokud ne, přiřadíme do filled FALSE a tím oznámíme, že ještě nejméně jedna linka nebyla vyplněná, a tudíž ještě nemůžeme tento level opustit.

```

    if (!hline[loop1][loop2])// Nebyla linka ještě přejetá?
    {
        filled=FALSE;// Všechno ještě není vyplněno
    }

```

Poté konečně vykreslíme horizontální linku. Protože je vodorovná, přiřadíme y-ové hodnotě obou bodů stejnou velikost. Přičítáme sedmdesátku, aby nad hracím polem zůstalo volné místo pro informace o počtu životů, levelu ap. Hodnoty na ose x se liší tím, že druhý bod je posunut o šedesát pixelů doprava (80-20=60). Opět přičítáme konstantu, v tomto případě dvacítku, aby hrací pole nebylo namačkáno na levý okraj a vpravo nebyla zbytečná mezera. Všimněte si, že linky jsou kresleny zleva doprava. Toto je důvod, proč nechceme kreslit jedenáctou - nevešla by se na obrazovku.

```

    glBegin(GL_LINES);// Začátek kreslení horizontálních linek
        glVertex2d(20+(loop1*60),70+(loop2*40));// Levý bod
        glVertex2d(80+(loop1*60),70+(loop2*40));// Pravý bod
    glEnd();// Konec kreslení
}

```

Na řadu přicházejí vertikální linky. Kód je téměř stejný, takže text popisu nebudu zbytečně opisovat. Linky se kreslí ze shora dolů namísto zleva doprava - jediná odlišnost.

```

glColor3f(0.0f,0.5f,1.0f);// Modrá barva

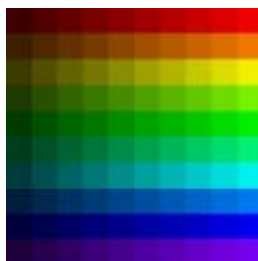
if (vline[loop1][loop2])// Byla už linka přejetá?
{
    glColor3f(1.0f,1.0f,1.0f);// Bílá barva
}

if (loop2<10)// Nekreslit úplně dolů
{
    if (!vline[loop1][loop2])// Nebyla linka ještě přejetá?
    {
        filled=FALSE;// Všechno ještě nebylo vyplněno
    }

    glBegin(GL_LINES);// Začátek kreslení vertikálních linek
        glVertex2d(20+(loop1*60),70+(loop2*40));// Horní bod
        glVertex2d(20+(loop1*60),110+(loop2*40));// Dolní bod
    glEnd();// Konec kreslení
}

```

Scéna je dohromady seskládaná z obdélníků o velikosti jedné desetiny obrázku scény. Na každý z nich je namapovaná určitá část velké textury, proto musíme zapnout mapování textur. Protože nechceme, aby měl kreslený obdélník barevný nádech, nastavíme barvu na bílou. Také nesmíme zapomenout zvolit texturu.



```

glEnable(GL_TEXTURE_2D);// Zapne mapování textur
glColor3f(1.0f,1.0f,1.0f);// Bílá barva
glBindTexture(GL_TEXTURE_2D, texture[1]);// Zvolí texturu

```

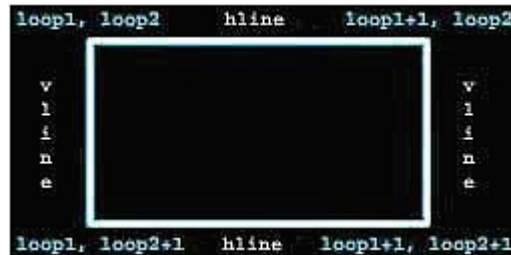
Dále prověříme, jestli aktuální obdélník ve scéně ještě existuje (není za hranou hrací plochy). Nacházíme se v cyklech, které postupně vykreslují 11 linek vodorovně a 11 svisle. Nicméně nevykreslujeme 11 obdélníků, ale pouze 10! Ověříme, jestli se nechystáme kreslit na jedenáctou pozici - loop1 i loop2 musí být menší než deset (0-9).

```

if ((loop1<10) && (loop2<10))// Pouze pokud je obdélník v hrací ploše
{

```

Zjistíme přejetí všech okolních linek obdélníku. Kraje testujeme v pořadí: horní, dolní, levý a pravý. Po každém průchodu vnitřním cyklem se inkrementuje loop1 a tím se z pravého okraje stává levý okraj následujícího obdélníku. V případě průchodu vnější smyčkou se ze spodních hran obdélníků v řádku stávají horní okraje nových obdélníků v řádku o jedno níže. Vše by mělo být zřejmé z diagramu.



Pokud jsou všechny okraje projety (rovnají se TRUE), můžeme namapovat texturu a vykreslit obdélník. Děláme to stejným stylem, jako jsme rozřezávali texturu znakové sady na jednotlivá písmena. Ani teď se neobejdeme bez matematiky. Dělíme loop1 i loop2 deseti, protože chceme rozdělit texturu mezi sto obdélníků (10x10). Koordináty jsou v rozmezí od nuly do jedné s krokem jedné desetiny (1/10=0,1).

Takže abychom dostali pravý horní roh, vydělíme hodnotu proměnných loop deseti a přičteme 0,1 k x-ovému koordinátu. Levý horní roh získáme dělením bez žádných dalších komplikací. Levý dolní bod spočívá opět v dělení deseti a přičtení 0,1 k ypsilonové složce. Dostáváme se k pravému dolnímu rohu, u kterého se po vydělení přičítá 0,1 k oběma souřadnicovým složkám. Doufám, že to dává smysl (Já taky - překl.).

Pokud budou oba loopy rovny devíti, ve výsledku dostaneme kombinaci 0,9 a 1,0, které dosadíme do parametrů funkce glTexCoord2f(x,y). souřadnice vrcholů obdélníků pro glVertex2d(x,y) získáme analogicky jako okraje linek mřížky. Přičítáme k nim, ale ještě konstanty (1, 59, 1, 39), které zajišťují zmenšení obdélníků - aby se vešly do políček mřížky a přitom nic nepřekryly.

```

// Jsou přejety všechny čtyři okraje obdélníku?
if (hline[loop1][loop2] && hline[loop1][loop2+1] && vline[loop1][loop2]
&& vline[loop1+1][loop2])
{
    glBegin(GL_QUADS); // Vykreslí otexturovaný obdélník

    glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float
(loop2/10.0f)));
    glVertex2d(20+(loop1*60)+59,(70+loop2*40+1)); // Pravý horní

    glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)));
    glVertex2d(20+(loop1*60)+1,(70+loop2*40+1)); // Levý horní

    glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)
+0.1f));
    glVertex2d(20+(loop1*60)+1,(70+loop2*40)+39); // Levý dolní

    glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float(loop2/10.0f)
+0.1f));
    glVertex2d(20+(loop1*60)+59,(70+loop2*40)+39); // Pravý dolní

    glEnd(); // Konec kreslení
}
}

```

V závěru vypneme mapování textur a po opuštění obou cyklů vrátíme šířku čáry na původní hodnotu.

```

    glDisable(GL_TEXTURE_2D); // Vypne mapování textur
}
}

glLineWidth(1.0f); // Šířka čáry 1.0f

```

V případě, že je anti rovno TRUE, zapneme zjemňování linek (antialiasing).

```

if (anti) // Má být zapnutý antialiasing?
{
    glEnable(GL_LINE_SMOOTH); // Zapne antialiasing
}

```

Abychom usnadnili hru, přidáme speciální objekt - přesýpací hodiny, jejichž sebráním se nepřítel na chvíli zastaví. Pro jejich umístění v hracím poli používáme proměnné x a y , nicméně protože se nebudou pohybovat, můžeme využít nepotřebné fx jako přepínač (0 jsou viditelné, 1 nejsou, 2 hráč je sebral). Fy implementujeme pro čítač, jak dlouho by měly být viditelné.

Začneme testem viditelnosti. Pokud se nemají zobrazit, přeskočíme vykreslení. Pokud ano, resetujeme matici a translaci je umístíme. Protože mřížka začíná na dvacítku, přičteme tuto hodnotu k $x*60$. Ze stejného důvodu na ose y přičítáme 70. Dále otáčujeme matici okolo osy z o úhel uložený v `hourglass.spin`. Před vykreslením ještě zvolíme náhodnou barvu.

```
if (hourglass.fx==1)// Hodiny se mají vykreslit
{
    glLoadIdentity();// Reset Matice

    glTranslatef(20.0f+(hourglass.x*60),70.0f+(hourglass.y*40),0.0f);// Umístění
    glRotatef(hourglass.spin,0.0f,0.0f,1.0f);// Rotace ve směru hodinových ručiček

    glColor3ub(rand()%255,rand()%255,rand()%255);// Náhodná barva
```

Pomocí `GL_LINES` oznámíme kreslení linek. Horní levý bod získáme odečtením pěti pixelů v obou směrech. Konec přímký leží pět pixelů směrem vpravo dolů od aktuální pozice. Druhou linku začneme vpravo nahoře a skončíme vlevo dole. Tvar písmene `X` doplníme o horní a dolní uzavírací linku.

```
glBegin(GL_LINES);// Vykreslení přesýpacích hodin

    glVertex2d(-5,-5);// Levý horní bod
    glVertex2d( 5, 5);// Pravý dolní bod
    glVertex2d( 5,-5);// Pravý horní bod
    glVertex2d(-5, 5);// Levý dolní bod

    glVertex2d(-5, 5);// Levý dolní bod
    glVertex2d( 5, 5);// Pravý dolní bod
    glVertex2d(-5,-5);// Levý horní bod
    glVertex2d( 5,-5);// Pravý horní bod

    glEnd();// Konec kreslení
}
```

Dále vykreslíme hráče. Opět resetujeme matici a určíme pozici ve scéně. Všimněte si, že pro jemný neskokový pohyb používáme fx a fy . Natočíme matici o uložený úhel, zvolíme světle zelenou barvu a pomocí linek vykreslíme tvar písmene `X`.

```
glLoadIdentity();// Reset Matice

glTranslatef(player.fx+20.0f,player.fy+70.0f,0.0f);// Přesun na pozici
glRotatef(player.spin,0.0f,0.0f,1.0f);// Rotace po směru hodinových ručiček

glColor3f(0.0f,1.0f,0.0f);// Zelená barva

glBegin(GL_LINES);// Vykreslení hráče

    glVertex2d(-5,-5);// Levý horní bod
    glVertex2d( 5, 5);// Pravý dolní bod
    glVertex2d( 5,-5);// Pravý horní bod
    glVertex2d(-5, 5);// Levý dolní bod

    glEnd();// Konec kreslení
```

Aby nevypadal až tak nudně, přidáme ještě tvar znaménka `+`, které se otáčí trochu rychleji, má tmavší barvu a je o dva pixely větší.

```
glRotatef(player.spin*0.5f,0.0f,0.0f,1.0f);// Rotace po směru hodinových ručiček

glColor3f(0.0f,0.75f,0.0f);// Tmavší zelená barva

glBegin(GL_LINES);// Pokračování kreslení hráče

    glVertex2d(-7, 0);// Levý středový bod
    glVertex2d( 7, 0);// Pravý středový bod
    glVertex2d( 0,-7);// Horní středový bod
    glVertex2d( 0, 7);// Dolní středový bod

    glEnd();// Konec kreslení
```

Ještě zbývá vykreslit nepřátele, takže se do nich pustíme. Deklarujeme cyklus procházející všechny nepřátele, kteří jsou viditelní v konkrétním levelu. Tento počet získáme vynásobením levelu s obtížností. Jejich maximální počet je devět. Uvnitř smyčky resetujeme matici a umístíme právě vykreslovaného nepřítele pomocí fx a fy (může se pohybovat). Zvolíme růžovou barvu a pomocí linek vykreslíme čtverec postavený na špičku, který nerotuje.

```
for (loop1=0; loop1<(stage*level); loop1++)// Vykreslí nepřátele
{
    glLoadIdentity();// Reset matice

    glTranslatef(enemy[loop1].fx+20.0f,enemy[loop1].fy+70.0f,0.0f);// Přesun na
    pozici
    glColor3f(1.0f,0.5f,0.5f);// Růžová barva

    glBegin(GL_LINES);// Vykreslení nepřátel

        glVertex2d( 0,-7);// Horní bod
        glVertex2d(-7, 0);// Levý bod
        glVertex2d(-7, 0);// Levý bod
        glVertex2d( 0, 7);// Dolní bod
        glVertex2d( 0, 7);// Dolní bod
        glVertex2d( 7, 0);// Pravý bod
        glVertex2d( 7, 0);// Pravý bod
        glVertex2d( 0,-7);// Horní bod

    glEnd();// Konec kreslení
```

Přidáme krvavě červené X, které se otáčí okolo osy z a poté ukončíme obrovskou vykreslovací funkci.

```
glRotatef(enemy[loop1].spin,0.0f,0.0f,1.0f);// Rotace vnitřku nepřítele

glColor3f(1.0f,0.0f,0.0f);// Krvavá barva

glBegin(GL_LINES);// Pokračování kreslení nepřátel

    glVertex2d(-7,-7);// Levý horní bod
    glVertex2d( 7, 7);// Pravý dolní bod
    glVertex2d(-7, 7);// Levý dolní bod
    glVertex2d( 7,-7);// Pravý horní bod

    glEnd();// Konec kreslení
}

return TRUE;// Konec funkce
}
```

Změn ve funkci WinMain() bude také trochu víc. Protože se jedná o hru, musíme ošetřit ovládání klávesnicí, časování a vše ostatní, co jsme dosud neudělali.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)
{
    MSG msg;
    BOOL done=FALSE;

    if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
Fullscreen?", MB_YESNO|MB_ICONQUESTION) == IDNO)
    {
        fullscreen=FALSE;
    }
}
```

Změníme titulok okna na "NeHe's Line Tutorial" a přidáme volání funkce ResetObjects(), která inicializuje pozici hráče na levý horní roh a nepřítelům předělí náhodné umístění, nejméně však pět políček od hráče. Poté zavoláme funkci pro inicializaci timeru.

```
if (!CreateGLWindow("NeHe's Particle Tutorial",640,480,16,fullscreen))
{
    return 0;
}

ResetObjects();// Inicializuje pozici hráče a nepřátel
TimerInit();// Zprovoznění timeru

while(!done)
```

```

{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        if (msg.message==WM_QUIT)
        {
            done=TRUE;
        }
        else
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    else
    {

```

Ted' zajistíme, aby pracoval kód pro časování. Předtím než vykreslíme scénu, nagrabujeme aktuální čas a uložíme jej do desetinné proměnné nazvané start. Potom vykreslíme scénu a prohodíme buffery.

```

float start=TimerGetTime();// Nagrabujeme aktuální čas
if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
{
    done=TRUE;
}
else
{
    SwapBuffers(hdc);
}

```

Vytvoříme časové zpoždění a to tak, že vkládáme prázdné příkazy tak dlouho, dokud je aktuální hodnota časovače (TimerGetTime()) menší než počáteční hodnota sečtená s rychlostí kroky hry krát dva. tímto velmi jednoduše zpomalíme OPRAVDU rychlé systémy.

Protože používáme krokování rychlosti (určené proměnnou adjust) program vždy poběží stejnou rychlostí. Například, pokud je hodnota kroku rovna jedné, měli bychom čekat dokud timer nebude větší nebo roven dvěma (2*1). Ale pokud zvětšíme rychlost kroku na dva (způsobí, že se hráč bude pohybovat o dvakrát tolik pixelů najednou), zpoždění se zvětší na čtyři (2*2). Ačkoli se pohybujeme dvakrát tak rychle, zpoždění trvá dvakrát déle a tudíž hra běží stejně rychle (ale více trhaně).

Spousta lidí jde ale jinou cestou než my. Je třeba brát v úvahu čas který uběhl mezi jednotlivými cykly ve kterých se renderuje. Na začátku každého cyklu se uloží aktuální čas, od kterého se odečte čas v minulém cyklu a tímto rozdílem se vydělí rychlost, kterou se má objekt pohybovat. Například: máme auto, které má jet rychlostí 10 jednotek za sekundu. Víme, že mezi tímto a předchozím cyklem uběhlo 20 ms. Objekt musíme tedy posunout o $10/(20*1000) = 0,0005$ jednotek. Bohužel v tomto programu to takto provést nemůžeme, protože používáme mřížku a ne např. otevřenou krajinu. Hodnoty fx a fy musí být přesně určené. Pokud hráčova fx bude řekněme 59 a počítač rozhodne posunout hráče o dva pixely doprava, tak po stisku šipky nahoru hráč nepůjde po "šedesátých pixelech", ale o kousek vedle.

Překl.: Nicméně i naše metoda má jeden velký error - okno nemůže v čekacích cyklech zpracovávat žádné zprávy. Řekněme, že bude (poněkud přezenu) časové zpoždění 5 sekund. Okno není aktivní a uživateli připadá, že v programu nastala fatální chyba. Pokusí se ho ukončit, ale i to se mu podaří až za těchto pět sekund. A pokud se bude zpomalovací kód volat častěji (např. po každém překreslení)... chápete? I u nás je tento problém trochu znatelný. Pokud se pokoušíte zatočit do určité linky, někdy se strefíte až na několikátý pokus - program nezareaguje včas. Proč vlastně vzniklo vícevláknové programování? Aby odstranilo zdánlivě "spadnuté programy" při náročných a dlouho trvajících výpočtech. Já osobně, bych se takovému časování za každou cenu vyhnul.

```

// Plýtvá cykly procesoru na rychlých systémech
while(TimerGetTime() < start + float(steps[adjust] * 2.0f))
{
}

if (keys[VK_F1])
{
    keys[VK_F1]=FALSE;
    KillGLWindow();
    fullscreen =! fullscreen;

    if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen))
    {
        return 0;
    }
}

```



```
}
```

Přejdeme k ovládání klávesnicí. Po stisku 'A' znegujeme proměnnou anti a tím oznámíme kódu pro kreslení, že má nebo nemá používat antialiasing.

```
if (keys['A'] && !ap)// Stisk A
{
    ap = TRUE;// Nastaví příznak
    anti=!anti;// Zapne/vypne antialiasing
}

if (!keys['A'])// Uvolnění A
{
    ap=FALSE;// Vypne příznak
}
```

Ted' pohyb a logika nepřátel. Chtěl jsem udržet kód opravdu jednoduchý, takže nečekejte žádné zázraky. Pracuje tak, že nepřítel zjistí, kde je hráč a poté se vydají jeho směrem (na pozici x, y). Mohou například vidět, že je v hracím poli nahoře, ale v čase, kdy testovali pozici x, hráč už může být díky fx úplně někde jinde. Častokrát se dostanou tam, kde byl o krok předtím. Někdy vypadají opravdu zmateně.

Začneme ujištěním se, jestli už není konec hry a jestli je okno aktivní. Pokud se například minimalizovalo, nepřítel se nebudou na pozadí pohybovat.

Vytvoříme cyklus, který i tentokrát prochází všechny nepřátele.

```
if (!gameover && active)// Není-li konec hry a okno je aktivní
{
    for (loop1=0; loop1<(stage*level); loop1++)// Prochází všechny nepřátele
    {
```

V případě, že bude x pozice nepřítele menší než x pozice hráče a zároveň se také musí rovnat y*40 pozici y (jsme v průsečíku vertikální a horizontální linky) posuneme nepřítele doprava. Analogickým způsobem implementujeme i pohyb doleva, nahoru a dolů.

Poznámka: po změně pozic x a y nelze vidět žádný pohyb, protože při vykreslování objekty umísťujeme pomocí proměnných fx a fy. Změnou x a y jenom určujeme požadovaný směr pohybu.

```
if ((enemy[loop1].x < player.x) && (enemy[loop1].fy==enemy
[loop1].y*40))
{
    enemy[loop1].x++;// Přesun o políčko doprava
}

if ((enemy[loop1].x > player.x) && (enemy[loop1].fy==enemy
[loop1].y*40))
{
    enemy[loop1].x--;// Přesun o políčko doleva
}

if ((enemy[loop1].y < player.y) && (enemy[loop1].fx==enemy
[loop1].x*60))
{
    enemy[loop1].y++;// Přesun o políčko dolů
}

if ((enemy[loop1].y > player.y) && (enemy[loop1].fx==enemy
[loop1].x*60))
{
    enemy[loop1].y--;// Přesun o políčko nahoru
}
```

Následující kód provádí opravdový pohyb. Zjistíme, zda je proměnná delay větší než tři minus level. Pokud jsme v levelu jedna, program pojde cyklem dvakrát (3-1=2), předtím než se nepřítel opravdu pohne. V levelu tři (nejvyšší možný) se nepřítel budou pohybovat stejnou rychlostí jako hráč - tedy bez zpoždění. Také ověřujeme, jestli se hourglass.fx nerovná dvěma. Tato proměnná označuje hráčovo sebrání přesýpacích hodin. V takém případě nepřítelem nepohybujeme.

Pokud je zpoždění vyšší než tři minus level a hráč nesebral hodiny, pohneme nepřítelem úpravou proměnných fx a fy. Nejprve vynulujeme zpoždění, takže ho budeme moci znovu počítat a potom opět deklarujeme cyklus, který prochází všechny viditelné nepřátele.

```
if (delay > (3-level) && (hourglass.fx!=2))// Hráč nesebral
```

```

přesýpací hodiny
{
    delay=0;// Reset delay na nulu

    for (loop2=0; loop2<(stage*level); loop2++)// Prochází všechny
nepřátele
    {

```

Nepřítel se vždy pohybuje pomocí fx/fy směrem k x/y. V prvním if zjistíme jestli je fx menší než x*60. V takovém případě ho posuneme doprava o vzdálenost steps[adjust]. Také změníme jeho úhel natočení, aby vznikl dojem rolování doprava.

Úplně stejně provedeme pohyby doleva, dolů a nahoru.

```

        if (enemy[loop2].fx < enemy[loop2].x*60)// Fx je menší než
        x
        {
            enemy[loop2].fx+=steps[adjust];// Zvýšit fx
            enemy[loop2].spin+=steps[adjust];// Rotace ve směru
            hodinových ručiček
        }

        if (enemy[loop2].fx > enemy[loop2].x*60)// Fx je větší než
        x
        {
            enemy[loop2].fx-=steps[adjust];// Snižit fx
            enemy[loop2].spin-=steps[adjust];// Rotace proti směru
            hodinových ručiček
        }

        if (enemy[loop2].fy < enemy[loop2].y*40)// Fy je menší než
        y
        {
            enemy[loop2].fy+=steps[adjust];// Zvýšit fy
            enemy[loop2].spin+=steps[adjust];// Rotace ve směru
            hodinových ručiček
        }

        if (enemy[loop2].fy > enemy[loop2].y*40)// Fy je větší než
        y
        {
            enemy[loop2].fy-=steps[adjust];// Snižit fy
            enemy[loop2].spin-=steps[adjust];// Rotace proti směru
            hodinových ručiček
        }
    }
}

```

Pohyb tedy máme. nyní potřebujeme vyřešit náraz nepřátel do hráče. V případě, že se obě fx i obě fy rovnají... hráč zemře. Dekrementujeme životy a v případě jejich nulové hodnoty prohlásíme hru za skončenou. Resetujeme všechny objekty a necháme zahrát úmrtní skladbu.

Zvuky jsou v našich tutoriálech novinkou. rozhodl jsem se použít tu nezákladnější dostupnou rutinu... PlaySound(). Předáváme jí tři parametry. První určuje cestu k souboru se zvukem. Druhý parametr pomocí nulového ukazatele ignorujeme. Třetí parametr je flag stylu. Dva nejčastěji používané jsou: SND_SYNC, který zastaví provádění programu, dokud přehrávání zvuku neskončí. Druhá možnost, SND_ASYNC, přehrává zvuk nezávisle na běhu programu. Dáme přednost maličkému zpoždění, takže funkci předáme SND_SYNC.

Na začátku tutoriálu jsem zapomněl na jednu věc: Abychom mohli používat funkci PlaySound(), potřebujeme inkludovat knihovnu WINMM.LIB (Windows Multimedia Library). Ve Visual C++ to lze provést v nabídce Project/Setting/Link.

```

// Setkání nepřítel s hráčem
if ((enemy[loop1].fx==player.fx) && (enemy[loop1].fy==player.fy))
{
    lives--;// Hráč ztrácí život

    if (lives==0)// Nulový počet životů
    {
        gameover=TRUE;// Konec hry
    }

    ResetObjects();// Reset pozice hráče a nepřátel

```

```

        PlaySound("Data/Die.wav", NULL, SND_SYNC); // Zahraje umíráček
    }
}

```

Ošetříme stisk kurzorových kláves. Vyřešíme šipku doprava, ostatní směry jsou zcela analogické. Abychom nevypadli pryč z hracího pole musí být `player.x` menší než deset (šířka mřížky). Nechceme, aby mohl změnit směr uprostřed přesunu a tak kontrolujeme, zda se `fx==player.x*60` a `fy==player.y*40`. Nastanou-li obě rovnosti, můžeme s určitostí říci, že se nachází v průsečíku rovnoběžné se svislou linkou a tedy dokončil svůj pohyb. Platí-li všechny podmínky, označíme linku pod hráčem jako přejetou a posuneme jej na následující pozici.

```

if (keys[VK_RIGHT] && (player.x<10) && (player.fx==player.x*60) &&
    (player.fy==player.y*40))
{
    hline[player.x][player.y]=TRUE; // Označení linky
    player.x++; // Doprava
}

if (keys[VK_LEFT] && (player.x>0) && (player.fx==player.x*60) &&
    (player.fy==player.y*40))
{
    hline[player.x][player.y]=TRUE; // Označení linky
    player.x--; // Doleva
}

if (keys[VK_DOWN] && (player.y<10) && (player.fx==player.x*60) &&
    (player.fy==player.y*40))
{
    vline[player.x][player.y]=TRUE; // Označení linky
    player.y++; // Dolů
}

if (keys[VK_UP] && (player.y>0) && (player.fx==player.x*60) &&
    (player.fy==player.y*40))
{
    vline[player.x][player.y]=TRUE; // Označení linky
    player.y--; // Nahoru
}

```

Hráče máme, dá se říci, přesunutého - ale pouze v programu! Je viditelný stále na stejném místě, protože ho vykreslujeme pomocí `fx` a `fy`. Provnáme, polohu `fx` vzhledem k `x` a pokud se nerovnají, snížíme vzdálenost meziminim o přesně daný úsek. Po několika překresleních se začnou obě hodnoty rovnat, což značí, že dokončil pohyb a nyní se nachází v průsečíku linek. Při následném stisku klávesy můžeme začít hráče znovu posunovat (viz. kód výše).

```

if (player.fx<player.x*60) // Fx je menší než x
{
    player.fx+=steps[adjust]; // Zvětší fx
}

if (player.fx>player.x*60) // Fx je větší než x
{
    player.fx-=steps[adjust]; // Zmenší fx
}

if (player.fy<player.y*40) // Fy je menší než y
{
    player.fy+=steps[adjust]; // Zvětší fy
}

if (player.fy>player.y*40) // Fy je větší než y
{
    player.fy-=steps[adjust]; // Zmenší fy
}
}

```

Nastane-li konec hry, projde program větví `else`. V ní je pouze test stisku mezerníku, který znovu spustí hru. Nastavíme `filled` na `TRUE` a díky tomu si program bude myslet, že je mřížka kompletně vyplněná - resetuje se pozice hráče i nepřátel. Abychom byli přesní, program si vlastně myslí, že jsme dokončili `level`, a proto inkrementuje do `stage` přiřazenou nulu na jedna. Přesně tohle chceme. Život vrátíme na počáteční hodnotu.

```

else// Jinak (if (!gameover && active))
{

```

```

        if (keys[' '])// Stisknutý mezerník
        {
            gameover = FALSE;// Konec hry
            filled = TRUE;// Mřížka vyplněná

            level = 1;// Level
            level2 = 1;// Zobrazovaný level
            stage = 0;// Obtížnost hry

            lives = 5;// Počet životů
        }
    }
}

```

Následující část testuje, zda je mřížka kompletně vyplněná. Filled může být nastaveno na TRUE celkem dvěma způsoby. Buď je mřížka úplně vyplněná, nebo skončila hra (zabitím hráče; nula životů) a uživatel stiskl mezerník, aby ji restartoval.

```

    if (filled)// Vyplněná mřížka?
    {

```

Ať už to způsobil kterýkoli případ je nám to celkem jedno. Vždy zahrajeme zvuk značící ukončení levelu. Už jsme jednou vysvětloval, jak PlaySound() pracuje. Předáním SND_SYNC vytvoříme časové zpoždění, kdy program čeká až zvuk dohraje.

```

        PlaySound("Data/Complete.wav", NULL, SND_SYNC);// Zvuk ukončení levelu

```

Potom inkrementujeme stage a zjistíme, jestli není větší než tři. Pokud ano, vrátíme ho na jedno, zvětšíme vnitřní i zobrazovaný level o jedničku.

```

        stage++;// Inkrementace obtížnosti
        if (stage > 3)// Je větší než tři?
        {
            stage=1;// Reset na jedničku
            level++;// Zvětší level
            level2++;// Zvětší zobrazovaný level

```

Pokud bude vnitřní level větší než tři, vrátíme ho zpět na trojku a přidáme hráči jeden život, ale pouze do maximálních pěti. Více živý nikdy nebude.

```

            if (level>3)// Je level větší než tři?
            {
                level=3;// Vrátil ho zpátky na tři
                lives++;// Život navíc

                if (lives > 5)// Má víc životů než pět?
                {
                    lives = 5;// Maximální počet životů pět
                }
            }
        }
    }
}

```

Resetujeme všechny objekty ve hře (hráč, nepřátelé) a vynulujeme flag projetí všech linek na FALSE. Pokud bychom to neudělali, další level by byl předčasně ukončen - program by opět skočil do tohoto kódu. Mimochodem, je úplně stejný jako kód pro vykreslování mřížky.

```

    ResetObjects();// Reset pozice hráče a nepřátel
    for (loop1=0; loop1<11; loop1++)// Cyklus skrz x koordináty mřížky
    {
        for (loop2=0; loop2<11; loop2++)// Cyklus skrz y koordináty mřížky
        {
            if (loop1 < 10)// X musí být menší než deset
            {
                hline[loop1][loop2] = FALSE;// Nulování
            }

            if (loop2 < 10)// Y musí být menší než deset
            {
                vline[loop1][loop2] = FALSE;// Nulování
            }
        }
    }
}

```

```
}
```

Pokusíme se implementovat hráčovo sebrání přesýpacích hodin. Že si musí polohy odpovídat je, myslím si, jasné. Nicméně přidáváme ještě podmínku `hourglass.fx==1`. Nejedná se o žádnou polohu. `Fx` používáme jako indikátor toho, že jsou zobrazené na monitoru.

```
// Hráč sebral přesýpací hodiny
if ((player.fx==hourglass.x*60) && (player.fy==hourglass.y*40) &&
    (hourglass.fx==1))
{
```

Necháme zahrát zvuk zmrazení. Aby zvuk zněl na pozadí, používáme `SND_ASYNC`. Díky OR-ování se symbolickou konstantou `SND_LOOP` docílíme toho, že se po dokončení přehrávání zvuku sám znovu spustí. Zastavit ho můžeme buď požadavkem na zastavení, nebo přehráním jiného zvuku.

Aby hodiny nebyly dále zobrazené nastavíme `fx` na dva. Také přiřadíme do `fy` nulu. `Fy` je něco jako čítač, který inkrementujeme do určité hodnoty, po jejímž přetečení změníme hodnotu `fx`.

```
    PlaySound("Data/freeze.wav", NULL, SND_ASYNC | SND_LOOP); // Zvuk
    zmrazení

    hourglass.fx=2; // Skryje hodiny
    hourglass.fy=0; // Nuluje čítač
}
```

Následující kód zajišťuje narůstání rotace hráče o polovinu nižší rychlostí než má hra. V případě, že bude hodnota vyšší než 360° odečteme 360. Tím zajistíme, aby nebyla moc vysoká.

```
player.spin += 0.5f * steps[adjust]; // Rotace hráče
if (player.spin > 360.0f) // Úhel je větší než 360°
{
    player.spin -= 360; // Odečte 360
}
```

Aby se hodiny točily opačným směrem než hráč, namísto zvyšování, úhel snižujeme. Rychlost je čtvrtinová oproti rychlosti hry. Opět ošetříme podtečení proměnné.

```
hourglass.spin -= 0.25f * steps[adjust]; // Rotace přesýpacích hodin
if (hourglass.spin < 0.0f) // Úhel je menší než 0°
{
    hourglass.spin += 360.0f; // Přičte 360
}
```

Zvětšíme hodnotu čítače přesýpacích hodin, o které jsme mluvili před chvílí. Opět podle rychlosti hry. Dále zjistíme, jestli se `hourglass.fx` rovná nule (nejsou zobrazené) a zároveň jestli je čítač větší než 6000 děleno `level`. V takovém případě přehrajeme zvuk zobrazení, vygenerujeme novou pozici a přes `fx=1` hodiny zobrazíme. Vynulujeme čítač, aby mohl počítat znovu.

```
hourglass.fy += steps[adjust]; // Zvětšení hodnoty čítače přesýpacích hodin
if ((hourglass.fx==0) && (hourglass.fy > 6000/level)) // Hodiny jsou skryté a
přetekl čítač
{
    PlaySound("Data/hourglass.wav", NULL, SND_ASYNC); // Zvuk zobrazení hodin

    hourglass.x = rand()%10+1; // Náhodná pozice
    hourglass.y = rand()%11; // Náhodná pozice

    hourglass.fx = 1; // Zobrazení hodin
    hourglass.fy = 0; // Nulování čítače
}
```

Překli-li čítač v době, kdy jsou hodiny viditelné (`fx==1`), schováme je a opět vynulujeme čítač.

```
if ((hourglass.fx==1) && (hourglass.fy > 6000/level)) // Hodiny jsou zobrazené
a přetekl čítač
{
    hourglass.fx = 0; // Skrýt hodiny
    hourglass.fy = 0; // Nulování čítače
}
```

Při hráčově sebrání hodin jsme zmrazili všechny nepřátele. Nyní je rozmrazíme. `Fx==2` indikuje, že byly hodiny sebrány.

Fy porovnávané s vypočtenou hodnotou. Jsou-li obě podmínky pravdivé, vypneme zvuk, který zní ve smyčce na pozadí a to tak, že přehrajeme nulový zvuk. Zneviditelníme hodiny a vynulujeme jejich čítač.

```
if ((hourglass.fx==2) && (hourglass.fy>500+(500*level)))// Nepřátelé
zmrazení a přetekl čítač
{
    PlaySound(NULL, NULL, 0);// Vypne zvuk zmrazení

    hourglass.fx = 0;// Skrýt hodiny
    hourglass.fy = 0;// Nulování čítače
}
```

Na samém konci hlavní smyčky programu inkrementujeme proměnnou delay. To je, myslím si, vše.

```
    delay++;// Inkrementuje čítač zpoždění nepřátel
}

KillGLWindow();// Zruší okno
return (msg.wParam);// Ukončí program
}
```

Psaním tohoto tutoriálu jsem strávil spoustu času. Začínal jako zcela jednoduchý tutoriál o linkách, který se úplně nečekaně rozvinul v menší hru. Doufejme, že budete moci ve svých programech využít vše, co jste se zde naučili. Víím, že se spousta z vás ptala po hře s kostičkami a políčky. Nemohli jste dostat více kostičkovatější a více políčkovatější hru než je tato. Ačkoli lekce nevysvětluje mnoho nových věcí o OpenGL, myslím si, že časování a zvuky jsou také důležité - zvlášť ve hrách. Co ještě napsat? Asi nic...

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 22 - Bump Mapping & Multi Texturing

Pravý čas vrátit se zpátky na začátek a začít si opakovat. Nováčkům v OpenGL se absolutně nedoporučuje! Pokud, ale máte odvalu, můžete zkusit dobrodružství s nadupanou grafikou. V této lekci modifikujeme kód z šesté lekce, aby podporoval hardwarový multi texturing přes opravdu skvělý vizuální efekt nazvaný bump mapping.

Při překladu této lekce jsem zvažoval zda mám některé termíny překládat do češtiny. Ale vzhledem k tomu, že jsou to většinou názvy, které se běžně v oboru počítačové grafiky objevují, rozhodl jsem se nechat je v původním znění. Aby však i ti, kteří se s nimi setkávají poprvé, věděli o čem je řeč, tak je zde v rychlosti vysvětlím:

OpenGL extension je funkce, která není v běžné specifikaci OpenGL dostupná, ale kvůli novým možnostem grafických akcelerátorů a novým postupům při programování byla do OpenGL dodatečně přidána. Tyto funkce ve svém názvu obsahují EXT nebo ARB. Firmy se samozřejmě snaží, aby jejich akcelerátor podporoval těchto rozšíření co nejvíce, protože mnohé z nich zrychlují práci, přidávají nové možnosti nebo zvyšují výkon.

Bumpmapa je textura, která obsahuje informace o reliéfu. Většinou bývá ve stupních šedi, kde tmavá místa udávají vyvýšeniny a světlá rýhy, nebo naopak - to záleží na programátorovi.

Emboss bumpmapping je postup vytváření reliéfovaných textur, u kterých se zdá, že jsou tvarované i do hloubky - hlavní téma této lekce.

Alpha kanál je poslední složka RGBA barvy, která obsahuje informace o průhlednosti. Pokud je alpha maximální (255 nebo 1.0f), tak není objekt vůbec průhledný. Pokud je alpha nulová je objekt neviditelný.

Blending je míchání alpha kanálu s barevnou texturou. Dociluje se jím průhlednosti.

Artefakt je nějaký vizuální prvek, který by se v renderované scéně neměl objevovat. Nicméně vzhledem k tomu, že postupy, které by je nezanedbávaly jsou většinou velmi pomalé, musí se používat jiné, které na úkor kvality zvýší rychlost renderování.

Další názvy typu **vertex**, **pipeline**, ... by měly být dobře známé z předchozích tutoriálů.

Doufám, že Vám překlad i téma budou srozumitelné a že Vám pomohou vytvářet kvalitní OpenGL aplikace. Pokud byste narazili na nějaký problém, není nic jednoduššího než poslat emailem dotaz. Rád Vám na všechny otázky odpovím, případně opravím nedostatky v textu.

Tato lekce byla napsána Jensem Schneiderem. Volně vychází z 6. lekce, i když vzniklo mnoho změn. Naučíte se zde:

- Jak ovládat multitexturovací možnosti grafického akcelerátoru.
- Jak vytvořit zdání emboss bumpmappingu (reliéf na texturách).
- Jak udělat pomocí blendingu profesionálně vypadající loga, která "plují" nad renderovanou scénou.
- Základy multi-pass (několika fázových) renderovacích technik.
- Jak využívat efektivně transformace matice.

Nejméně tři z výše uvedených bodů mohou být považovány za "pokročilé renderovací techniky". Měli byste mít již základní představu o tom, jak funguje renderovací pipeline OpenGL. Měli byste znát většinu příkazů užitých v tutoriálu a měli byste být obeznámeni s vektorovou matematikou. Sekce, které začínají slovy "začátek teorie(...)" a končí slovy "konec teorie(...)", se snaží vysvětlit problematiku uvedenou v závorkách. Tohle je zde jen pro jistotu. Pokud danou problematiku znáte, můžete tyto části jednoduše přeskočit. Pokud budete mít problémy s porozuměním kódu, zvažte návrat zpět k teoretickým částem textu. Poslední, ale neméně důležité: Tato lekce obsahuje více než 1 200 řádek kódu a velká část z nich je nejen nudná, ale i dobře známá těm, kteří četli předchozí tutoriály. Proto nebudu komentovat každý řádek, ale jen podstatu této lekce. Pokud narazíte na něco jako ><, znamená to, že zde byly vynechány nějaké nepodstatné řádky kódu.

Takže, jdeme na to:

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

#include "glext.h"// Hlavičkový soubor pro multitexturing
```

```
#include <string.h>// Hlavičkový soubor pro řetězce
#include <math.h>// Hlavičkový soubor pro matematiku
```

GLfloat MAX_EBOSS udává "sílu" bumpmappingu. Vyšší hodnoty hodně zvýrazní efekt, ale stejně tak sníží kvalitu obrazu tím, že zanechávají v rozích ploch takzvané "artefakty".

```
#define MAX_EBOSS (GLfloat)0.008f// Maximální posunutí efektem
```

Fajn, připravíme se na použití GL_ARB_multitexture. Je to celkem jednoduché:

Většina grafických akceleratorů má dnes více než jednu texturovací jednotku. Abychom mohli této výhody využít, musíme prověřit, zda akcelerator podporuje GL_ARB_multitexture, který umožňuje namapovat dvě nebo více textur na jeden útvar při jednom průchodu pipeline. Nezní to příliš významně, ale opak je pravdou! Skoro vždy když něco programujete, přidáním další textury na objekt, razantně zvýšíte jeho vizuální kvalitu. Dříve bylo nutno použít dvě prokládané textury při vícenásobném vykreslování geometrie, což může vést k velkému poklesu výkonu. Dále v tutoriálu bude multitexturing ještě podrobněji popsán.

Teď zpět ke kódu: __ARB_ENABLE je užito pro určení toho, zda chceme využít multitexturingu, když bude dostupný. Pokud chcete poznat vaši kartu podporovaná OpenGL rozšíření, pouze odkomentujte #define EXT_INFO. Dále chceme prověřit podporu extensions při běhu programu, abychom zajistili přenositelnost kódu. Proto potřebujeme místo pro pár řetězců. Dále chceme rozlišovat mezi možností používat extensions a samotným používáním. Nakonec potřebujeme vědět, kolik texturovacích jednotek máme k dispozici (použijeme ale pouze dvě). Alespoň jedna texturovací jednotka je vždy přítomna na akceleratoru podporujícím OpenGL, takže nastavíme maxTexelUnits na hodnotu 1.

```
#define __ARB_ENABLE true// Použito pro vyřazení multitexturingu

// #define EXT_INFO// Odkomentujte, pokud chcete při startu vidět podporovaná rozšíření OpenGL

#define MAX_EXTENSION_SPACE 10240// Místo pro řetězce s OpenGL rozšířeními
#define MAX_EXTENSION_LENGTH 256// Maximum znaků v jednom řetězci s rozšířením

bool multitextureSupported = false;// Indikátor podpory multitexturingu
bool useMultitexture = true;// Použít multitexturing?

GLint maxTexelUnits = 1;// Počet texturovacích jednotek - nejméně 1
```

Následující řádky slouží k tomu, aby spojily rozšíření s voláním funkcí v C++. Pouze využijeme PNF-kdo-to-kdy-přečetl jako předdefinovaného datového typu schopného popsat volání funkcí. Zpočátku není jisté, zda získáme přístup k těmto prototypům funkcí, tudíž je nastavíme na NULL. Příkazy glMultiTexCoordfARB odkazují na dobře známé příkazy glTexCoord(), udávající i-rozměrné souřadnice textury. Všimněte si, že proto mohou úplně nahradit příkazy glTexCoord. Dříve jsme používali pouze verzi pro typ GLfloat, my potřebujeme pouze prototypy k příkazům končícím na "f" - ostatní jsou potom taky dostupné (fv, i, ...). Poslední dva prototypy slouží k určení texturovací jednotky, která bude přijímat informace o textuře (glActiveTextureARB()) a k určení, která texturovací jednotka je asociována s příkazem ArrayPointer (glClientActiveTextureARB). Mimochodem: ARB je zkratkou "Architectural Review Board". Rozšíření s ARB v názvu nejsou vyžadovány pro implementaci kompatibilní s OpenGL, ale jsou široce využívány a podporovány.

```
PFNGLMULTITEXCOORD1FARBPROC glMultiTexCoord1fARB = NULL;
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLMULTITEXCOORD3FARBPROC glMultiTexCoord3fARB = NULL;
PFNGLMULTITEXCOORD4FARBPROC glMultiTexCoord4fARB = NULL;

PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = NULL;
```

Potřebujeme globální proměnné:

- filter - udává, jaký filtr se má použít. Použijeme nejspíše GL_LINEAR, takže filter inicializujeme číslem 1.
- texture - textury, potřebujeme 3 - na každý filtr jednu
- bump - bumpmapy
- invbump - převrácené bump mapy - jejich význam je popsán v jedné z teoretických částí této lekce
- glLogo a multiLogo - využijeme pro textury, které budou přidány do scény v poslední fázi rendrování
- proměnné s Light v názvu - jsou pole nesoucí informace o osvětlení scény

```
GLuint filter=1;// Jaký filtr použít
GLuint texture[3];// Místo pro tři textury

GLuint bump[3];// Naše bumpmapy
GLuint invbump[3];// Invertované bumpmapy
```



```

GLuint glLogo;// Místo pro OpenGL Logo
GLuint multiLogo;// Místo pro logo s multitexturingem

GLfloat LightAmbient[] = { 0.2f, 0.2f, 0.2f};// Barva ambientního světla je 20% bílá
GLfloat LightDiffuse[] = { 1.0f, 1.0f, 1.0f};// Difúzní světlo je bílé
GLfloat LightPosition[] = { 0.0f, 0.0f, 2.0f};// Pozice je někde uprostřed scény

GLfloat Gray[] = { 0.5f, 0.5f, 0.5f, 1.0f };// Barva okraje textury

bool emboss = false;// Jenom Emboss, žádná základní textura
bool bumps = true;// Používat bumpmapping?

GLfloat xrot;// X rotace
GLfloat yrot;// Y rotace

GLfloat xspeed;// Rychlost x rotace
GLfloat yspeed;// Rychlost y rotace

GLfloat z = -5.0f;// Hloubka v obrazovce

```

Další část kódu obsahuje souřadnice kostky sestavené z GL_QUADS. Každých pět čísel reprezentuje jednu sadu 2D texturovacích souřadnic a jednu sadu 3D vertexových souřadnic bodu. Data jsou uvedena v poli kvůli snazšímu vykreslování ve for smyčkách. Během jednoho renderovacího cyklu budeme tyto souřadnice potřebovat vícekrát.

```

GLfloat data[] =
{
    // Přední stěna
    0.0f, 0.0f, -1.0f, -1.0f, +1.0f,
    1.0f, 0.0f, +1.0f, -1.0f, +1.0f,
    1.0f, 1.0f, +1.0f, +1.0f, +1.0f,
    0.0f, 1.0f, -1.0f, +1.0f, +1.0f,
    // Zadní stěna
    1.0f, 0.0f, -1.0f, -1.0f, -1.0f,
    1.0f, 1.0f, -1.0f, +1.0f, -1.0f,
    0.0f, 1.0f, +1.0f, +1.0f, -1.0f,
    0.0f, 0.0f, +1.0f, -1.0f, -1.0f,
    // Horní stěna
    0.0f, 1.0f, -1.0f, +1.0f, -1.0f,
    0.0f, 0.0f, -1.0f, +1.0f, +1.0f,
    1.0f, 0.0f, +1.0f, +1.0f, +1.0f,
    1.0f, 1.0f, +1.0f, +1.0f, -1.0f,
    // Dolní stěna
    1.0f, 1.0f, -1.0f, -1.0f, -1.0f,
    0.0f, 1.0f, +1.0f, -1.0f, -1.0f,
    0.0f, 0.0f, +1.0f, -1.0f, +1.0f,
    1.0f, 0.0f, -1.0f, -1.0f, +1.0f,
    // Pravá stěna
    1.0f, 0.0f, +1.0f, -1.0f, -1.0f,
    1.0f, 1.0f, +1.0f, +1.0f, -1.0f,
    0.0f, 1.0f, +1.0f, +1.0f, +1.0f,
    0.0f, 0.0f, +1.0f, -1.0f, +1.0f,
    // Levá stěna
    0.0f, 0.0f, -1.0f, -1.0f, -1.0f,
    1.0f, 0.0f, -1.0f, -1.0f, +1.0f,
    1.0f, 1.0f, -1.0f, +1.0f, +1.0f,
    0.0f, 1.0f, -1.0f, +1.0f, -1.0f
};

```

Další část kódu rozhoduje o použití OpenGL extensions za běhu programu.

Předpokládejme, že máme dlouhý řetězec obsahující názvy všech podporovaných rozšíření oddělených znakem nového řádku -'\n'. Potřebujeme vyhledat znak nového řádku a tuto část začít porovnávat s hledaným řetězcem, dokud nenarazíme na další znak nového řádku, nebo dokud nalezený řetězec neodpovídá tomu hledanému. V prvním případě vrátíme true, v druhém případě vezmeme další sub-řetězec dokud nenarazíme na konec řetězce. Budeme si muset dát pozor na to, zda řetězec nezačíná znakem nového řádku.

Poznámka: Kontrola podpory rozšíření by se měla VŽDY provádět až za běhu programu.

```

bool isInString(char *string, const char *search)
{
    int pos = 0;

```

```

int maxpos = strlen(search)-1;
int len = strlen(string);
char *other;

for (int i=0; i<len; i++)
{
    if ((i==0) || ((i>1) && string[i-1]!='\n'))// Nové rozšíření začíná zde
    {
        other = &string[i];
        pos=0;// Začít nové hledání

        while (string[i]!='\n')// Hledání celého řetězce jména rozšíření
        {
            if (string[i]==search[pos])
                pos++;// Další znak

            if ((pos>maxpos) && string[i+1]!='\n')
                return true; // A máme to!

            i++;
        }
    }
}

return false;// Smůla, nic jsme nenašli!
}

```

Ted' musíme získat řetězec obsahující názvy extensions a převést ho tak, aby jednotlivé názvy byly odděleny znakem nového řádku. Pokud najdeme sub-řetězec "GL_ARB_multitexture", tak je tato funkce podporovaná. Ale my jí použijeme, jen když je `__ARB_ENABLE` nastaveno na true. Ještě potřebujeme zjistit podporu `GL_EXT_texture_env_combine`. Toto rozšíření zavádí nový způsob interakce s texturovacími jednotkami. My to potřebujeme, protože `GL_ARB_multitexture` pouze přenáší výstup z jedné texturovací jednotky do další s vyšším číslem. Než bychom používali další komplexní rovnice pro výpočet blendingu (které by ale mohly mít odlišný efekt), raději zajistíme podporu tohoto rozšíření. Pokud jsou všechna rozšíření podporována, zjistíme kolik texturovacích jednotek máme k dispozici a hodnotu uložíme do `maxTexelUnits`. Pak musíme spojit funkce s našimi jmény. To provedeme pomocí funkce `wglGetProcAddress()` s parametrem obsahujícím název funkce.

```

bool initMultitexture(void)
{
    char *extensions;

    extensions = strdup((char *) glGetString(GL_EXTENSIONS));// Získání řetězce s
    rozšířeními
    int len = strlen(extensions);// Délka řetězce

    for (int i = 0; i<len; i++)// Rozdělit znakem nového řádku místo mezery
        if (extensions[i] == ' ')
            extensions[i] = '\n';

#ifdef EXT_INFO
    MessageBox(hWnd,extensions,"supported GL extensions",MB_OK | MB_ICONINFORMATION);
#endif

    if (isInString(extensions,"GL_ARB_multitexture")// Je multitexturing podporován?
        && __ARB_ENABLE// Příznak pro povolení multitexturingu
        && isInString(extensions,"GL_EXT_texture_env_combine"))// Je podporováno texture-
        environment-combining?
    {
        glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, &maxTexelUnits);

        glMultiTexCoord1fARB = (PFNGLMULTITEXCOORD1FARBPROC)wglGetProcAddress
        ("glMultiTexCoord1fARB");
        glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)wglGetProcAddress
        ("glMultiTexCoord2fARB");
        glMultiTexCoord3fARB = (PFNGLMULTITEXCOORD3FARBPROC)wglGetProcAddress
        ("glMultiTexCoord3fARB");
        glMultiTexCoord4fARB = (PFNGLMULTITEXCOORD4FARBPROC)wglGetProcAddress
        ("glMultiTexCoord4fARB");

        glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)wglGetProcAddress
        ("glActiveTextureARB");
    }
}

```

```

    glClientActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC)wglGetProcAddress
        ("glClientActiveTextureARB");

#ifdef EXT_INFO
    MessageBox(hWnd,"The GL_ARB_multitexture extension will be used.,""feature
        supported!","",MB_OK | MB_ICONINFORMATION);
#endif

    return true;
}

useMultitexture = false;// Nemůžeme to používat, pokud to není podporováno!
return false;
}

```

initLights() pouze inicializuje osvětlení. Je volána funkcí initGL().

```

void initLights(void)
{
    glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);// Načtení informace o světlech do
    GL_LIGHT1
    glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
    glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);

    glEnable(GL_LIGHT1);
}

```

V této lekci vytvoříme hodně textur. Nyní k naší načítací funkci. Nejdříve loadujeme základní bitmapu a připravíme z ní tři filtrované textury (GL_NEAREST, GL_LINEAR, GL_LINEAR_MIPMAP_NEAREST). Použijeme pouze jednu datovou strukturu na uložení bitmap. Navíc zavedeme novou strukturu nazvanou alpha, která bude obsahovat informace o alpha kanálu (průhlednosti) textury. Proto uložíme RGBA obrázky jako dvě bitmapy: jednu 24 bitovou RGB a jednu osmi bitovou ve stupních šedi pro alpha kanál. Aby fungovalo načítání správně, musíme po každém načtení smazat Image, jinak nebudeme upozorněni na případné chyby při nahrávání textur.



Také je u specifikace typu textury vhodné uvést místo čísla 3 proměnnou GL_RGB8, a to kvůli lepší kompatibilitě s dalšími verzemi OpenGL. Tato změna je označena v kódu **takto**.

```

int LoadGLTextures()// Loading bitmapy a konverze na texturu
{
    bool status=true;// Indikuje chyby
    AUX_RGBImageRec *Image=NULL;// Ukládá bitmapu
    char *alpha=NULL;

    if (Image = auxDIBImageLoad("Data/Base.bmp"))// Nahraje bitmapu
    {
        glGenTextures(3, texture);// Generuje tři textury

        // Vytvoření nelineárně filtrované textury
        glBindTexture(GL_TEXTURE_2D, texture[0]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, Image->sizeX, Image->sizeY, 0, GL_RGB,
            GL_UNSIGNED_BYTE, Image->data);

        // Vytvoření lineárně filtrované textury
        glBindTexture(GL_TEXTURE_2D, texture[1]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, Image->sizeX, Image->sizeY, 0, GL_RGB,
            GL_UNSIGNED_BYTE, Image->data);
    }
}

```

```

// Vytvoření mipmapované textury
glBindTexture(GL_TEXTURE_2D, texture[2]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB8, Image->sizeX, Image->sizeY, GL_RGB,
GL_UNSIGNED_BYTE, Image->data);
}
else
    status = false;

if (Image) // Pokud obrázek existuje
{
    if (Image->data) // Pokud existují data obrázku
        delete Image->data; // Uvolní data obrázku

    delete Image; // Uvolní strukturu obrázku

    Image = NULL; // Nastaví ukazatel na NULL
}

```

Načteme bumpmapu. Z důvodů uvedených níže musí mít pouze 50% intenzitu, takže ji musíme nějakým způsobem ztmavit. Já jsem se rozhodl použít funkci `glPixelTransferf()`, která udává jakým způsobem budou bitmapy převedeny na textury. My tuto funkci použijeme na ztmavení jednotlivých RGB kanálů bitmapy na 50% původní intenzity. Pokud dosud nepoužíváte rodinu funkcí `glPixelTransferf()`, měli byste se na ně podívat - jsou celkem užitečné.



```

// Loading bumpmap
if (Image = auxDIBImageLoad("Data/Bump.bmp"))
{
    glPixelTransferf(GL_RED_SCALE, 0.5f); // Snížení intenzity RGB na 50% - poloviční
    intenzita
    glPixelTransferf(GL_GREEN_SCALE, 0.5f);
    glPixelTransferf(GL_BLUE_SCALE, 0.5f);
}

```

Další problém je, že nechceme, aby se bitmapa v textuře pořád opakovala, chceme ji namapovat pouze jednou na texturovací souřadnice od (0.0f, 0.0f) do (1.0f, 1.0f). Vše kolem nich by mělo být namapováno černou barvou. Toho dosáhneme zavoláním dvou funkcí `glTexParameteri()`, které není třeba popisovat.

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP); // Bez wrappingu
(zalamování)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, Gray); // Barva okraje
textury

glGenTextures(3, bump); // Vytvoří tři textury

// Vytvoření nelineárně filtrované textury
><

// Vytvoření lineárně filtrované textury
><

// Vytvoření mipmapované textury
><

```

Nyní musíme vytvořit ještě invertovanou bumpmapu, o které jsme již psali a jejíž význam bude vysvětlen dále. Odečtením barvy každého bodu bumpmapy od bílé barvy {255, 255, 255} získáme obrázek s invertovanými barvami. Předtím nesmíme nastavit intenzitu zpět na 100% (než jsem na to přišel strávil jsem nad tím asi 3 hodiny), invertovaná bitmapa musí být tedy také ztmavená na 50%.

```

for (int i = 0; i < 3 * Image->sizeX * Image->sizeY; i++) // Invertování bumpmapy

```

```

        Image->data[i] = 255 - Image->data[i];
    glGenTextures(3, invbump); // Vytvoří tři textury
    // Vytvoření nelineárně filtrované textury
    >-<
    // Vytvoření lineárně filtrované textury
    >-<
    // Vytvoření mipmapované textury
    >-<

    glPixelTransferf(GL_RED_SCALE,1.0f); // Vrácení intenzity RGB zpět na 100%
    glPixelTransferf(GL_GREEN_SCALE,1.0f);
    glPixelTransferf(GL_BLUE_SCALE,1.0f);
}
else
    status = false;
if (Image) // Pokud obrázek existuje
{
    if (Image->data) // Pokud existují data obrázku
        delete Image->data; // Uvolní data obrázku

    delete Image; // Uvolní strukturu obrázku

    Image = NULL; // Nastaví ukazatel na NULL
}

```

Načítání bitmap log je velmi jednoduché až na zkombinování RGB-A kanálů, nicméně kód by měl být dostatečně jasný. Všimněte si, že tato textura je vytvořena z dat alpha, nikoliv z dat Image. Bude zde použit pouze jeden filtr.



```

// Načte bitmapy log
if (Image = auxDIBImageLoad("Data/OpenGL_ALPHA.bmp"))
{
    alpha = new char[4*Image->sizeX*Image->sizeY]; // Alokuje paměť pro RGBA8-Texturu

    for (int a=0; a < Image->sizeX * Image->sizeY; a++)
        alpha[4*a+3] = Image->data[a*3]; // Vezme pouze červenou barvu jako alpha
        kanál

```



```

if (!(Image = auxDIBImageLoad("Data/OpenGL.bmp")))
    status = false;

for (a = 0; a < Image->sizeX * Image->sizeY; a++)
{
    alpha[4*a]=Image->data[a*3]; // R
    alpha[4*a+1]=Image->data[a*3+1]; // G
    alpha[4*a+2]=Image->data[a*3+2]; // B
}

glGenTextures(1, &glLogo); // Vytvoří jednu texturu

// Vytvoří lineárně filtrovanou RGBA8-Texturu
glBindTexture(GL_TEXTURE_2D, glLogo);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Image->sizeX, Image->sizeY, 0, GL_RGBA,
GL_UNSIGNED_BYTE, alpha);

```

```

    delete alpha;// Uvolní alokovanou paměť
}
else
    status = false;
if (Image)// Pokud obrázek existuje
{
    if (Image->data)// Pokud existují data obrázku
        delete Image->data;// Uvolní data obrázku

    delete Image;// Uvolní strukturu obrázku

    Image = NULL;// Nastaví ukazatel na NULL
}

```

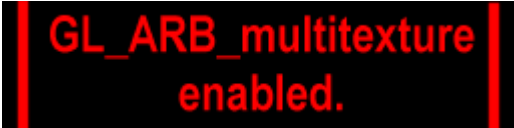


```

if (Image = auxDIBImageLoad("Data/multi_on_alpha.bmp"))
{
    alpha = new char[4*Image->sizeX*Image->sizeY];// Alokuje paměť pro RGBA8-Texturu

    for (int a = 0; a < Image->sizeX * Image->sizeY; a++)
        alpha[4*a+3]=Image->data[a*3];// Vezme pouze červenou barvu jako alpha kanál

```



```

if (!(Image=auxDIBImageLoad("Data/multi_on.bmp")))
    status = false;

```

```

for (a=0; a < Image->sizeX * Image->sizeY; a++)
{
    alpha[4*a] = Image->data[a*3];// R
    alpha[4*a+1] = Image->data[a*3+1];// G
    alpha[4*a+2] = Image->data[a*3+2];// B
}

```

```

glGenTextures(1, &multiLogo);// Vytvoří jednu texturu

```

```

// Vytvoří lineárně filtrovanou RGBA8-Texturu
glBindTexture(GL_TEXTURE_2D, multiLogo);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, Image->sizeX, Image->sizeY, 0, GL_RGBA,
GL_UNSIGNED_BYTE, alpha);

```

```

delete alpha;

```

```

}
else
    status = false;
if (Image)// Pokud obrázek existuje
{
    if (Image->data)// Pokud existují data obrázku
        delete Image->data;// Uvolní data obrázku

    delete Image;// Uvolní strukturu obrázku

    Image = NULL;// Nastaví ukazatel na NULL
}

return status;// Vrátí status
}

```

Následuje funkce doCube(), která kreslí krychli spolu s normálami. Všimněte si, že tato verze zatěžuje pouze texturovací jednotku #0, glTexCoord(s, t) pracuje stejně jako glMultiTexCoord(GL_TEXTURE0_ARB, s, t). Krychle může být taky vykreslena pomocí prokládaných polí, to ale teď nebudeme řešit. Nemůže však být uložena na display listu, ty používají pravděpodobně přesnost různou od GLfloat, což vede k nepěkným vedlejším efektům.

```

void doCube(void)
{
    int i;

    glBegin(GL_QUADS);

    // Přední stěna
    glNormal3f( 0.0f, 0.0f, +1.0f);
    for (i=0; i<4; i++)
    {
        glTexCoord2f(data[5*i],data[5*i+1]);
        glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Zadní stěna
    glNormal3f( 0.0f, 0.0f,-1.0f);
    for (i=4; i<8; i++)
    {
        glTexCoord2f(data[5*i],data[5*i+1]);
        glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Horní stěna
    glNormal3f( 0.0f, 1.0f, 0.0f);
    for (i=8; i<12; i++)
    {
        glTexCoord2f(data[5*i],data[5*i+1]);
        glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Spodní stěna
    glNormal3f( 0.0f,-1.0f, 0.0f);
    for (i=12; i<16; i++)
    {
        glTexCoord2f(data[5*i],data[5*i+1]);
        glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Pravá stěna
    glNormal3f( 1.0f, 0.0f, 0.0f);
    for (i=16; i<20; i++)
    {
        glTexCoord2f(data[5*i],data[5*i+1]);
        glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    // Levá stěna
    glNormal3f(-1.0f, 0.0f, 0.0f);
    for (i=20; i<24; i++)
    {
        glTexCoord2f(data[5*i],data[5*i+1]);
        glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
    }

    glEnd();
}

```

Přichází čas na inicializaci OpenGL. Vše je jako v lekcí 06, kromě toho, že zavoláme funkci `initLights()`, místo toho, abychom světla nastavovali zde. A ještě samozřejmě voláme nastavení případného multitexturingu.

```

int InitGL(GLvoid)// Všechno nastavení OpenGL
{
    multitextureSupported = initMultitexture();

    if (!LoadGLTextures())// Vytvoření textur
        return false;

    glEnable(GL_TEXTURE_2D);// Zapne texturové mapování
    glShadeModel(GL_SMOOTH);// Zapne smooth shading
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Černé pozadí
    glClearDepth(1.0f);// Nastavení hloubkového bufferu

```

```

glEnable(GL_DEPTH_TEST); // Povolení testování hloubky
glDepthFunc(GL_LEQUAL); // Typ testování hloubky
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Kvalitní výpočty perspektivy

initLights(); // Inicializace světel

return true; // Vše v pořádku
}

```

Začátek teorie (Emboss Bump Mapping)

Zde je asi 95% práce. Vše u čeho bylo napsáno, že bude vysvětleno později, je v následující teoretické sekci. Jedná se o přepsání prezentace v PowerPointu do HTML.

Emboss Bump Mapping

Michael I. Gold - NVidia Corporation

Bump Mapping

Skutečný bump mapping používá per-pixel osvětlení.

- Výpočet osvětlení na každém pixelu založené na různých normálových vektorech.
- Výpočetně velmi náročné.
- Pro více informací se podívejte na: Blinn, J. : Simulation of Wrinkled Surfaces, Computer Graphics. 12,3 (August 1978) 286-292.
- Pro informace na webu zajděte na: <http://www.objectecture.com/> a podívejte se na Cass Everitt's Orthogonal Illumination Thesis. (pozn.: Jens)

Emboss Bump Mapping

Emboss Bump Mapping je pouze náhražka.

- Pouze difusní osvětlení, žádné odražené.
- Výskyt artefaktů (může vést k rozmazanému pohybu pozn.: Jens)
- Dostupné na dnešním hardwaru
- Vypadá celkem slušně

Výpočet difúzního osvětlení

$C = (L * N) \times D_I \times D_m$

- L je vektor světla
- N je normálový vektor
- D_I je barva difusního světla
- D_m je difusní barva materiálu
- Bump Mapping mění pro každý pixel N
- Emboss Bump Mapping se blíží $L * N$

Přibližný stupeň rozptylu $L * N$

Textura reprezentuje výškovou mapu

- $[0,1]$ určuje interval prohybu (výškového rozdílu)
- První odvození reprezentuje sklon (úhel) m - m je pouze jednorozměrné - reprezentuje sklon na souřadnicích (s,t) dané textury (pozn.: Jens)
- m zvyšuje nebo snižuje základní stupeň rozptylu F_d
- $(F_d + m)$ se blíží $(L * N)$ na pixel

Přibližné odvození

Zohlednění přibližných údajů

- Vyvýšení H_0 v bodě o souřadnicích (s,t)
- Vyvýšení H_1 v bodě mírně posunutém ke zdroji světla $(s + ds, t + dt)$
- Odečtení původní výšky H_0 od posunuté H_1
- Rozdíl je okamžitým sklonem $m = H_1 - H_0$

Spočítání reliéfu

1) Původní reliéf (H_0).



2) Původní reliéf (H_0) proložený druhým (H_1), který je mírně posunutý směrem ke světlu.



3) Odečtení původního od posunutého reliéfu ($H_0 - H_1$) - vede ke vzniku světlých (B) a tmavých (D) ploch.



Výpočet osvětlení

Určíme hodnotu barvy (C_f) dané plochy

- $C_f = (L \cdot N) \times D_I \times D_m$
- $(L \cdot N) \sim (F_d + (H_0 - H_1))$
- $D_m \times D_I$ je vlastně již určen texturou C_t . Jinak můžete pracovat s D_I a D_m samostatně, pokud jste dostatečně zdatní. (To se provádí pomocí OpenGL-Lighting! pozn.: Jens)
- $C_f = (F_d + (H_0 - H_1)) \times C_t$

Je to vše? Takhle jednoduché?!

Ještě nejsme úplně hotoví. Stále musíme:

- Vytvořit texturu (pomocí grafického editoru pozn.: Jens)
- Spočítat posunutí textury (ds, dt)
- Spočítat stupeň rozptylu F_d (pomocí OpenGL-Lighting! pozn.: Jens)
- Obojí je odvozeno z normály N a vektoru světla L (v našem případě se spočítá přesně pouze (ds, dt) ! pozn.: Jens)
- Teď si dáme trošku matematiky

Tvorba textury

Uchovávejte textury!

- Současný multitexturovací hardware podporuje pouze dvě textury!
- Bumpmapa v alpha kanálu (my to tímto způsobem neděláme, ale můžete si to zkusit jako takové cvičení, pokud máte TNT chipset pozn.: Jens)
- Maximální prohyb = 1.0
- Základní výška = 0.5
- Maximální pokles = 0.0
- Barva povrchu v RGB kanálech

- Nastavit interní formát na GL_RGBA8 !!

Výpočet offsetu textury

Pootočení vektoru světla

- Potřeba je normální souřadnicový systém
- Odvození souřadnicového systému z normálového a "horního" vektoru (my předáme směr texturovacích souřadnic do našeho generátoru posunutí explicitně pozn.: Jens)
- Normála je osa z
- Meziprodukt je osa x
- Zahození "horního" vektoru, odvození osy y z os x a z
- Vytvoření matice M_n 3x3 ze spočítaných os
- Transformace vektoru světla do normálního prostoru (M_n se taky nazývá ortonormální základ pozn.: Jens)

Výpočet offsetu textury (pokračování)

Použijte pro posunutí vektor světla normálního prostoru

- $L' = M_n \times L$
- Použít L'_x , L'_y pro (ds, dt)
- Použít L'_z pro stupeň rozptylu! (Raději ne! Pokud nevládníte TNT, použijte místo toho OpenGL-Lighting, jinak byste museli renderovat jeden cyklus navíc! pozn.: Jens)
- Pokud je vektor světla blízky normále, L'_x , L'_y jsou nízké
- Pokud se vektor světla blíží tangentské rovině, L'_x , L'_y jsou vysoké
- Co když je L'_z menší než nula?
- Světlo je na opačné straně než normála
- Pak se bude rovnat nule.

Implementace na TNT

Spočítejte vektory, texturovací souřadnice na hostiteli

- Předejte stupeň rozptylu v alpha kanálu
- Mohli byste využít barvu vertexu pro barvu rozptýleného světla
- H_0 a barvu z texturovací jednotky 0
- H_1 z texturovací jednotky 1 (stejná textura jiné souřadnice)
- ARB_multitexture extension
- Zkombinuje extension (precizněji: NVIDIA_multitexture_combiners extension, podporované všemi akcelerátory rodiny TNT pozn.: Jens)

Implementace na TNT (pokračování)

Nastavení alpha kanálu na combineru

- $(1-T_0a) + T_1a - 0.5$ (T_0a zastupuje "texturovací jednotku 0, alpha kanál" pozn.: Jens)
- (T_1a-T_0a) se namapuje na (-1,1), ale hardware ji připevní na (0,1)
- Přednastavení 0.5 vyvažuje ztrátu oproti uchycení (zvažte užití 0.5, mohli byste dosáhnout větší rozmanitosti bumpmap, pozn.: Jens)
- Můžete přizpůsobit barvu rozptýleného světla T_0c
- RGB nastavení combineru 0:
- $(T_0c * C_0a + T_0c * Fda - 0.5)*2$
- 0.5 vyvažuje ztrátu oproti uchycení
- Násobení dvěma prosvětlí obraz

Konec teorie (Emboss Bump Mapping)

My to ale uděláme trochu jinak než podle TNT implementace, abychom umožnili našemu programu běžet na VŠECH akcelerátorech. Zde se můžeme přiučit dvě nebo tři věci. Jedna z nich je, že bumpmapping je více fázový algoritmus na většině karet (ne na TNT, kde se to dá nahradit jednou dvou-texturovací fází). Už byste si měli být schopni představit, jak hezký multitexturing ve skutečnosti je. Nyní implementujeme 3-fázový netexturovací algoritmus, který pak může být (a bude) vylepšen na 2 fázový texturovací algoritmus.

Teď byste si měli uvědomit, že musíme udělat nějaké násobení matice maticí (a násobení vektoru maticí). Ale to není nic čeho bychom se měli obávat: OpenGL zvládne násobení matice maticí za nás a násobení vektoru maticí je celkem jednoduché: funkce `VMatMult(M,v)` vynásobí matici `M` s vektorem `v` a výsledek uloží zpět ve `v`: $v = M * v$. Všechny matice a vektory předané funkci musejí mít stejný tvar: matice 4x4 a 4-rozměrné vektory. To je pro zajištění kompatibility s OpenGL.

```
void VMatMult(GLfloat *M, GLfloat *v)
{
    GLfloat res[3];

    res[0] = M[0]*v[0]+M[1]*v[1]+M[ 2]*v[2]+M[ 3]*v[3];
    res[1] = M[4]*v[0]+M[5]*v[1]+M[ 6]*v[2]+M[ 7]*v[3];
    res[2] = M[8]*v[0]+M[9]*v[1]+M[10]*v[2]+M[11]*v[3];

    v[0]=res[0];
    v[1]=res[1];
    v[2]=res[2];

    v[3]=M[15]; // Homogenní souřadnice
}
```

Začátek teorie (algoritmy pro Emboss Bump Mapping)

Zde se zmíníme o dvou odlišných algoritmech. První popisuje program, který se jmenuje `GL_BUMP` a napsal ho Diego Tártara v roce 1999. I přes pár nevýhod velmi pěkně implementuje bumpmapping. Teď se na tento algoritmus podíváme:

1. Všechny vektory musí být BUĎ v prostoru objektu NEBO v prostoru scény
2. Spočítání vektoru v z aktuální pozice vertexu vzhledem ke světlu
3. Normalizace v
4. Promítnutí v do tangenoidního prostoru. (To je plocha, která se dotýká daného vertexu. Pokud pracujete s rovnými plochami, tak je to zpravidla plocha samotná.)
5. Posuneme souřadnice (s,t) o složky x,y vektoru v

To nevypadá špatně! V podstatě je to algoritmus popsáný Michaelem I. Goldem výše. Má však zásadní nevýhodu: Támara používá projekci pouze pro rovinu xy . To pro naše potřeby nestačí, protože zjednodušuje promítací krok pouze na složky x a y a se složkou z vektoru v vůbec nepočítá.

Ale tato implementace vytvoří rozptýlené světlo stejným způsobem, jako ho budeme dělat my: s použitím `v` v OpenGL zabudované podpory osvětlení. Takže nemůžeme použít metodu kombinerů, jakou navrhuje Gold (Chceme, aby naše programy běžely i na jiných než TNT kartách!), nemůžeme uložit stupeň rozptylu do `alpha` kanálu. Tak již máme problém s 3 fázovým netexturováním a 2 fázovým texturováním, proč na poslední průchod nepoužít `OpenGL-Lighting`, aby za nás dodělal ambientní světlo a barvy? Je to možné (a výsledek vypadá celkem dobře), ale jen proto, že nyní nepoužíváme složitou geometrii. Tohle byste si měli zapamatovat. Pokud budete chtít renderovat několik tisíc bumpmapovaných trojúhelníků, zkuste objevit něco jiného.

Navíc, používá multitexturing (jak můžeme vidět) ne tak jednoduše jako my s ohledem na tento speciální případ.

Ale teď k naší implementaci. Vypadá podobně jako algoritmus předtím, kromě projekční fáze, kde použijeme vlastní postup:

- Použijeme **SOUŘADNICE OBJEKTU**, to znamená, že nepoužijeme matici modelu při výpočtech. Tohle má za příčinu nemilý vedlejší efekt: když chceme otáčet krychli, souřadnice v objektu se nezmění, ale souřadnice vertexu v souřadnicích scény (vzhledem k očím) se změní. Ale pozice našeho světla by se neměla pohybovat s krychlí, měla by být statická, což znamená, že souřadnice by se neměly měnit. Abychom to vykompenzovali, použijeme malý trik, běžně užívaný s počítačové grafice: místo transformace každého vertexu do prostoru světa kvůli bumpmapám, převedeme souřadnice světla do prostoru objektu s použitím inverzní matice modelu. Tohle je velmi snadné, vzhledem k tomu, že přesně víme, jak jsme vytvořili matici modelu, není problém tento postup obrátit. K tomu se ještě dostaneme.
- Spočítáme daný vertex c na povrchu.

- Pak spočítáme normálu n s délkou 1 (většinou známe n pro každou stěnu krychle). To je důležité, můžeme tak ušetřit čas při zjišťování normalizovaných vektorů. Spočítáme vektor světla v z vektoru c směřujícímu k pozici světla l .
- Pokud je třeba ještě něco udělat, sestavíme matici M_n reprezentující ortonormální projekci.
- Spočítáme posunutí souřadnic textury vynásobením daných souřadnic textury (s a t) a v a MAX_EMBOSS : $ds = s*v*MAX_EMBOSS$, $dt = t*v*MAX_EMBOSS$. Všimněte si, že s, t a v jsou vektory, ale MAX_EMBOSS není.
- V druhé fázi přidáme posunutí k souřadnicím textury.

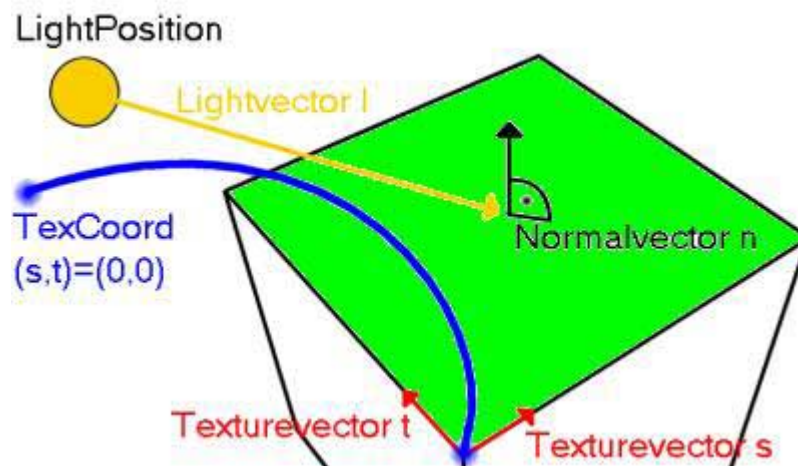
Proč je to dobré?

- Rychlost (jen pár odmocnin a násobení vertexů)
- Vypadá dobře!
- Funguje se všemi povrchy, nejen s rovinami.
- Běží na všech akcelerátorech.
- Je glBegin/glEnd přátelská: nepotřebuje "zakázané" GL příkazy.

Nevýhody:

- Není úplně fyzikálně správné.
- Zanechává menší artefakty.

Tento náčrtek ukazuje, kde se nacházejí jednotlivé vektory. Můžete jednoduše získat t a s odečtením jasnosti jednotlivých vektorů, ale ujistěte se, že jsou správně natočené a normalizované. Modrý bod označuje vertex, kde je namapován `texCoord2f(0.0f, 0.0f)`.



Konec teorie (algoritmy pro Emboss Bump Mapping)

Teď se podívejme na generátor posunutí textury. Tato funkce se jmenuje `SetUpBumps()`.

```
// Funkce nastaví posunutí textury
// n : normála k ploše, musí mít délku 1
// c : nějaký bod na povrchu
// l : pozice světla
// s : směr texturovacích souřadnic s (musí být normalizován!)
// t : směr texturovacích souřadnic t (musí být normalizován!)

void SetUpBumps(GLfloat *n, GLfloat *c, GLfloat *l, GLfloat *s, GLfloat *t)
{
    GLfloat v[3]; // Vertex z aktuální pozice ke světlu
    GLfloat lenQ; // Použito při normalizaci

    // Spočítání v z aktuálního vertexu c ke světlu a jeho normalizace
    v[0] = l[0] - c[0];
    v[1] = l[1] - c[1];
}
```

```

v[2] = l[2] - c[2];

lenQ = (GLfloat) sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);

v[0] /= lenQ;
v[1] /= lenQ;
v[2] /= lenQ;

// Zohlednění v tak, abychom dostali texturovací souřadnice
c[0] = (s[0]*v[0] + s[1]*v[1] + s[2]*v[2]) * MAX_EMOSS;
c[1] = (t[0]*v[0] + t[1]*v[1] + t[2]*v[2]) * MAX_EMOSS;
}

```

Nepřipadá vám to tak komplikované jako předtím? Teorie je ale důležitá, abyste pochopili jak efekt funguje a jak ho ovládat. Během psaní tutoriálu jsem se to sám naučil :-]

Vždycky jsem chtěl zobrazit logo při běhu ukázkového programu. My teď taky dvě zobrazíme. Zavoláme funkci doLogo (). Ta vyresetuje GL_MODELVIEW matici, která musí být při posledním průchodu zavolána.

Tato funkce zobrazí dvě loga: OpenGL logo a logo multitexturingu, pokud je povolen. Loga jsou zčásti průhledná. Protože mají alpha kanál, smícháme je pomocí GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA podle OpenGL dokumentace. Obě dvě jsou ploché, nemáme pro ně souřadnice z. Čísla použita pro hrany jsou zjištěny "empiricky" (pokus-chyba), tak aby loga padla pěkně do rožků. Musíme zapnout blending a vypnout světla, abychom se vyhnuli chybným efektům. Abychom zajistili, že loga budou vždy vepředu, vyresetujeme GL_MODELVIEW matici a nastavíme funkci na testování hloubky na GL_ALWAYS.

```

void doLogo(void) // MUSÍ SE ZAVOLAT AŽ NAKONEC!!! Zobrazí dvě loga
{
    glDepthFunc(GL_ALWAYS);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glEnable(GL_BLEND);
    glDisable(GL_LIGHTING);

    glLoadIdentity();

    glBindTexture(GL_TEXTURE_2D, glLogo);

    glBegin(GL_QUADS);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(0.23f, -0.4f, -1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(0.53f, -0.4f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(0.53f, -0.25f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(0.23f, -0.25f, -1.0f);
    glEnd();

    if (useMultitexture)
    {
        glBindTexture(GL_TEXTURE_2D, multiLogo);

        glBegin(GL_QUADS);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.53f, -0.4f, -1.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f(-0.33f, -0.4f, -1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.33f, -0.3f, -1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.53f, -0.3f, -1.0f);
        glEnd();
    }

    glDepthFunc(GL_LEQUAL);
}

```

Teď přichází funkce na bumpmapping bez texturingu. Je to tří-průchodová implementace. Jako první GL_MODELVIEW matice se převrátí pomocí aplikace všech provedených kroků v opačném pořadí a obráceně na matici dané identity. Výsledkem je matice, která při aplikaci na objekt "vrací" GL_MODELVIEW. My jí jednoduše získáme funkcí glGetFloatv (). Pamatujte, že matice musí být pole s 16 prvky a že je tato matice "přesunuta"!

Mimochodem: Když přesně nevíte, jak se s maticí manipuluje, zvažte použití globálních souřadnic, protože převrácení matice je složité a náročné na čas. Ale pokud používáte mnoho vertexů, převrácení matice může být daleko rychlejší.

```

bool doMesh1TexelUnits(void)
{
    GLfloat c[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Aktuální vertex
    GLfloat n[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Normalizovaná normála daného povrchu
    GLfloat s[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Směr texturovacích souřadnic s,

```

```

normalizováno
GLfloat t[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Směr texturovacích souřadnic t,
normalizováno

GLfloat l[4]; // Pozice světla, která bude transformována do prostoru objektu
GLfloat Minv[16]; // Převrácená modelview matice

int i;

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
buffer

// Sestavení převrácené modelview matice; nahradí funkce Push a Pop jednou funkcí
glLoadIdentity()
// Jednoduché sestavení tím, že všechny transformace provedeme opačně a v opačném
pořadí

glLoadIdentity();

glRotatef(-yrot, 0.0f, 1.0f, 0.0f);
glRotatef(-xrot, 1.0f, 0.0f, 0.0f);
glTranslatef(0.0f, 0.0f, -z);

glGetFloatv(GL_MODELVIEW_MATRIX, Minv);

glLoadIdentity();

glTranslatef(0.0f, 0.0f, z);
glRotatef(xrot, 1.0f, 0.0f, 0.0f);
glRotatef(yrot, 0.0f, 1.0f, 0.0f);

// Transformace pozice světla do souřadnic objektu:
l[0] = LightPosition[0];
l[1] = LightPosition[1];
l[2] = LightPosition[2];
l[3] = 1.0f; // Homogení souřadnice

VMatMult(Minv, l);

```

První fáze:

- Použití bump textury
- Vypnutí blendingu
- Vypnutí světla
- Použití texturovacích souřadnic bez posunutí
- Vytvoření geometrie

Tohle vyrenderuje krychli pouze z bumpmap.

```

glBindTexture(GL_TEXTURE_2D, bump[filter]);
glDisable(GL_BLEND);
glDisable(GL_LIGHTING);
doCube();

```

Druhá fáze:

- Použití převrácené bumpmapy
- Povolení blendingu GL_ONE, GL_ONE
- Ponechá vypnutá světla
- Použití posunutých texturovacích souřadnic (Před každou stěnou krychle musíme zavolat funkci SetUpBumps())
- Vytvoření geometrie

Tohle vyrenderuje krychli se správným emboss bumpmappingem, ale bez barev.

Mohli bychom ušetřit čas rotací vektoru světla opačným směrem. To však nefunguje úplně správně, tak to uděláme jinou cestou: otočíme každou normálu a prostřední bod stejně jako naši geometrii.

```

glBindTexture(GL_TEXTURE_2D, invbump[filter]);

```

```

glBlendFunc(GL_ONE, GL_ONE);
glDepthFunc(GL_EQUAL);
glEnable(GL_BLEND);

glBegin(GL_QUADS);
    // Přední stěna
    n[0] = 0.0f;
    n[1] = 0.0f;
    n[2] = 1.0f;

    s[0] = 1.0f;
    s[1] = 0.0f;
    s[2] = 0.0f;

    t[0] = 0.0f;
    t[1] = 1.0f;
    t[2] = 0.0f;

    for (i=0; i<4; i++)
    {
        c[0] = data[5*i+2];
        c[1] = data[5*i+3];
        c[2] = data[5*i+4];

        SetUpBumps(n, c, l, s, t);
        glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
        glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
    }

    // Zadní stěna
    n[0] = 0.0f;
    n[1] = 0.0f;
    n[2] = -1.0f;

    s[0] = -1.0f;
    s[1] = 0.0f;
    s[2] = 0.0f;

    t[0] = 0.0f;
    t[1] = 1.0f;
    t[2] = 0.0f;

    for (i=4; i<8; i++)
    {
        c[0] = data[5*i+2];
        c[1] = data[5*i+3];
        c[2] = data[5*i+4];

        SetUpBumps(n, c, l, s, t);
        glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
        glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
    }

    // Horní stěna
    n[0] = 0.0f;
    n[1] = 1.0f;
    n[2] = 0.0f;

    s[0] = 1.0f;
    s[1] = 0.0f;
    s[2] = 0.0f;

    t[0] = 0.0f;
    t[1] = 0.0f;
    t[2] = -1.0f;

    for (i=8; i<12; i++)
    {
        c[0] = data[5*i+2];
        c[1] = data[5*i+3];
        c[2] = data[5*i+4];
    }

```

```

        SetUpBumps(n,c,l,s,t);
        glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
        glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
    }

    // Spodní stěna
    n[0] = 0.0f;
    n[1] = -1.0f;
    n[2] = 0.0f;

    s[0] = -1.0f;
    s[1] = 0.0f;
    s[2] = 0.0f;

    t[0] = 0.0f;
    t[1] = 0.0f;
    t[2] = -1.0f;

    for (i=12; i<16; i++)
    {
        c[0] = data[5*i+2];
        c[1] = data[5*i+3];
        c[2] = data[5*i+4];

        SetUpBumps(n,c,l,s,t);
        glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
        glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
    }

    // Pravá stěna
    n[0] = 1.0f;
    n[1] = 0.0f;
    n[2] = 0.0f;

    s[0] = 0.0f;
    s[1] = 0.0f;
    s[2] = -1.0f;

    t[0] = 0.0f;
    t[1] = 1.0f;
    t[2] = 0.0f;

    for (i=16; i<20; i++)
    {
        c[0] = data[5*i+2];
        c[1] = data[5*i+3];
        c[2] = data[5*i+4];

        SetUpBumps(n,c,l,s,t);
        glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
        glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
    }

    // Levá stěna
    n[0] = -1.0f;
    n[1] = 0.0f;
    n[2] = 0.0f;

    s[0] = 0.0f;
    s[1] = 0.0f;
    s[2] = 1.0f;

    t[0] = 0.0f;
    t[1] = 1.0f;
    t[2] = 0.0f;

    for (i=20; i<24; i++)
    {
        c[0] = data[5*i+2];
        c[1] = data[5*i+3];
        c[2] = data[5*i+4];
    }

```



```

        SetUpBumps(n, c, l, s, t);
        glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
        glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
    }
    glEnd();

```

Třetí fáze:

- Použití základní barevné textury
- Povolení blendingu `GL_DST_COLOR`, `GL_SRC_COLOR`
- Tuto blending rovnici násobit dvěma: $(Cdst * Csrc) + (Csrc * Cdst) = 2(Csrc * Cdst)$!
- Povolení světel, aby vytvořily ambientní a rozptýlené světlo
- Vrácení `GL_TEXTURE` matice zpět na "normální" texturovací souřadnice
- Vytvořit geometrii

Tohle dokončí renderování krychle s osvětlením. Nejdříve musíme nastavit texture environment na `GL_MODULATE`. Můžeme zapínat a vypínat multitexturing. Tuto fázi provedeme, jen pokud uživatel nechce vidět pouze emboss.

```

    if (!emboss)
    {
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
        glBindTexture(GL_TEXTURE_2D, texture[filter]);
        glBlendFunc(GL_DST_COLOR, GL_SRC_COLOR);
        glEnable(GL_LIGHTING);

        doCube();
    }

```

Poslední fáze:

- Pootočení krychle pro příští kreslení
- Nakreslení log

```

    xrot += xspeed;
    yrot += yspeed;

    if (xrot > 360.0f)
        xrot -= 360.0f;

    if (xrot < 0.0f)
        xrot += 360.0f;

    if (yrot > 360.0f)
        yrot -= 360.0f;

    if (yrot < 0.0f)
        yrot += 360.0f;

    doLogo(); // Nakonec loga

    return true;
}

```

Další funkce udělá tohle všechno ve dvou fázích s podporou multitexturingu. Použijeme dvě texturovací jednotky. Více by bylo extrémně obtížné vzhledem k blendingovým rovnicím. Lépe použít TNT. Všimněte si, že se funkce liší od `doMesh1TexelUnits()` jen tím, že posíláme dvě sady texturovacích souřadnic na každý vertex!

```

bool doMesh2TexelUnits(void)
{
    GLfloat c[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Aktuální vertex
    GLfloat n[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Normalizovaná normála povrchu
    GLfloat s[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Směr texturovacích souřadnic s,
    normalizováno
    GLfloat t[4] = {0.0f, 0.0f, 0.0f, 1.0f}; // Směr texturovacích souřadnic t,
    normalizováno

    GLfloat l[4]; // Pozice světla k převedení na souřadnice objektu
    GLfloat Minv[16]; // Převrácená modelview matice

```

```

int i;

glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
buffer

// Sestavení převrácené modelview matice, tohle nahradí funkce Push a Pop jednou
funkcí glLoadIdentity()
// Jednoduché sestavení tím, že všechny transformace provedeme opačně a v opačném
pořadí

glLoadIdentity();

glRotatef(-yrot, 0.0f, 1.0f, 0.0f);
glRotatef(-xrot, 1.0f, 0.0f, 0.0f);
glTranslatef(0.0f, 0.0f, -z);

glGetFloatv(GL_MODELVIEW_MATRIX, Minv);

glLoadIdentity();

glTranslatef(0.0f, 0.0f, z);

glRotatef(xrot, 1.0f, 0.0f, 0.0f);
glRotatef(yrot, 0.0f, 1.0f, 0.0f);

// Transformace pozice světla na souřadnice objektu:
l[0] = LightPosition[0];
l[1] = LightPosition[1];
l[2] = LightPosition[2];
l[3] = 1.0f; // Homogení souřadnice

VMatMult(Minv, l);

```

První fáze:

- Bez blendingu
- Bez světel

Nastavení texture combineru 0 na

- Použití bumpmapy
- Použití neposunutých texturovacích souřadnic
- Nastavení operace s texturou na GL_REPLACE, která pouze vykreslí texturu

Nastavení texture combineru 1 na

- Posunutě texturovací souřadnice
- Nastavení operace s texturou na GL_ADD, což je multitexturovacím ekvivalentem k ONE, ONE blendingu

Tohle vyrenderuje krychli skládající se z šedých map.

```

// TEXTUROVACÍ JEDNOTKA #0:
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, bump[filter]);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);

// TEXTUROVACÍ JEDNOTKA #1:
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, invbump[filter]);
glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_ADD);

// Obecné přepínače
glDisable(GL_BLEND);
glDisable(GL_LIGHTING);

```

Ted' pouze vyrenderujeme stěny jednu po druhé jako v doMesh1TexelUnits(). Pouze jedna novinka: používá glMultiTexCoordfARB() místo glTexCoord2f(). Všimněte si, že v prvním parametru je uvedeno, které texturovací jednotce přísluší souřadnice. Parametr musí být GL_TEXTUREi_ARB, kde i je v intervalu od 0 do 31.

```
glBegin(GL_QUADS);
// Přední stěna
n[0] = 0.0f;
n[1] = 0.0f;
n[2] = 1.0f;

s[0] = 1.0f;
s[1] = 0.0f;
s[2] = 0.0f;

t[0] = 0.0f;
t[1] = 1.0f;
t[2] = 0.0f;

for (i=0; i<4; i++)
{
    c[0] = data[5*i+2];
    c[1] = data[5*i+3];
    c[2] = data[5*i+4];

    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}

// Zadní stěna
n[0] = 0.0f;
n[1] = 0.0f;
n[2] = -1.0f;

s[0] = -1.0f;
s[1] = 0.0f;
s[2] = 0.0f;

t[0] = 0.0f;
t[1] = 1.0f;
t[2] = 0.0f;

for (i=4; i<8; i++)
{
    c[0] = data[5*i+2];
    c[1] = data[5*i+3];
    c[2] = data[5*i+4];

    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}

// Horní stěna
n[0] = 0.0f;
n[1] = 1.0f;
n[2] = 0.0f;

s[0] = 1.0f;
s[1] = 0.0f;
s[2] = 0.0f;

t[0] = 0.0f;
t[1] = 0.0f;
t[2] = -1.0f;

for (i=8; i<12; i++)
{
    c[0] = data[5*i+2];
    c[1] = data[5*i+3];
```

```

    c[2] = data[5*i+4];

    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}

// Dolní stěna
n[0] = 0.0f;
n[1] = -1.0f;
n[2] = 0.0f;

s[0] = -1.0f;
s[1] = 0.0f;
s[2] = 0.0f;

t[0] = 0.0f;
t[1] = 0.0f;
t[2] = -1.0f;

for (i=12; i<16; i++)
{
    >c[0] = data[5*i+2];
    c[1] = data[5*i+3];
    c[2] = data[5*i+4];

    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}

// Pravá stěna
n[0] = 1.0f;
n[1] = 0.0f;
n[2] = 0.0f;

s[0] = 0.0f;
s[1] = 0.0f;
s[2] = -1.0f;

t[0] = 0.0f;
t[1] = 1.0f;
t[2] = 0.0f;

for (i=16; i<20; i++)
{
    c[0] = data[5*i+2];
    c[1] = data[5*i+3];
    c[2] = data[5*i+4];

    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}

// Levá stěna
n[0] = -1.0f;
n[1] = 0.0f;
n[2] = 0.0f;

s[0] = 0.0f;
s[1] = 0.0f;
s[2] = 1.0f;

t[0] = 0.0f;
t[1] = 1.0f;
t[2] = 0.0f;

for (i=20; i<24; i++)

```

```

    {
        c[0] = data[5*i+2];
        c[1] = data[5*i+3];
        c[2] = data[5*i+4];

        SetUpBumps(n,c,l,s,t);
        glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
        glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
        glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
    }
    glEnd();

```

Druhá fáze:

- Použití základní textury
- Povolení osvětlení
- Neposunuté texturovací souřadnice - vyresetovat GL_TEXTURE matice
- Nastavení texture environment na GL_MODULATE

Tohle vyrenderuje celou bumpmapovanou krychli.

```

glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);

if (!emboss)
{
    glTexEnvf (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    glBindTexture(GL_TEXTURE_2D,texture[filter]);
    glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);

    glEnable(GL_BLEND);
    glEnable(GL_LIGHTING);

    doCube();
}

```

Poslední fáze:

- Pootočení krychle
- Nakreslení log

```

xrot += xspeed;
yrot += yspeed;

if (xrot>360.0f)
    xrot -= 360.0f;

if (xrot<0.0f)
    xrot += 360.0f;

if (yrot>360.0f)
    yrot -= 360.0f;

if (yrot<0.0f)
    yrot += 360.0f;

doLogo();// Nakonec loga

return true;
}

```

Konečně funkce na renderování bez bumpmappingu - abychom mohli vidět ten rozdíl!

```

bool doMeshNoBumps(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity();// Reset matice

```

```

glTranslatef(0.0f,0.0f,z);

glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);

if (useMultitexture)
{
    glActiveTextureARB(GL_TEXTURE1_ARB);
    glDisable(GL_TEXTURE_2D);
    glActiveTextureARB(GL_TEXTURE0_ARB);
}

glDisable(GL_BLEND);
glBindTexture(GL_TEXTURE_2D,texture[filter]);
glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
glEnable(GL_LIGHTING);

doCube();

xrot += xspeed;
yrot += yspeed;

if (xrot>360.0f)
    xrot -= 360.0f;

if (xrot<0.0f)
    xrot += 360.0f;

if (yrot>360.0f)
    yrot -= 360.0f;

if (yrot<0.0f)
    yrot += 360.0f;

doLogo();// Nakonec loga

return true;
}

```

Vše co musí drawGLScene() udělat je rozhodnout jakou doMesh funkci zavolat.

```

bool DrawGLScene(GLvoid)// Všechno kreslení
{
    if (bumps)
    {
        if (useMultitexture && maxTexelUnits > 1)
            return doMesh2TexelUnits();
        else
            return doMesh1TexelUnits();
    }
    else
        return doMeshNoBumps();
}

```

Hlavní funkce Windows, přidány některé klávesy:

- E: přepínání Emboss/bumpmapový mód
- M: vypínání a zapínání multitexturingu
- B: vypínání a zapínání bumpmappingu, pouze v emboss módu
- F: přepínání filtrů, GL_NEAREST není vhodný pro bumpmapping
- KURSOROVÉ KLÁVESY: otáčení krychle

```

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow)
{
    // Začátek zůstává nezměněn
    if (keys['E'])
    {
        keys['E']=false;
        emboss=!emboss;
    }
}

```

```

    }

    if (keys['M'])
    {
        keys['M']=false;
        useMultitexture=((!useMultitexture) && multitextureSupported);
    }

    if (keys['B'])
    {
        keys['B']=false;
        bumps=!bumps;
    }

    if (keys['F'])
    {
        keys['F']=false;
        filter++;
        filter%=3;
    }

    if (keys[VK_PRIOR])
    {
        z-=0.02f;
    }

    if (keys[VK_NEXT])
    {
        z+=0.02f;
    }

    if (keys[VK_UP])
    {
        xspeed-=0.01f;
    }

    if (keys[VK_DOWN])
    {
        xspeed+=0.01f;
    }

    if (keys[VK_RIGHT])
    {
        yspeed+=0.01f;
    }

    if (keys[VK_LEFT])
    {
        yspeed-=0.01f;
    }
    // Konec také nezměněn
}

```

Ted' když jsme zvládli tento tutoriál, pár slov o generování textur a bumpmapových objektů. Předtím, než začnete programovat ambiciózní hry a budete se divit, proč bumpmapping není tak rychlý a nevypadá tak dobře, přečtěte si toto:

- Neměli byste používat textury 256x256 jako v této lekci. To vše hodně zpomalí. Používejte je pouze při demonstracích.
- Bumpmapovaná krychle není běžná. Točící se ještě méně. Důvodem je úhel pohledu: Čím ostřejší úhel, tím více optických chyb se kvůli filtrování objeví. Skoro všechny multifázové algoritmy tímto trpí. Abyste se vyhlí používaní velmi detailních textur, zredukujte úhly viditelnosti na minimum a předfiltrujte textury tak, aby dokonale sedly na tento rozptýl úhlů.
- Nejdříve byste měli mít barevnou texturu. Z ní se dá velmi snadno pomocí průměrného grafického programu udělat textura ve stupních šedi.
- Bumpmapa by měla být "ostřejší" a mít větší kontrast než barevná textura. Toho většinou docílíte použitím nějakého "sparinging filtru". Z počátku to možná bude vypadat divně, ale k dosažení kvalitního efektu je to nutné.
- Bumpmapa by se barvama měla blížit 50% šedé (RGB 127,127,127). Tato barva znamená hladký povrch, světlejší místa reprezentují rýhy. Tohoto můžete dosáhnout užitím histogramu v některých grafických

programech.

- Bumpmapa může být čtyřikrát menší než barevná textura bez vážného snížení kvality obrazu.

Poděkování:

- Michael I. Gold za dokumentaci o bumpmappingu
- Diego Tártara za ukázkový kód
- nVidia za ukázky na www
- NeHe za to, že mě naučil mnoho o OpenGL

napsal: Jens Schneider <schneide (zavináč) pool.informatik.rwth-aachen.de>
přeložil: Václav Slováček - Wessan <horizont (zavináč) host.sk>

Lekce 23 - Mapování textur na kulové quadratiky

Tento tutoriál je napsán na bázi lekce 18. V lekci 15 (Mapování textur na fonty) jsem psal o automatickém mapování textur. Vysvětlil jsem jak můžeme poprosit OpenGL o automatické generování texturových koordinátů, ale protože lekce 15 byla celkem skromná, rozhodl jsem se přidat mnohem více detailů o této technice.

Mapování kulového prostředí (Sphere Environment Mapping) je rychlá cesta pro přidání zrcadlení na kovové nebo zrcadlové objekty. Třebaže není tak přesná jako skutečné zrcadlení nebo jako krychlová mapa prostředí (cube environment map) je o hodně rychlejší. Jako základ použijeme kód z lekce 18, ale nepoužijeme žádnou z předchozích textur. Použijeme jednu kulovou mapu (sphere map) a jeden obrázek pro pozadí.

Než začneme... Red Book definuje kulovou mapu jako obraz scény na kovové kouli z nekonečné vzdálenosti a nekonečného ohniskového bodu. To je idální a ve skutečném životě nemožné. Nejlepší způsob, bez použití čoček rybího oka (fish eye lens), který jsem našel je použití programu Adobe Photoshop:

Nejdříve budeme potřebovat obrázek prostředí, které chceme namapovat na kouli. Otevřeme obrázek v Adobe Photoshopu a vybereme celý obrázek. Zkopírujeme obrázek a vytvoříme nový obrázek PSD (Photoshop formát). Nový obrázek by měl být stejné velikosti jako obrázek který jsme právě zkopírovali. Vložíme kopii původního obrázku do nového. Důvodem proč děláme kopii je, že tak může Photoshop aplikovat své filtry. Namísto kopírování obrázku můžeme vybrat mód z lokálního menu (na kliknutí pravého tlačítka myši) a zvolit mód RGB. Poté budou dostupné všechny filtry.

Dále potřebujeme změnit velikost obrázku tak že bude mocninou dvou. Pamatujte, že abyste mohli použít obrázek jako texturu musí mít rozměry 128x128, 256x256 atd. V menu image tedy vybereme image size, odškrtneme constraint proportions (zachovat poměr stran) a změníme velikost obrázku na platnou velikost textury. Pokud má váš obrázek velikost 100x90 je lepší vytvořit texturu o velikosti 128x128 než 64x64. Vytvářením menšího obrázku ztratíte hodně detailů.

Jako poslední vybereme menu filter (filtry) a v něm distort (zdeformovat) a použijte spherize modifier (modifikátor koule). Můžeme vidět, že střed obrázku je nafouklý jako balón. V normálním kulovém mapování by byla vnější plocha černá, ale to nemá skutečný vliv. Uložíme obrázek jako BMP a jsme připraveni k programování.



V této lekci nebudeme přidávat žádné nové globální proměnné, ale pouze upravíme index pole pro uložení šesti textur.

```
GLuint texture[6]; // Šest textur
```

Dále modifikujeme funkci LoadGLTextures(), abychom mohli nahrát 2 textury a aplikovat 3 filtry. Jednoduše dvakrát projdeme cyklem a v každém průchodu vytvoříme 3 textury pokaždé s jiným filtrovacím módem. Skoro celý tento kód je nový nebo modifikovaný.

```
int LoadGLTextures() // Loading bitmap a konverze na textury
{
    int Status=FALSE; // Indikuje chyby
    AUX_RGBImageRec *TextureImage[2]; // Ukládá dvě bitmapy
```

```

memset(TextureImage,0,sizeof(void *)*2);// Vynuluje paměť

// Nahraje bitmapy a kontroluje vzniklé chyby
if ((TextureImage[0]=LoadBMP("Data/BG.bmp")) &&// Textura pozadí
(TextureImage[1]=LoadBMP("Data/Reflect.bmp")))// Textura kulové mapy (sphere map)
{
    Status=TRUE;// Vše je bez problémů

    glGenTextures(6, &texture[0]);// Generuje šest textur

    for (int loop=0; loop<=1; loop++)
    {
        // Vytvoří nelineárně filtrovanou texturu
        glBindTexture(GL_TEXTURE_2D, texture[loop]);// Textury 0 a 1
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage
[loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);

        // Vytvoří lineárně filtrovanou texturu
        glBindTexture(GL_TEXTURE_2D, texture[loop+2]);// Textury 2 a 3
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage
[loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);

        // Vytvoří mipmapovanou texturu
        glBindTexture(GL_TEXTURE_2D, texture[loop+4]);// Textury 2 a 3
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri
        (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);

        gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[loop]->sizeX, TextureImage
[loop]->sizeY, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
    }

    for (loop=0; loop<=1; loop++)
    {
        if (TextureImage[loop])// Pokud obrázek existuje
        {
            if (TextureImage[loop]->data)// Pokud existují data obrázku
            {
                free(TextureImage[loop]->data);// Uvolní paměť obrázku
            }

            free(TextureImage[loop]);// Uvolní strukturu obrázku
        }
    }

    return Status;// Oznamí případné chyby
}

```

Trochu upravíme kód kreslení krychle. Namísto použití hodnot 1.0 a -1.0 pro normály, použijeme 0.5 a -0.5. Změnou hodnot normál můžeme měnit velikost odrazové mapy dovnitř a ven. Pokud je hodnota normály velká, odražený obrázek je větší a mohl by se zobrazovat čtverečkovane. Snížením hodnoty normál na 0.5 a -0.5 je obrázek trochu zmenšen, takže obrázek odražený na krychli nevypadá tak čtverečkovane. Nastavením příliš malých hodnot získáme nežádoucí výsledky.

```

GLvoid glDrawCube()
{
    glBegin(GL_QUADS);
    // Přední stěna
    glNormal3f( 0.0f, 0.0f, 0.5f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
    // Zadní stěna

```

```

    glNormal3f( 0.0f, 0.0f,-0.5f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    // Vrchní stěna
    glNormal3f( 0.0f, 0.5f, 0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
    // Spodní stěna
    glNormal3f( 0.0f,-0.5f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
    // Pravá stěna
    glNormal3f( 0.5f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
    // Levá stěna
    glNormal3f(-0.5f, 0.0f, 0.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
}

```

Do InitGL přidáme volání dvou nových funkcí. Tyto dvě volání nastaví mód generování textur na S a T pro kulové mapování (sphere mapping). Texturové koordináty S, T, R a Q souvisí s koordinátami objektu x, y, z a w. Pokud používáte jednorozměrnou (1D) texturu, použijete souřadnici S. Pokud použijete dvourozměrnou texturu použijete souřadnice S a T.

Takže následující kód říká OpenGL jak automaticky generovat S a T koordináty na kulově mapovaném (sphere-mapping) vzorci. Koordináty R a Q jsou obvykle ignorovány. Koordinát Q může být použit pro pokročilé techniky mapování textur a koordinát R může být užitečný až bude do OpenGL přidáno mapování 3D textur. Ale pro teď budeme koordináty R a Q ignorovat. Koordinát S běží horizontálně přes čelo našeho polygonu a T zase vertikálně.

```

// Funkce InitGL()
// Nastavení módu generování textur pro S koordináty pro kulové mapování
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
// Nastavení módu generování textur pro T koordináty pro kulové mapování
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);

```

Máme téměř hotovo. Vše co musíme ještě udělat je nastavit vykreslování. Odstranil jsem několik typů kvadratiků, protože nepracovali dobře s mapováním prostředí (environment mapping). Zprvė potřebujeme povolit generování textur. Potom vybereme odrazovou texturu (kulovou mapu - sphere map) a vykreslíme náš objekt. Před vykreslením pozadí vypneme kulové mapování. Všimněte si, že příkaz glBindTexture() může vypadat docela složitě. Vše co děláme je výběr filtru pro kreslení naší kulové mapy nebo obrázku pozadí.

```

int DrawGLScene(GLvoid) // Všechno kreslení
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f,0.0f,z);

    glEnable(GL_TEXTURE_GEN_S); // Povolí generování texturových koordinátů S
    glEnable(GL_TEXTURE_GEN_T); // Povolí generování texturových koordinátů T

    glBindTexture(GL_TEXTURE_2D, texture[filter+(filter+1)]); // Zvolí texturu kulové
    mapy
    glPushMatrix();
    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);

```

```

switch(object)// Vybere, co se bude kreslit
{
    case 0:
        glDrawCube();// Krychle
        break;

    case 1:
        glTranslatef(0.0f,0.0f,-1.5f);// Vycentrování
        gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32);// Válec
        break;

    case 2:
        gluSphere(quadratic,1.3f,32,32);// Koule
        break;

    case 3:
        glTranslatef(0.0f,0.0f,-1.5f);// Vycentrování
        gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32);// Kužel
        break;
}

glPopMatrix();
glDisable(GL_TEXTURE_GEN_S);// Vypne automatické generování koordinátů S
glDisable(GL_TEXTURE_GEN_T);// Vypne automatické generování koordinátů T

glBindTexture(GL_TEXTURE_2D, texture[filter*2]);// Zvolí texturu pozadí
glPushMatrix();
glTranslatef(0.0f, 0.0f, -24.0f);

glBegin(GL_QUADS);
    glNormal3f( 0.0f, 0.0f, 1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-13.3f, -10.0f, 10.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 13.3f, -10.0f, 10.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 13.3f, 10.0f, 10.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-13.3f, 10.0f, 10.0f);
glEnd();

glPopMatrix();

xrot+=xspeed;
yrot+=yspeed;
return TRUE;
}

```

Poslední věc, kterou v této lekci uděláme je upravení kódu kontrolujícího stisk mezerníku - odstranili jsme disky.

```

    if (keys[' '] && !sp)
    {
        sp=TRUE;
        object++;

        if(object>3)
            object=0;
    }

```

A máme hotovo. Umíme vytvářet skutečně působivé efekty s použitím zrcadlení okolí na objektu - například téměř přesného odrazu pokoje. Původně jsem chtěl také ukázat, jak vytvářet krychlové mapování prostředí, ale moje aktuální videokarta ho nepodporuje. Možná za měsíc nebo tak nějak, až si koupím GeForce2 :-]. Mapování okolí jsem se naučil sám (hlavně proto, že jsem o tom nemohl najít téměř žádné informace), takže pokud je v tomto tutoriálu něco nepřesné, pošlete mi email nebo uvědomte NeHe-ho. Díky a hodně štěstí.

napsal: GB Schmick - TipTup
přeložil: Milan Turek <nalim.kerut (zavináč) email.cz>

Lekce 24 - Vypis OpenGL rozšíření, ořezávací testy a textury z TGA obrázků

V této lekci se naučíte, jak zjistit, která OpenGL rozšíření (extensions) podporuje vaše grafická karta. Vypíšeme je do středu okna, se kterým budeme moci po stisku šipek rolovat. Použijeme klasický 2D texturový font s tím rozdílem, že texturu vytvoříme z TGA obrázku. Jeho největšími přednostmi jsou jednoduchá práce a podpora alfa kanálu. Odbouráním bitmap už nebudeme muset inkluodovat knihovnu glaux.

Tento tutoriál je daleko od prezentace grafické nádherly, ale naučíte se několik nových věcí. Pár lidí se mě ptalo na OpenGL rozšíření a na to, jak zjistit, které jsou podporovány konkrétním typem grafické karty. Mohu směle říci, že s tímto po dočtení nebudete mít nejmenší problémy. Také se dozvíte, jak rolovat částí scény bez toho, aby se ovlivnilo její okolí. Použijeme ořezávací testy (scissor testing). Dále si ukážeme, jak vykreslovat linky pomocí line strips a co je důležitější, kompletně odbouráme knihovnu glaux, kterou jsme používali kvůli texturám z bitmapových obrázků. Budeme používat Targa (TGA) obrázky, se kterými se snadno pracuje a které podporují alfa kanál.

Začneme programovat. První věcí, které si všimneme u vkládání hlavičkových souborů je, že neinkluujeme knihovnu glaux (glaux.h). Také nepřilikujeme soubor glaux.lib. Už nebudeme pracovat s bitmapami, takže tyto soubory v projektu nepotřebujeme.

```
#include <windows.h> // Hlavičkový soubor pro Windows

#include <stdio.h> // Hlavičkový soubor pro standardní vstup/výstup
#include <stdarg.h> // Hlavičkový soubor pro funkce s proměnným počtem parametrů
#include <string.h> // Hlavičkový soubor pro práci s řetězci

#include <gl\gl.h> // Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h> // Hlavičkový soubor pro Glu32 knihovnu

HDC hDC = NULL; // Privátní GDI Device Context
HGLRC hRC = NULL; // Trvalý Rendering Context
HWND hWnd = NULL; // Obsahuje Handle našeho okna
HINSTANCE hInstance; // Obsahuje instanci aplikace

bool keys[256]; // Pole pro ukládání vstupu z klávesnice
bool active = TRUE; // Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE; // Ponese informaci o tom, zda je program ve fullscreenu
```

Přidáme proměnné. Scroll bude použito pro rolování částí scény nahoru a dolů. Druhá proměnná, maxtokens, bude ukládat záznam kolik rozšíření je podporováno grafickou kartou. Base už tradičně ukazuje na display listy fontu. Do swidth a sheight nagrabujeme aktuální velikost okna, pomohou nám vypočítat koordináty pro ořezání oblasti okna, které umožní rolování.

```
int scroll; // Pro rolování okna
int maxtokens; // Počet podporovaných rozšíření

GLuint base; // Základní display list fontu

int swidth; // Šířka ořezané oblasti
int sheight; // Výška ořezané oblasti
```

Napišeme strukturu, která bude ukládat informace o nahrávaném TGA obrázku. Pointer imageData bude ukazovat na data, ze kterých vytvoříme obrázek. Bpp označuje barevnou hloubku (bits per pixel), která může být 24 nebo 32, podle přítomnosti alfa kanálu. Width a height definuje rozměry. Do texID vytvoříme texturu. Celou strukturu nazveme TextureImage.

```
typedef struct // Struktura textury
{
    GLubyte *imageData; // Data obrázku
    GLuint bpp; // Barevná hloubka obrázku
    GLuint width; // Šířka obrázku
    GLuint height; // Výška obrázku

    GLuint texID; // Vytvořená textura
} TextureImage; // Jméno struktury
```

V tomto programu budeme používat pouze jednu texturu, takže vytvoříme pole textur o velikosti jedna.

```
TextureImage textures[1]; // Jedna textura
```

Na řadu přichází asi nejobtížnější část - nahrávání TGA obrázku a jeho konvertování na texturu. Musím ještě poznamenat, že kód následující funkce umožňuje loadovat buď 24 nebo 32 bitové **nekomprimované** TGA soubory. Zabralo dost času zprovoznit kód, který by pracoval s oběma typy. Nikdy jsem neřekl, že jsem génius. Rád bych poukázal, že úplně všechno není z mé hlavy. Spoustu opravdu dobrých nápadů jsem získal pročítáním internetu. Pokusil jsem se je zkombinovat do funkčního kódu, který pracuje s OpenGL. Nic snadného, nic extrémně složitého!

Funkci předáváme dva parametry. První ukazuje do paměti, kam uložíme texturu. Druhý určuje diskovou cestu k souboru, který chceme nahrát.

```
bool LoadTGA(TextureImage *texture, char *filename) // Do paměti nahraje TGA soubor
{
```

Pole TGAheader[] definuje 12 bytů. Porovnáme je s prvními 12 bity, které načteme z TGA souboru - TGAcompare[], abychom se ujistili, že je to opravdu Targa obrázek a ne nějaký jiný.

```
GLubyte TGAheader[12] = { 0,0,2,0,0,0,0,0,0,0,0,0 }; // Nekomprimovaná TGA hlavička
GLubyte TGAcompare[12]; // Pro porovnání TGA hlavičky
```

Header[] ukládá prvních šest DŮLEŽITÝCH bytů z hlavičky souboru (šířka, výška, barevná hloubka).

```
GLubyte header[6]; // Prvních 6 užitečných bytů z hlavičky
```

Do bytesPerPixel přiřadíme výsledek operace, kdy vydělíme barevnou hloubku v bitech osmi, abychom získali barevnou hloubku v bytech na pixel. ImageSize definuje počet bytů, které jsou zapotřebí k vytvoření obrázku (šířka*výška*barevná hloubka).

```
GLuint bytesPerPixel; // Počet bytů na pixel použitý v TGA souboru
GLuint imageSize; // Ukládá velikost obrázku při alokování RAM
```

Temp umožní prohodit byty dále v programu. A konečně poslední proměnnou použijeme ke zvolení správného parametru při vytváření textury. Bude záviset na tom, zda je TGA 24 nebo 32 bitová. V případě 24 bitů předáme GL_RGB a máme-li 32 bitový obrázek použijeme GL_RGBA. Implicitně předpokládáme, že je obrázek 32 bitový, tudíž do type přiřadíme GL_RGBA.

```
GLuint temp; // Pomocná proměnná
GLuint type = GL_RGBA; // Implicitním GL módem je RGBA (32 BPP)
```

Pomocí funkce fopen() otevřeme TGA soubor filename pro čtení v binárním módu (rb). Následuje větvení if, ve kterém děláme hned několik věcí najednou. Nejprve testujeme jestli soubor obsahuje data. Pokud tam žádná nejsou, vrátíme false. Obsahuje-li informace, přečteme prvních dvanáct bytů do TGAcompare. Použijeme funkci fread(), která po jednom bytu načte ze souboru file dvanáct bytů (sizeof(TGAcompare)) a výsledek uloží do TGAcompare. Vrací počet přečtených bytů, které porovnáme se sizeof(TGAcompare). Mělo by jich být, jak tušíte :-), dvanáct. Pokud jsme bez potíží došli až tak daleko, porovnáme funkcí memcmp() pole TGAheader a TGAcompare. Nebudou-li stejné zavřeme soubor a vrátíme false, protože se nejedná o TGA obrázek. Do header nakonec načteme dalších šest bytů. Při chybě opět zavřeme soubor a funkci ukončíme.

```
FILE *file = fopen(filename, "rb"); // Otevře TGA soubor

if(file == NULL || // Existuje soubor?
    fread(TGAcompare,1,sizeof(TGAcompare),file) != sizeof(TGAcompare) || // Podařilo
    se načíst 12 bytů?
    memcmp(TGAheader,TGAcompare,sizeof(TGAheader)) != 0 || // Mají potřebné hodnoty?
    fread(header,1,sizeof(header),file) != sizeof(header)) // Pokud ano, načte
    dalších šest bytů
{
    if (file == NULL) // Existuje soubor?
        return false; // Konec funkce
    else
    {
        fclose(file); // Zavře soubor
        return false; // Konec funkce
    }
}
```

Pokud program prošel kódem bez chyby máme dost informací pro definování některých proměnných. První bude šířka obrázku. Problém spočívá v tom, že toto číslo je rozděleno do dvou bytů. Nižší byte může nabývat 256 hodnot (8 bitů), takže vynásobíme vyšší byte 256 a k němu přičteme nižší. Získali jsme šířku obrázku. Stejným postupem dostaneme i výšku, akorát použijeme jiné indexy v poli.

```
texture->width = header[1] * 256 + header[0]; // Získá šířku obrázku
texture->height = header[3] * 256 + header[2]; // Získá výšku obrázku
```

Zkontrolujeme jestli je šířka i výška větší než nula. Pokud ne zavřeme soubor a vrátíme false. Zároveň zkontrolujeme i barevnou hloubku, kterou hledáme v header[4]. Musí být buď 24 nebo 32 bitová.

```
if(texture->width <= 0 ||// Platná šířka?
    texture->height <= 0 ||// Platná výška?
    (header[4] != 24 && header[4] != 32)) // Platná barevná hloubka?
{
    fclose(file); // Zavře soubor
    return false; // Konec funkce
}
```

Spočítali a zkontrolovali jsme šířku a výšku, můžeme přejít k barevné hloubce v bitech a bytech a velikosti paměti potřebné k uložení dat obrázku. Už víme, že v header[4] je barevná hloubka v bitech na pixel. Přičítáme ji do bpp. Jeden byte se skládá z 8 bitů. Z toho plyne, že barevnou hloubku v bytech získáme dělením bpp osmi. Velikost dat obrázku získáme vynásobením šířky, výšky a bytů na pixel.

```
texture->bpp = header[4]; // Bitů na pixel (24 nebo 32)
bytesPerPixel = texture->bpp / 8; // Byty na pixel
imageSize = texture->width * texture->height * bytesPerPixel; // Velikost paměti pro data obrázku
```

Potřebujeme alokovat paměť pro data obrázku. Funkci malloc() předáme požadovanou velikost. Měla by vrátit ukazatel na zabrané místo v RAM. Následující if má opět několik úloh. V prvé řadě testuje správnost alokace. Pokud při ní něco nevyšlo, ukazatel má hodnotu NULL. V takovém případě zavřeme soubor a vrátíme false. Nicméně pokud se alokace podařila, tak pomocí fread() načteme data obrázku a uložíme je do právě alokované paměti. Pokud se data nepodaří zkopírovat, uvolníme paměť, zavřeme soubor a ukončíme funkci.

```
texture->imageData = (GLubyte *)malloc(imageSize); // Alokace paměti pro data obrázku
if(texture->imageData == NULL ||// Podařilo se paměť alokovat?
    fread(texture->imageData, 1, imageSize, file) != imageSize) // Podařilo se kopírování dat?
{
    if(texture->imageData != NULL) // Byla data nahrána?
        free(texture->imageData); // Uvolní paměť
    fclose(file); // Zavře soubor
    return false; // Konec funkce
}
```

Pokud se až do teď nestalo nic, čím bychom ukončovali funkci, máme vyhráno. Stojí před námi, ale ještě jeden úkol. Formát TGA specifikuje pořadí barevných složek BGR (modrá, zelená, červená) narozdíl od OpenGL, které používá RGB. Pokud bychom neprohodili červenou a modrou složku, tak všechno, co má být v obrázku modré by bylo červené a naopak. Deklarujeme cyklus, jehož řídicí proměnná i nabývá hodnot od nuly do velikosti obrázky. Každým průchodem se zvětšuje o 3 nebo o 4 v závislosti na barevné hloubce. (24/8=3, 32/8=4). Uvnitř cyklu prohodíme R a B složky. Modrá je na indexu i a červená i+2. Modrá by byla na i+1, ale s tou nic neděláme, protože je umístěná správně.

```
for(GLuint i=0; i < int(imageSize); i += bytesPerPixel) // Prochází data obrázku
{
    temp = texture->imageData[i]; // B uložíme do pomocné proměnné
    texture->imageData[i] = texture->imageData[i + 2]; // R je na správném místě
    texture->imageData[i + 2] = temp; // B je na správném místě
}
```

Po této operaci máme v paměti uložen obrázek TGA ve formátu, který podporuje OpenGL. Nic nám nebrání, abychom zavřeli soubor. Už ho k ničemu nepotřebujeme.

```
fclose(file); // Zavře soubor
```

Můžeme začít vytvářet texturu. Tento postup je v principu úplně stejný, jako ten, který jsme používali v minulých tutoriálech. Požádáme OpenGL o vygenerování jedné textury na adrese texture[0].texID, kterou jsme získali předáním parametru ve funkci glGenTextures(). Pokud bychom chtěli vytvořit druhou texturu z jiného obrázku TGA, tak se tato funkce vůbec nezmění. V glGenTextures() bychom provedli volání dvakrát, ale s jinými parametry. Programujeme obecněji...

```
glGenTextures(1, &texture[0].texID); // Generuje texturu
```

Zvolíme právě vytvářenou texturu za aktuální a nastavíme jí lineární filtrování pro zmenšení i zvětšení.

```

glBindTexture(GL_TEXTURE_2D, texture[0].texID); // Zvolí texturu

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Lineární
filtrování
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Lineární
filtrování

```

Zkontrolujeme, jestli je textura 24 nebo 32 bitová. V prvním případě nastavíme type na GL_RGB (bez alfa kanálu), jinak ponecháme implicitní hodnotu GL_RGBA (s alfa kanálem). Pokud bychom test neprovedli, program by se s největší pravděpodobností zhroutil.

```

if (texture[0].bpp == 24) // Je obrázek 24 bitový?
{
    type = GL_RGB; // Nastaví typ na GL_RGB
}

```

Ted' konečně sestavíme texturu. Jako obvykle, tak i tentokrát, použijeme funkci glTexImage2D(). Místo ručního zadání typu textury (GL_RGB, GL_RGBA) předáme hodnotu pomocí proměnné. Jednoduše řečeno: Program sám detekuje, co má předat.

```

glTexImage2D(GL_TEXTURE_2D, 0, type, texture[0].width, texture[0].height, 0, type,
GL_UNSIGNED_BYTE, texture[0].imageData); // Vytvoří texturu

return true; // Všechno je v pořádku
}

```

ReSizeGLScene() nastavuje pravoúhrou projekci. Souřadnice [0; 1] jsou levým horním rohem okna a [640; 480] pravým dolním. Dostáváme rozlišení 640x480. Na začátku nastavíme globální proměnné swidth a sheight na aktuální rozměry okna. Při každém přesunutí nebo změně velikosti okna se aktualizují. Ostatní kód znáte.

```

GLvoid ReSizeGLScene(GLsizei width, GLsizei height) // Změna velikosti a inicializace
OpenGL okna
{
    swidth = width; // Šířka okna
    sheight = height; // Výška okna

    if (height == 0) // Zabezpečení proti dělení nulou
    {
        height = 1; // Nastaví výšku na jedna
    }

    glViewport(0,0,width,height); // Resetuje aktuální nastavení
    glMatrixMode(GL_PROJECTION); // Zvolí projekční matici
    glLoadIdentity(); // Reset matice

    glOrtho(0.0f,640,480,0.0f,-1.0f,1.0f); // Pravoúhlá projekce 640x480, [0; 0] vlevo
    nahoře
    glMatrixMode(GL_MODELVIEW); // Zvolí matici Modelview
    glLoadIdentity(); // Reset matice
}

```

Inicializace OpenGL se minimalizovala. Zůstala z ní jenom kostra. Nahrajeme TGA obrázek a vytvoříme z něj texturu. V prvním parametru je určeno, kam ji uložíme a v druhém disková cesta k obrázku. Vráť-li funkce z jakéhokoli důvodu false, inicializace se přeruší, program zobrazí chybovou zprávu a ukončí se. Pokud byste chtěli nahrát druhou nebo i další texturu použijte volání několik. Podmínka se logicky ORuje.

```

int InitGL(GLvoid) // Nastavení OpenGL
{
    if (!LoadTGA(&textures[0], "Data/Font.TGA")) // Nahraje texturu fontu z TGA obrázku
    {
        return false; // Při chybě ukončí program
    }
}

```

Po úspěšném nahrání textury vytvoříme font. Je důležité upozornit, že se BuildFont() musí volat až po funkci LoadTGA(), protože používá jí vytvořenou texturu. Dále nastavíme vyhlazené stínování, černé pozadí, povolíme mazání depth bufferu a zvolíme texturu fontu.

```

BuildFont(); // Sestaví font

glShadeModel(GL_SMOOTH); // Vyhlazené stínování
glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
glClearDepth(1.0f); // Nastavení hloubkového bufferu
glBindTexture(GL_TEXTURE_2D, textures[0].texID); // Zvolí texturu

```



```

    return TRUE;// Inicializace v pořádku
}

```

Přejdeme k vykreslování. Začneme deklarováním proměnných. O ukazateli token zatím jen tolik, že bude ukládat řetězec jednoho podporovaného rozšíření a cnt je pro zjištění jeho pořadí.

```

int DrawGLScene(GLvoid) // Vykreslování
{
    char* token;// Ukládá jedno rozšíření
    int cnt = 0;// Čítač rozšíření

```

Smažeme obrazovku a hloubkový buffer. Potom nastavíme barvu na středně tmavě červenou a do horní části okna vypíšeme slova Renderer (jméno grafické karty), Vendor (její výrobce) a Version (verze). Důvod, proč nejsou všechny umístěny 50 pixelů od okraje na ose x, je ten, že je nezarovnáváme doleva, ale doprava.

```

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smaže obrazovku a hloubkový
    buffer

    glColor3f(1.0f,0.5f,0.5f);// Červená barva
    glPrint(50,16,1,"Renderer");// Výpis nadpisu pro grafickou kartu
    glPrint(80,48,1,"Vendor");// Výpis nadpisu pro výrobce
    glPrint(66,80,1,"Version");// Výpis nadpisu pro verzi

```

Změníme červenou barvu na oranžovou a nagrabujeme informace z grafické karty. Použijeme funkci glGetString(), která vrátí požadované řetězce. Kvůli glPrint() přetypujeme výstup funkce na char*. Výsledek vypíšeme doprava od nadpisů.

```

    glColor3f(1.0f,0.7f,0.4f);// Oranžová barva
    glPrint(200,16,1,(char *)glGetString(GL_RENDERER));// Výpis typu grafické karty
    glPrint(200,48,1,(char *)glGetString(GL_VENDOR));// Výpis výrobce
    glPrint(200,80,1,(char *)glGetString(GL_VERSION));// Výpis verze

```

Definujeme modrou barvu a dolů na scénu vypíšeme NeHe Productions.

```

    glColor3f(0.5f,0.5f,1.0f);// Modrá barva
    glPrint(192,432,1,"NeHe Productions");// Výpis NeHe Productions

```

Kolem právě vypsáního textu vykreslíme bílý rámeček. Resetujeme matici, protože v glPrint() se volají funkce, které ji mění. Potom definujeme bílou barvu.

```

    glLoadIdentity();// Reset matice
    glColor3f(1.0f,1.0f,1.0f);// Bílá barva

```

Vykreslování linek pomocí GL_LINE_STRIP je velmi jednoduché. První bod definujeme úplně vpravo, 63 pixelů (480-417=63) nad spodním okrajem okna. Druhý vertex umístíme ve stejné výšce, ale vlevo. OpenGL je spojí přímkou. Třetí bod posuneme dolů do levého dolního rohu. OpenGL opět zobrazí linku, tentokrát mezi druhým a třetím bodem. Čtvrtý bod patří do pravého dolního rohu a k pátému projedeme výchozím vertexem nahoru. Ukončíme triangle strip, abychom mohli začít vykreslovat z nové pozice a stejným způsobem vykreslíme druhou část rámečku, ale tentokrát nahoře.

Asi jste pochopili, že pokud vykreslujeme více na sebe navazujících přímk, tak LINE_STRIP ušetří spoustu zbytečného kódu, který vzniká opakovaným definováním vertexů při obyčejném GL_LINES.



```

glBegin(GL_LINE_STRIP);// Začátek kreslení linek
    glVertex2d(639,417);// 1
    glVertex2d(0,417);// 2
    glVertex2d(0,480);// 3
    glVertex2d(639,480);// 4
    glVertex2d(639,128);// 5
glEnd();// Konec kreslení

glBegin(GL_LINE_STRIP);// Začátek kreslení linek
    glVertex2d(0,128);// 6
    glVertex2d(639,128);// 7
    glVertex2d(639,1);// 8
    glVertex2d(0,1);// 9

```

```

    glVertex2d(0,417); // 10
    glEnd(); // Konec kreslení

```

Nám neznámá funkce `glScissor(x, y, v, š)` vytváří něco, co by se dalo popsat jako okno. Pokud zapneme `GL_SCISSOR_TEST`, bude se ořezávat okolí této části obrazovky, tudíž se objekty budou moci vykreslovat pouze uvnitř definovaného obdélníku. Určíme ho parametry předanými funkcí. V našem případě je to první pixel na ose x ve výšce 13,5% (0,135...f) od spodního okraje. dále bude 638 pixelů široký (width-2) a 59,7% (0,597...f) výšky okna vysoký. Druhým řádkem povolíme ořezávací testy. Můžete se pokusit vykreslit obrovský obdélník přes celé okno, ale uvidíte pouze část v neořezané oblasti. zbytek dosud nakreslené scény zůstane nezměněn. Perfektní příkaz!

```

glScissor(1, int(0.135416f*sheight), swidth-2, int(0.597916f*sheight)); // Definování
ořezávací oblasti
glEnable(GL_SCISSOR_TEST); // Povolí ořezávací testy

```

Na řadu přichází asi nejtěžší část této lekce - vypsání podporovaných OpenGL rozšíření. V první fázi je musíme získat. Pomocí funkce `malloc()` alokujeme buffer pro řetězec znaků text. Předává se jí velikost požadované paměti. `strlen()` spočítá počet znaků řetězce vráceného `glGetString(GL_EXTENSIONS)`. Přičteme k němu ještě jeden znak pro `'\0'`, který uzavírá každý c-ěčkový řetězec. `strcpy()` zkopíruje řetězec podporovaných rozšíření do proměnné text.

```

char* text = (char *)malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1); // Alokace
paměti pro řetězec
strcpy(text, (char *)glGetString(GL_EXTENSIONS)); // Zkopíruje seznam rozšíření do
text

```

Nyní jsme do text nagrabovali z grafické karty řetězec, který vypadá nějak takto: "GL_ARB_multitexture GL_EXT_abgr GL_EXT_bgra". Pomocí `strtok()` z něj vyjmeme v pořadí první rozšíření. Funkce pracuje tak, že prochází řetězec a v případě, že najde mezeru zkopíruje příslušnou část z text do token. První hodnota token tedy bude "GL_ARB_multitexture". Zároveň se však změní i text. První mezera se nahradí oddělovačem. Více dále.

```

token = strtok(text, " "); // Získá první podřetězec

```

Vytvoříme cyklus, který se zastaví tehdy, když v token nebudou už žádné další informace - bude se rovnat NULL. Každým průchodem inkrementujeme čítač a zkontrolujeme, jestli je jeho hodnota větší než `maxtokens`. Touto cestou velice snadno získáme maximální hodnotu v čítači, kterou využijeme při rolování po stisku kláves.

```

while(token != NULL) // Prochází podporovaná rozšíření
{
    cnt++; // Inkrementuje čítač

    if (cnt > maxtokens) // Je maximum menší než hodnota čítače?
    {
        maxtokens = cnt; // Aktualizace maxima
    }
}

```

V této chvíli máme v token uložené první rozšíření. Jeho pořadové číslo napíšeme zeleně do levé části okna. Všimněte si, že ho na ose x napíšeme na souřadnici 0. Tím bychom mohli zlikvidovat levý (bílý) rámeček, který jsme už vykreslili, ale protože máme zapnuté ořezávání, pixely na nule nebudou modifikovány. Na ose y začínáme kreslit na 96. Abychom nevykreslovali všechno na sebe, přičítáme pořadí násobené výškou textu (`cnt*32`). Při vypisování prvního rozšíření se `cnt==1` a text se nakreslí na `96+(32*1)=128`. U druhého je výsledkem 160. Také odečítáme scroll. Implicitně se rovná nule, ale po stisku šipek se jeho hodnota mění. Umožníme tím rolování ořezaného okna, do kterého se vejde celkem devět řádek (výška okna/výška textu = $288/32 = 9$). Změnou scrollu můžeme změnit offset textu a tím ho posunout nahoru nebo dolů. Efekt je podobný filmovému projektoru. Film roluje tak, aby v jednom okamžiku byl vidět vždy jen jeden frame. Nemůžete vidět oblast nad nebo pod ním i když máte větší plátno. Objektiv sehrává stejnou roli jako ořezávací testy.

```

glColor3f(0.5f,1.0f,0.5f); // Zelená barva
glPrint(0, 96+(cnt*32)-scroll, 0, "%i", cnt); // Pořadí aktuálního rozšíření

```

Po vykreslení pořadového čísla zaměníme zelenou barvu za žlutou a konečně vypíšeme text uložený v proměnné token. Vlevo se začne na padesátém pixelu.

```

glColor3f(1.0f,1.0f,0.5f); // Žlutá barva
glPrint(50,96+(cnt*32)-scroll,0,token); // Vypíše jedno rozšíření

```

Po zobrazení prvního rozšíření potřebujeme připravit půdu pro další průchod cyklem. Nejprve zjistíme, jestli je v text ještě nějaké další rozšíření. Namísto opětovného volání `token = strtok(text, " ")`, napíšeme `token = strtok(NULL, " ")`; NULL určuje, že se má hledat DALŠÍ podřetězec a ne všechno provádět od znova. V našem příkladě jsem výše napsal, že se mezera nahradí oddělovačem - "GL_ARB_multitextureoddělovačGL_EXT_abgr GL_EXT_bgra". Najdeme tedy oddělovač a až od něj se bude hledat další mezera. Poté se do token zkopíruje podřetězec mezi oddělovačem a mezerou (GL_EXT_abgr) a text bude modifikován na "GL_ARB_multitextureoddělovačGL_EXT_abgoddělovačGL_EXT_bgra". Po dosažení konce textu se token nastaví na NULL a cyklus se ukončí.

```

    token = strtok(NULL, " "); // Najde další rozšíření
}

```

Tím jsme ukončili vykreslování, ale ještě nám zbývá po sobě uklidit. Vypneme ořezávací testy a uvolníme dynamickou paměť - informace získané pomocí `glGetString(GL_EXTENSIONS)` uložené v RAM. Příště až budeme volat `DrawGLScene()` se paměť opět alokuje a provedou se znovu všechny rozborů řetězců.

```

glDisable(GL_SCISSOR_TEST); // Vypne ořezávací testy

free(text); // Uvolní dynamickou paměť

```

Příkaz `glFlush()` není bezpodmínečně nutný, ale myslím, že je dobrý nápad se o něm zmínit. Nejjednodušší vysvětlení je takové, že oznámí OpenGL, aby dokončilo, co právě dělá (některé grafické karty např. používají vyrovnávací paměti, jejichž obsah se tímto pošle na výstup). Pokud si někdy všimnete mihotání nebo blikání polygonů, zkuste přidat na konec všeho vykreslování volání `glFlush()`. Vyprázdní renderovací pipeline a tím zamezí mihotání, které vzniká tehdy, když program nemá dostatek času, aby dokončil rendering.

```

glFlush(); // Vyprázdni renderovací pipeline

return TRUE; // Všechno v pořádku
}

```

Na konec `KillGLWindow()` přidáme volání `KillFont`, které smaže display listy fontu.

```

// Konec KillGLWindow()
KillFont(); // Smaže font
}

```

V programu testujeme stisk šipky nahoru a dolů. V obou případech přičteme nebo odečteme od `scroll` dvojku, ale pouze tehdy, pokud bychom nerolovali mimo okno. U šipky nahoru je situace jednoduchá - nula je vždy nejnižší možné rolování. Maximum u šipky dolů získáme násobením výšky řádku a počtu rozšíření. Devítku odečítáme, protože se v jednom okamžiku vejde na scénu devět řádků.

```

// Funkce WinMain()
    if (keys[VK_UP] && (scroll > 0)) // Šipka nahoru?
    {
        scroll -= 2; // Posune text nahoru
    }

    if (keys[VK_DOWN] && (scroll < 32*(maxtokens-9))) // Šipka dolů?
    {
        scroll += 2; // Posune text dolů
    }
}

```

Doufám, že byl pro vás tento tutoriál zajímavý. Již víte, jak získat informace o výrobci, jménu a verzi grafické karty a také, která OpenGL rozšíření podporuje. Měli byste vědět, jak použít ořezávací testy a neméně důležitou věcí je nahrávání TGA místo bitmapových obrázků a jejich konverze na textury.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 25 - Morfování objektů a jejich nahrávání z textového souboru

V této lekci se naučíte, jak nahrát souřadnice vrcholů z textového souboru a plynulou transformaci z jednoho objektu na druhý. Nezaměříme se ani tak na grafický výstup jako spíše na efekty a potřebnou matematiku okolo. Kód může být velice jednoduše modifikován k vykreslování linkami nebo polygony.

Poznamenejme, že každý objekt by měl být seskládán ze stejného počtu bodů jako všechny ostatní. Je to sice hodně omezující požadavek, ale co se dá dělat - chceme přece, aby změny vypadaly dobře. Začneme vložením hlavičkových souborů. Tentokrát nepoužíváme textury, takže se obejdeme bez glaux.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup
#include <math.h>// Hlavičkový soubor pro matematickou knihovnu

#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
```

```
HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace
```

```
bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Po deklarování všech standardních proměnných přidáme nové. Rot ukládají aktuální úhel rotace na jednotlivých souřadnicových osách, speed definují rychlost rotace. Poslední tři desetinné proměnné určují pozici ve scéně.

```
GLfloat xrot, yrot, zrot;// Rotace
GLfloat xspeed, yspeed, zspeed;// Rychlost rotace
GLfloat cx, cy, cz = -15;// Pozice
```

Aby se program zbytečně nezpomaloval při pokusech morfovat objekt sám na sebe, deklaruje key, který označuje právě zobrazený objekt. Morph v programu indikuje, jestli právě provádíme transformaci objektů nebo ne. V ustáleném stavu má hodnotu FALSE.

```
int key = 1;// Právě zobrazený objekt
bool morph = FALSE;// Probíhá právě morfování?
```

Přiřazením 200 do steps určíme, že změna jednoho objektu na druhý bude trvat 200 překreslení. Čím větší číslo zadáme, tím budou přeměny plynulejší, ale zároveň méně pomalé. No a step definuje číslo právě prováděného kroku.

```
int steps = 200;// Počet kroků změny
int step = 0;// Aktuální krok
```

Struktura VERTEX obsahuje x, y, z složky pozice jednoho bodu ve 3D prostoru.

```
typedef struct// Struktura pro bod ve 3D
{
    float x, y, z;// X, y, z složky pozice
} VERTEX;// Nazvaný VERTEX
```

Pokusíme se o vytvoření struktury objektu. co všechno budeme potřebovat? Tak určitě to bude nějaké pole pro uložení všech vrcholů. Abychom ho mohli v průběhu programu libovolně měnit, deklaruje jej jako ukazatel do dynamické paměti. Celočíslná proměnná vert specifikuje maximální možný index tohoto pole a vlastně i počet bodů, ze kterých se skládá.

```
typedef struct// Struktura objektu
{
    int verts;// Počet bodů, ze kterých se skládá
    VERTEX* points;// Ukazatel do pole vertexů
} OBJECT;// Nazvaný OBJECT
```

Pokud bychom se nedrželi zásady, že všechny objekty musí mít stejný počet vrcholů, vznikly by komplikace. Dají se

vyřešit proměnnou, která obsahuje číslo maximálního počtu souřadnic. Uvedme příklad: jeden objekt bude krychle s osmi vrcholy a druhý pyramida se čtyřmi. Do maxver tedy uložíme číslo osm. Nicméně stejně doporučuji, aby měly všechny objekty stejný počet bodů - vše bude jednodušší.

```
int maxver;// Eventuálně ukládá maximální počet bodů v jednom objektu
```

První tři instance struktury OBJECT ukládají data, která nahrajeme ze souborů. Do čtvrtého vygenerujeme náhodná čísla - body náhodně rozházené po obrazovce. Helper je objekt pro vykreslování. Obsahuje mezistavy získané kombinací objektů v určitém kroku morfingu. Poslední dvě proměnné jsou ukazatele na zdrojový a výsledný objekt, které chce uživatel zaměnit.

```
OBJECT morph1, morph2, morph3, morph4;// Koule, toroid, válec (trubka), náhodné body  
OBJECT helper, *sour, *dest;// Pomocný, zdrojový a cílový objekt
```

Ve funkci objallocate() alokujeme paměť pro strukturu objektu, na který ukazuje pointer *k předaný parametrem. Celočíslné n definuje počet vrcholů objektu.

Funkci malloc(), která vrací ukazatel na dynamicky alokovanou paměť předáme její požadovanou velikost. Získáme ji operátorem sizeof() vynásobeným počtem vertexů. Protože malloc() vrací ukazatel na void, musíme ho přetypovat.

Pozn. překl.: Program by ještě měl otestovat jestli byla opravdu alokována. Kdyby se operace nezdařila, program by přistupoval k nezabrané paměti a aplikace by se zcela jistě zhroutila. Malloc() v případě neúspěchu vrací NULL.

```
void objallocate(OBJECT *k,int n)// Alokuje dynamickou paměť pro objekt  
{  
    k->points = (VERTEX*) malloc(sizeof(VERTEX) * n);// Alokuje paměť  
  
    // Překladač:  
    // if(k->points == NULL)  
    // {  
        //MessageBox(NULL,"Chyba při alokaci paměti pro objekt", "ERROR", MB_OK |  
        MB_ICONSTOP);  
        // Ukončit program  
    // }  
}
```

Po každé alokaci dynamické paměti musí VŽDY přijít její uvolnění. Funkci opět předáváme ukazatel na objekt.

```
void objfree(OBJECT *k)// Uvolní dynamickou paměť objektu  
{  
    free(k->points);// Uvolní paměť  
}
```

Funkce readstr() je velmi podobná (úplně stejná) jako v lekci 10. Načte jeden řádek ze souboru f a uloží ho do řetězce string. Abychom mohli udržet data souboru přehledná funkce přeskakuje prázdné řádky (\n) a c-ěčkovské komentáře (řádky začínající //, respektive /').

```
void readstr(FILE *f,char *string)// Načte jeden použitelný řádek ze souboru  
{  
    do  
    {  
        fgets(string, 255, f);// Načti řádek  
    } while ((string[0] == '/') || (string[0] == '\n'));// Pokud není použitelný, načti  
    další  
    return;  
}
```

Napišeme funkci pro loading objektu z textového souboru. Name specifikuje diskovou cestu k souboru a k je ukazatel na objekt, do kterého uložíme výsledek.

```
void objload(char *name,OBJECT *k)// Nahraje objekt ze souboru  
{
```

Začneme deklarací lokálních proměnných funkce. Do ver načteme počet vertexů, který určuje první řádka v souboru (více dále). Dá se říct, že rx, ry, rz jsou pouze pro zpřehlednění zdrojového kódu programu. Ze souboru do nich načteme jednotlivé složky bodu. Ukazatel filein ukazuje na soubor (po otevření). Oneline je znakový buffer. Vždy do něj načteme jednu řádku, analyzujeme ji a získáme informace, které potřebujeme.

```
int ver;// Počet bodů  
float rx, ry, rz;// X, y, z pozice  
FILE* filein;// Handle souboru  
char oneline[255];// Znakový buffer
```

Pomocí funkce fopen() otevřeme soubor pro čtení. Pozn. překl.: Stejně jako u alokace paměti i zde chybí ošetření chyb.

```
filein = fopen(name, "rt");// Otevře soubor
// Překladač:
// if(filein == NULL)
// {
//     // MessageBox(NULL,"Chyba při otevření souboru s daty", "ERROR", MB_OK |
//     MB_ICONSTOP);
//     // Ukončit program
// }
```

Do znakového bufferu načteme první řádku. Měla by být ve tvaru: Vertices: x\n. Z řetězce tedy potřebujeme vydolovat číslo x, které udává počet vertexů definovaných v souboru. Tento počet uložíme do vnitřní proměnné struktury a potom alokujeme tolik paměti, aby se do ní všechny koordináty souřadnic vešly.

```
readstr(filein, oneline);// Načte první řádku ze souboru
sscanf(oneline, "Vertices: %d\n", &ver);// Počet vertexů

k->verts = ver;// Nastaví položku struktury na správnou hodnotu

objallocate(k, ver);// Alokace paměti pro objekt
```

Už tedy víme z kolikati bodů je objekt vytvořen a máme alokovánou potřebnou paměť. Nyní ještě musíme načíst jednotlivé hodnoty. Provedeme to cyklem for s řídicí proměnnou i, která se každým průchodem inkrementuje. Postupně načteme všechny řádky do bufferu, ze kterého přes funkci sscanf() dostaneme číselné hodnoty složek vertexu pro všechny tři souřadnicové osy. Pomocné proměnné zkopírujeme do proměnných struktury. Po analýze celého souboru ho zavřeme.

Ještě musím upozornit, že je důležité, aby soubor obsahoval stejný počet bodů jako je definováno na začátku. Pokud by jich bylo více, tolik by to nevadilo - poslední by se prostě nenačetly. V žádném případě jich ale NESMÍ být méně! S největší pravděpodobností by to zhroutilo program. Vše, na co se pokouším upozornit by se dalo shrnout do věty: Jestliže soubor začíná "Vertices: 10", musí v něm být specifikováno 10 souřadnic (30 čísel - x, y, z).

```
for (int i = 0; i < ver; i++)// Postupně načítá body
{
    readstr(filein, oneline);// Načte řádek ze souboru

    sscanf(oneline, "%f %f %f", &rx, &ry, &rz);// Najde a uloží tři čísla

    k->points[i].x = rx;// Nastaví vnitřní proměnnou struktury
    k->points[i].y = ry;// Nastaví vnitřní proměnnou struktury
    k->points[i].z = rz;// Nastaví vnitřní proměnnou struktury
}

fclose(filein);// Zavře soubor
```

Otestujeme, zda není proměnná ver (počet bodů aktuálního objektu) větší než maxver (v současnosti maximální známý počet vertexů v jednom objektu). Pokud ano přiřadíme ver do maxver.

```
if(ver > maxver)// Aktualizuje maximální počet vertexů
    maxver = ver;
}
```

Na řadu přichází trochu méně pochopitelná funkce - zvlášť pro ty, kteří nemají v lásce matematiku. Bohužel morfining na ní staví. Co tedy dělá? Spočítá o kolik máme posunout bod specifikovaný parametrem i. Na začátku deklarujeme pomocný vertex, podle vzorce spočítáme jeho jednotlivé x, y, z složky a v závěru ho vrátíme volající funkci.

Použitá matematika pracuje asi takto: od souřadnice i-tého bodu zdrojového objektu odečteme souřadnici bodu, do kterého morfuje. Rozdíl vydělíme zamýšleným počtem kroků a konečný výsledek uložíme do a.

Řekněme, že x-ové souřadnice zdrojového objektu (sour) je rovna čtyřiceti a cílového objektu (dest) dvaceti. U deklarace globálních proměnných jsme steps přiřadili 200. Výpočtem $a.x = (40-20)/200 = 20/200 = 0,1$ zjistíme, že při přesunu ze 40 na 20 s krokem 200 potřebujeme každé překreslení pohnout na ose x bodem o desetinu jednotky. Nebo jinak: násobíme-li $200 \cdot 0,1$ dostaneme rozdíl pozic 20, což je také pravda ($40-20=20$). Mělo by to fungovat.

```
VERTEX calculate(int i)// Spočítá o kolik pohnout bodem při morfiningu
{
    VERTEX a;// Pomocný bod

    a.x = (sour->points[i].x - dest->points[i].x) / steps;// Spočítá posun
    a.y = (sour->points[i].y - dest->points[i].y) / steps;// Spočítá posun
    a.z = (sour->points[i].z - dest->points[i].z) / steps;// Spočítá posun
}
```

```

    return a; // Vrátí výsledek
}

```

Začátek inicializační funkce není žádnou novinkou, ale dále v kódu najdete změny.

```

int InitGL(GLvoid) // Všechno nastavení OpenGL
{
    glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Typ blendingu
    // glEnable(GL_BLEND); // Zapne blending (překl.: autor asi zapomněl)

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Černé pozadí

    glClearDepth(1.0); // Nastavení hloubkového bufferu
    glDepthFunc(GL_LESS); // Typ hloubkového testování
    glEnable(GL_DEPTH_TEST); // Povolení testování hloubky

    glShadeModel(GL_SMOOTH); // Jemné stínování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce
}

```

Protože ještě neznáme maximální počet bodů v jednom objektu, přiřadíme do maxver nulu. Poté pomocí funkce objload () načteme z disku data jednotlivých objektů (koule, toroid, válec). V prvním parametru předáváme cestu se jménem souboru, ve druhém adresu objektu, do kterého se mají data uložit.

```

    maxver = 0; // Nulování maximálního počtu bodů

    objload("data/sphere.txt", &morph1); // Načte kouli
    objload("data/torus.txt", &morph2); // Načte toroid
    objload("data/tube.txt", &morph3); // Načte válec
}

```

Čtvrtý objekt nenačítáme ze souboru. Budou jím po scéně rozházené body (přesně 486 bodů). Nejdříve musíme alokovat paměť pro jednotlivé vertexy a potom stačí v cyklu vygenerovat náhodné souřadnice. Budou v rozmezí od -7 do +7.

```

    objallocate(& morph4, 486); // Alokace paměti pro 486 bodů

    for(int i=0; i < 486; i++) // Cyklus generuje náhodné souřadnice
    {
        morph4.points[i].x = ((float) rand() % 14000) / 1000 - 7; // Náhodná hodnota
        morph4.points[i].y = ((float) rand() % 14000) / 1000 - 7; // Náhodná hodnota
        morph4.points[i].z = ((float) rand() % 14000) / 1000 - 7; // Náhodná hodnota
    }
}

```

Ze souborů jsme loadovali všechny objekty do struktur. Jejich data už nebudeme upravovat. Od teď jsou jen pro čtení. Potřebujeme tedy ještě jeden objekt, helper, který bude při morfingu ukládat jednotlivé mezistavy. Protože na začátku zobrazujeme morph1 (koule) načteme i do pomocného tento objekt.

```

    objload("data/sphere.txt", &helper); // Načtení koule do pomocného objektu
}

```

Nastavíme ještě pointery pro zdrojový a cílový objekt, tak aby ukazovali na adresu morph1.

```

    sour = dest = &morph1; // Inicializace ukazatelů na objekty

    return TRUE; // Ukončí funkci
}

```

Vykreslování začneme klasicky smazáním obrazovky a hloubkového bufferu, resetem matice, posunem a rotacemi. Místo abychom všechny pohyby prováděli na konci funkce, tentokrát je umístíme na začátek. Poté deklaruji pomocné proměnné. Do tx, ty, tz spočítáme souřadnice, které pak předáme funkci glVertex3f() kvůli nakreslení bodu. Q je pomocný bod pro výpočet.

```

void DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(cx, cy, cz); // Přesun na pozici
    glRotatef(xrot, 1, 0, 0); // Rotace na ose x
    glRotatef(yrot, 0, 1, 0); // Rotace na ose y
    glRotatef(zrot, 0, 0, 1); // Rotace na ose z

    xrot += xspeed; // Zvětší úhly rotace
    yrot += yspeed;
}

```

```

zrot += zspeed;

GLfloat tx, ty, tz; // Pomocné souřadnice
VERTEX q; // Pomocný bod pro výpočty

```

Přes glBegin(GL_POINTS) oznámíme OpenGL, že v blízké době budeme vykreslovat body. V cyklu for procházíme vertexy. Řídící proměnnou i bychom také mohli porovnávat s maxver, ale protože mají všechny objekty stejný počet souřadnic, můžeme s klidem použít počet vertexů prvního objektu - morph1.verts.

```

glBegin(GL_POINTS); // Začátek kreslení bodů
for(int i = 0; i < morph1.verts; i++) // Cyklus prochází vertexy
{

```

V případě morfingu spočítáme o kolik se má vykreslovaný bod posunout oproti pozici při minulém vykreslení. Takto vypočítané hodnoty odečteme od souřadnic pomocného objektu, do kterého každé překreslení ukládáme aktuální mezistav morfingu. Pokud se zrovna objekty mezi sebou netransformují odečítáme nulu, takže se souřadnice defakto nemění.

```

if(morph) // Pokud zrovna morfujeme
    q = calculate(i); // Spočítáme hodnotu posunutí
else // Jinak
    q.x = q.y = q.z = 0; // Budeme odečítat nulu, ale tím neposouváme

helper.points[i].x -= q.x; // Posunutí na ose x
helper.points[i].y -= q.y; // Posunutí na ose y
helper.points[i].z -= q.z; // Posunutí na ose z

```

Abychom si zpřehlednili program a také kvůli maličkému efektu, zkopírujeme právě získaná čísla do pomocných proměnných.

```

tx = helper.points[i].x; // Zpřehlednění + efekt
ty = helper.points[i].y; // Zpřehlednění + efekt
tz = helper.points[i].z; // Zpřehlednění + efekt

```

Všechno máme spočítáno, takže přejdeme k vykreslení. Nastavíme barvu na zelenomodrou a nakreslíme bod. Potom zvolíme trochu tmavší modrou barvu. Odečteme dvojnásobek souřadnic q od t a získáme umístění bodu při následujícím volání této funkce (ob jedno). Na této pozici znovu vykreslíme bod. Do třetice všeho dobrého znovu ztmavíme barvu a opět spočítáme další pozici, na které se vyskytne po čtyřech průchodech touto funkcí a opět ho vykreslíme.

Proč jsme krásně přehledný kód vlastně komplikovali? I když si to asi neuvědomujete, vytvořili jsme jednoduchý částicový systém. S použitím blendingu vytvoří perfektní efekt, který se ale bohužel projeví pouze při transformaci objektů z jednoho na druhý. Pokud zrovna nemorfujeme, v q souřadnicích jsou uloženy nuly, takže druhý a třetí bod kreslíme na stejné místo jako první.

```

glColor3f(0, 1, 1); // Zelenomodrá barva
glVertex3f(tx, ty, tz); // Vykreslí první bod

glColor3f(0, 0.5f, 1); // Modřejší zelenomodrá barva
tx -= 2*q.x; // Spočítání nových pozic
ty -= 2*q.y;
tz -= 2*q.z;
glVertex3f(tx, ty, tz); // Vykreslí druhý bod v nové pozici

glColor3f(0, 0, 1); // Modrá barva
tx -= 2*q.x; // Spočítání nových pozic
ty -= 2*q.y;
tz -= 2*q.z;
glVertex3f(tx, ty, tz); // Vykreslí třetí bod v nové pozici

```

Ukončíme tělo cyklu a glEnd() oznámí, že dále už nebudeme nic vykreslovat.

```

}
glEnd(); // Ukončí kreslení

```

Jako poslední v této funkci zkontrolujeme jestli transformujeme objekty. Pokud ano a zároveň musí být aktuální krok morfingu menší než celkový počet kroků, inkrementujeme aktuální krok. Po dokončení morfingu ho vypneme. Protože jsme už došli k cílovému objektu, uděláme z něj zdrojový. Krok reinitializujeme na nulu.

```

if(morph && step <= steps) // Morfujeme a krok je menší než maximum
{
    step++; // Příkladně pokračuj následujícím krokem
}

```



```

else// Nemorfujeme nebo byl právě ukončen
{
    morph = FALSE;// Konec morfinu
    sour = dest;// Cílový objekt je nyní zdrojový
    step = 0;// První (nulový) krok morfinu
}
}

```

KillGLWindow upravíme jenom málo. Uvolníme pouze dynamicky alokovanou paměť.

```

GLvoid KillGLWindow(GLvoid)// Zavírání okna
{
    objfree(&morph1);// Uvolní alokovanou paměť
    objfree(&morph2);// Uvolní alokovanou paměť
    objfree(&morph3);// Uvolní alokovanou paměť
    objfree(&morph4);// Uvolní alokovanou paměť
    objfree(&helper);// Uvolní alokovanou paměť

    // Zbytek nezměněn
}

```

Ve funkci WinMain() upravíme kód testující stisk kláves. Následujícími šesti testy regulujeme rychlost rotace objektu.

```

// Funkce WinMain()
    if(keys[VK_PRIOR])// PageUp?
        zspeed += 0.01f;

    if(keys[VK_NEXT])// PageDown?
        zspeed -= 0.01f;

    if(keys[VK_DOWN])// Šipka dolů?
        xspeed += 0.01f;

    if(keys[VK_UP])// Šipka nahoru?
        xspeed -= 0.01f;

    if(keys[VK_RIGHT])// Šipka doprava?
        yspeed += 0.01f;

    if(keys[VK_LEFT])// Šipka doleva?
        yspeed -= 0.01f;

```

Dalších šest kláves pohybuje objektem po scéně.

```

    if (keys['Q'])// Q?
        cz -= 0.01f;// Dále

    if (keys['Z'])// Z?
        cz += 0.01f;// Blíže

    if (keys['W'])// W?
        cy += 0.01f;// Nahoru

    if (keys['S'])// S?
        cy -= 0.01f;// Dolu

    if (keys['D'])// D?
        cx += 0.01f;// Doprava

    if (keys['A'])// A?
        cx -= 0.01f;// Doleva

```

Ted' ošetříme stisk kláves 1-4. Aby se kód provedl, nesmí být při stisku jedničky key roven jedné (nejde morfovat z prvního objektu na první) a také nesmíme právě morfovat (nevypadalo by to dobře). V takovém případě nastavíme pro příští průchod tímto místem key na jedna a morph na TRUE. Cílovým objektem bude objekt jedna. Klávesy 2, 3, 4 jsou analogické.

```

    if (keys['1'] && (key!=1) && !morph)// Klávesa 1?
    {
        key = 1;// Proti dvojnásobnému stisku
        morph = TRUE;// Začne morfovací proces
        dest = &morph1;// Nastaví cílový objekt
    }

```

```

if (keys['2'] && (key!=2) && !morph)// Klávesa 2?
{
    key = 2;// Proti dvojnásobnému stisku
    morph = TRUE;// Začne morfovací proces
    dest = &morph2;// Nastaví cílový objekt
}
if (keys['3'] && (key!=3) && !morph)// Klávesa 3?
{
    key = 3;// Proti dvojnásobnému stisku
    morph = TRUE;// Začne morfovací proces
    dest = &morph3;// Nastaví cílový objekt
}
if (keys['4'] && (key!=4) && !morph)// Klávesa 4?
{
    key = 4;// Proti dvojnásobnému stisku
    morph = TRUE;// Začne morfovací proces
    dest = &morph4;// Nastaví cílový objekt
}

```

Doufám, že jste si tento tutoriál užili. Ačkoli výstup není až tak fantastický jako v některých jiných, naučili jste se spoustu věcí. Hraním si s kódem lze docílit skvělých efektů - třeba po scéně náhodně rozházené body mění se ve slova. Zkuste použít polygony nebo linky namísto bodů, výsledek bude ještě lepší.

Před tím, než vznikla tato lekce bylo vytvořeno demo "Morph", které demonstruje mnohem pokročilejší verzi probíraného efektu. Lze ho najít na adrese <http://homepage.ntlworld.com/fj.williams/PgSoftware.html> .

napsal: Piotr Cieslak
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 26 - Odrazy a jejich ořezávání za použití stencil bufferu

Tutoriál demonstuje extrémně realistické odrazy za použití stencil bufferu a jejich ořezávání, aby "nevystoupily" ze zrcadla. Je mnohem více pokrokový než předchozí lekce, takže před začátkem čtení doporučuji menší opakování. Odrazy objektů nebudou vidět nad zrcadlem nebo na druhé straně zdi a budou mít barevný nádech zrcadla - skutečné odrazy.

Důležité: Protože grafické karty Voodoo 1, 2 a některé jiné nepodporují stencil buffer, nebude na nich tento tutoriál fungovat. Pokud si nejste jistí, že vaše karta stencil buffer podporuje, stáhněte si zdrojový kód a zkuste jej spustit. Kromě toho budete také potřebovat procesor a grafickou kartu se slušným výkonem. Na mé GeForce 1 občas vidím malé zpomalení. Demo běží nejlépe v 32 bitových barvách.

První část kódu je celkem standardní.

```
#include <windows.h>// Hlavičkový soubor pro Windows
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup

#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

HDC hDC = NULL;// Privátní GDI Device Context
HGLRC hRC = NULL;// Trvalý Rendering Context
HWND hWnd = NULL;// Obsahuje Handle našeho okna
HINSTANCE hInstance;// Obsahuje instanci aplikace

bool keys[256];// Pole pro ukládání vstupu z klávesnice
bool active = TRUE;// Ponese informaci o tom, zda je okno aktivní
bool fullscreen = TRUE;// Ponese informaci o tom, zda je program ve fullscreenu
```

Nastavíme pole pro definici osvětlení. Okolní světlo bude 70% bílé. Difúzní světlo nastavuje rozptýl osvětlení (množství světla rovnoměrně odrážené na plochách objektů). V tomto případě odrážíme plnou intenzitou. Poslední je pozice. Pokud bychom ho mohli spatřit, plulo by v pravém horním rohu monitoru.

```
// Parametry světla
static GLfloat LightAmb[] = {0.7f, 0.7f, 0.7f, 1.0f};// Okolní
static GLfloat LightDif[] = {1.0f, 1.0f, 1.0f, 1.0f};// Rozptýlené
static GLfloat LightPos[] = {4.0f, 4.0f, 6.0f, 1.0f};// Pozice
```

Ukazatel `q` je pro quadratic koule (plážový míč). `xrot` a `yrot` ukládají hodnoty natočení míče, `xrotspeed` a `yrotspeed` definují rychlost rotace. `zoom` používáme pro přibližování a oddalování scény a `height` je výška balónu nad podlahou. Pole `texture[]` už standardně ukládá textury.

```
GLUquadricObj *q;// Quadratic pro kreslení koule (míče)

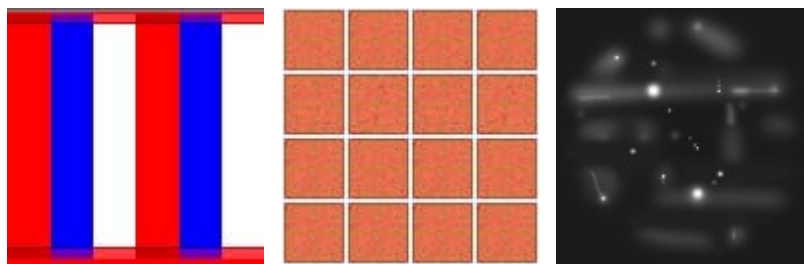
GLfloat xrot = 0.0f;// X rotace
GLfloat yrot = 0.0f;// Y rotace

GLfloat xrotspeed = 0.0f;// Rychlost x rotace
GLfloat yrotspeed = 0.0f;// Rychlost y rotace

GLfloat zoom = -7.0f;// Hloubka v obrazovce
GLfloat height = 2.0f;// Výška míče nad scénou

GLuint texture[3];// 3 textury
```

Vytváření lineárně filtrovaných textur z bitmap je standardní, v předchozích lekcích jsme jej používali velice často, takže ho sem nebudu opisovat. Na obrázcích vidíte texturu míče, podlahy a světla odráženého od míče.



Inicializace OpenGL.

```
int InitGL(GLvoid)// Nastavení OpenGL
{
    if (!LoadGLTextures())// Loading textur
    {
        return FALSE;// Ukončí program
    }

    glShadeModel(GL_SMOOTH);// Vyhlazené stínování
    glClearColor(0.2f, 0.5f, 1.0f, 1.0f);// Světle modré pozadí
    glClearDepth(1.0f);// Nastavení hloubkového bufferu
```

Příkaz `glClearStencil()` definuje chování funkce `glClear()` při mazání stencil bufferu. V tomto případě ho budeme vyplňovat nulami.

```
glClearStencil(0);// Nastavení mazání stencil bufferu

glEnable(GL_DEPTH_TEST);// Povolí testování hloubky
glDepthFunc(GL_LEQUAL);// Typ testování hloubky
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);// Perspektivní korekce
glEnable(GL_TEXTURE_2D);// Mapování textur
```

Nastavíme světla. Pro okolní použijeme hodnoty z pole `LightAmb[]`, rozptylové světlo definujeme pomocí `LightDif[]` a pozici z `LightPos[]`. Nakonec povolíme světla. Pokud bychom dále v kódu chtěli vypnout všechna světla, použili bychom `glDisable(GL_LIGHTING)`, ale při vypínání jenom jednoho postačí pouze `glDisable(GL_LIGHT(0až7))`. `GL_LIGHTING` v parametru zakazuje globálně všechna světla.

```
glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmb);// Okolní
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDif);// Rozptylové
glLightfv(GL_LIGHT0, GL_POSITION, LightPos);// Pozice

glEnable(GL_LIGHT0);// Povolí světlo 0
glEnable(GL_LIGHTING);// Povolí světla
```

Dále vytvoříme a nastavíme objekt `quadraticu`. Vygenerujeme mu normály pro světlo a texturové koordináty, jinak by měl ploché stínování a nešly by na něj namapovat textury.

```
q = gluNewQuadric();// Nový quadratic

gluQuadricNormals(q, GL_SMOOTH);// Normály pro světlo
gluQuadricTexture(q, GL_TRUE);// Texturové koordináty
```

Nastavíme mapování textur na vykreslované objekty a to tak, aby při natáčení míče byla viditelná stále stejná část textury. Zatím ho nezapínáme.

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);// Automatické mapování textur
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);// Automatické mapování textur

return TRUE;// Inicializace v pořádku
}
```

Následující funkci budeme volat pro vykreslení plážového míče. Bude jím `quadraticová` koule s nalepenou texturou. Nastavíme barvu na bílou, aby se textura nezabarvovala, poté zvolíme texturu a vykreslíme kouli o poloměru 0.35 jednotek, s 32 rovnoběžkami a 16 poledníky.

```
void DrawObject()// Vykreslí plážový míč
{
    glColor3f(1.0f, 1.0f, 1.0f);// Bílá barva
    glBindTexture(GL_TEXTURE_2D, texture[1]);// Zvolí texturu míče
    gluSphere(q, 0.35f, 32, 16);// Nakreslí kouli
```

Po vykreslení první koule vybereme texturu světla, nastavíme opět bílou barvu, ale tentokrát s 40% alfa. Povolíme blending, nastavíme jeho funkci založenou na zdrojové alfa hodnotě, zapneme kulové mapování textur a nakreslíme stejnou kouli jako před chvílí. Výsledkem je simulované odrazení světla od míče, ale vlastně se jedná jen o světlé body namapované na plážový míč. Protože je povoleno kulové mapování, textura je vždy natočena k pozorovateli stejnou částí bez ohledu na natočení míče. Je také zapnutý blending takže nová textura nepřebije starou (jednoduchá forma multitexturingu).

```
glBindTexture(GL_TEXTURE_2D, texture[2]); // Zvolí texturu světla
glColor4f(1.0f, 1.0f, 1.0f, 0.4f); // Bílá barva s 40% alfa

glEnable(GL_BLEND); // Zapne blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE); // Mód blendingu

glEnable(GL_TEXTURE_GEN_S); // Zapne kulové mapování
glEnable(GL_TEXTURE_GEN_T); // Zapne kulové mapování

gluSphere(q, 0.35f, 32, 16); // Stejná koule jako před chvílí
```

Vypneme kulové mapování a blending.

```
glDisable(GL_TEXTURE_GEN_S); // Vypne kulové mapování
glDisable(GL_TEXTURE_GEN_T); // Vypne kulové mapování

glDisable(GL_BLEND); // Vepne blending
}
```

Následující funkce kreslí podlahu, nad kterou se míč vznáší. Vybereme texturu podlahy a na ose z vykreslíme čtverec s jednoduchou texturou.

```
void DrawFloor() // Vykreslí podlahu
{
    glBindTexture(GL_TEXTURE_2D, texture[0]); // Zvolí texturu podlahy

    glBegin(GL_QUADS); // Kreslení obdélníků
        glNormal3f(0.0, 1.0, 0.0); // Normálová vektor míří vzhůru

        glTexCoord2f(0.0f, 1.0f); // Levý dolní bod textury
        glVertex3f(-2.0, 0.0, 2.0); // Levý dolní bod podlahy

        glTexCoord2f(0.0f, 0.0f); // Levý horní bod textury
        glVertex3f(-2.0, 0.0, -2.0); // Levý horní bod podlahy

        glTexCoord2f(1.0f, 0.0f); // Pravý horní bod textury
        glVertex3f( 2.0, 0.0, -2.0); // Pravý horní bod podlahy

        glTexCoord2f(1.0f, 1.0f); // Pravý dolní bod textury
        glVertex3f( 2.0, 0.0, 2.0); // Pravý dolní bod podlahy
    glEnd(); // Konec kreslení
}
```

Na tomto místě zkombinujeme všechny objekty a obrázky tak, abychom vytvořili výslednou scénu. Začneme mazáním obrazovky (`GL_COLOR_BUFFER_BIT`) na výchozí modrou barvu, hloubkového bufferu (`GL_DEPTH_BUFFER_BIT`) a stencil bufferu (`GL_STENCIL_BUFFER_BIT`). Při čištění stencil bufferu ho vyplňujeme nulami.

```
int DrawGLScene(GLvoid) // Vykreslí výslednou scénu
{
    // Smaže obrazovku, hloubkový buffer a stencil buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

Nadefinujeme rovnici ořezávací plochy (clipping plane equation). Bude použita při vykreslení odraženého míče. Hodnota na ose y je záporná, to znamená, že uvidíme pixely jen pokud jsou kresleny pod podlahou nebo na záporné části osy y. Při použití této rovnice se nezobrazí nic, co vykreslíme nad podlahou (odraz nemůže vystoupit ze zrcadla). Více později.

```
// Rovnice ořezávací plochy
double eqr[] = { 0.0f, -1.0f, 0.0f, 0.0f }; // Použito pro odražený objekt
```

Všemu, co bylo doposud probráno v této lekci byste měli rozumět. Teď přijde něco "maličko" horšího. Potřebujeme nakreslit odraz míče a to tak, aby se na obrazovce zobrazoval jenom na těch pixelech, kde je podlaha. K tomu využijeme stencil buffer. Pomocí funkce `glClear()` jsme ho vyplnili samými nulami. Různými nastaveními, které si vysvětlíme dále, docílíme toho, že se podlaha sice nezobrazí na obrazovce, ale na místech, kde se měla vykreslit se stencil buffer nastaví do jedničky. Pro pochopení si představte, že je to obrazovka v paměti, jejíž pixely jsou rovny jedničce, pokud se na nich objekt vykresluje a nule (nezměněný) pokud ne. Na místa, kde je stencil buffer v jedničce vykreslíme plochý odraz míče, ale ne do stencil bufferu - viditelně na obrazovku. Odras vlastně můžeme vykreslit i

kdekoli jinde, ale pouze tady bude vidět. Nakonec klasickým způsobem vykreslíme všechno ostatní. To je asi všechno, co byste měli o stencil bufferu prozatím vědět.

Nyní už konkrétně ke kódu. Resetujeme matici modelview a potom přesuneme scénu o šest jednotek dolů a o zoom do hloubky. Nejlepší vysvětlení pro translaci dolů bude na příkladě. Vezměte si list papíru a umístěte jej rovnoběžně se zemí do úrovně očí. Neuvidíte nic víc než tenkou linku. Posunete-li jím o maličko dolů, spatříte celou plochu, protože se na něj budete dívat více ze shora namísto přímo na okraj. Rozšířil se zorný úhel.

```
glLoadIdentity();// Reset matice
glTranslatef(0.0f, -0.6f, zoom);// Zoom a vyvýšení kamery nad podlahu
```

Novým příkazem definujeme barevnou masku pro vykreslované barvy. Funkci se předávají čtyři parametry reprezentující červenou, zelenou, modrou a alfu. Pokud například červenou složku nastavíme na jedna (GL_TRUE) a všechny ostatní na nulu (GL_FALSE), tak se bude moci zobrazit pouze červená barva. V opačném případě (0,1,1,1) se budou zobrazovat všechny barvy mimo červenou. Asi tušíte, že jsou barvy implicitně nastaveny tak, aby se všechny zobrazovaly. No, a protože v tuto chvíli nechceme nic zobrazovat zakážeme všechny barvy.



```
glColorMask(0,0,0,0);// Nastaví masku barev, aby se nic nezobrazilo
```

Začínáme pracovat se stencil bufferem. Napřed potřebujeme získat obraz podlahy vyjádřený jedničkami (viz. výše). Začneme zapnutím stencilového testování (stencil testing). Jakmile je povoleno jsme schopni modifikovat stencil buffer.

```
glEnable(GL_STENCIL_TEST);// Zapne stencil buffer pro paměťový obraz podlahy
```

Následující příkaz je možná těžko pochopitelný, ale určitě se velice těžko vysvětluje. Funkce glStencilFunc (GL_ALWAYS,1,1) oznamuje OpenGL, jaký typ testu chceme použít na každý pixel při jeho vykreslování. GL_ALWAYS zaručí, že test proběhne vždy. Druhý parametr je referenční hodnotou a třetí parametr je maska. U každého pixelu se hodnota masky ANDuje s referenční hodnotou a výsledek se uloží do stencil bufferu. V našem případě se do něj umístí pokaždé jednička (reference & maska = 1 & 1 = 1). Nyní víme, že na souřadnicích pixelu na obrazovce, kde by se vykreslil objekt, bude ve stencil bufferu jednička.

Pozn.: Stencilové testy jsou vykonávány na pixelech pokaždé, když se objekt vykresluje na scénu. Referenční hodnota ANDovaná s hodnotou masky se testuje proti aktuální hodnotě ve stencil bufferu ANDované s hodnotou masky.

```
glStencilFunc(GL_ALWAYS, 1, 1);// Pokaždé proběhne, reference, maska
```

glStencilOp() zpracuje tři rozdílné požadavky založené na stencilových funkcích, které jsme se rozhodli použít. První parametr říká OpenGL, co má udělat pokud test neuspěje. Protože je nastaven na GL_KEEP nechá hodnotu stencil bufferu tak, jak právě je. Nicméně test uspěje vždy, protože máme funkci nastavenou na GL_ALWAYS. Druhý parametr určuje co dělat, pokud stencil test proběhne, ale hloubkový test bude neúspěšný. Tato situace by nastala například, když by se objekt vykreslil za jiným objektem a hloubkový test by nepovolil jeho vykreslení. Opět může být ignorován, protože hned následujícím příkazem hloubkové testy vypínáme. Třetí parametr je pro nás důležitý. Definuje, co se má vykonat, pokud test uspěje (uspěje vždycky). V našem případě OpenGL nahradí nulu ve stencil bufferu na jedničku (referenční hodnota ANDovaná s maskou = 1).

```
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);// Vykreslením nastavíme konkrétní bit ve stencil bufferu na 1
```

Po nastavení stencilových testů vypneme hloubkové testy a zavoláme funkci pro vykreslení podlahy.

```
glDisable(GL_DEPTH_TEST);// Vypne testování hloubky
DrawFloor();// Vykreslí podlahu (do stencil bufferu ne na scénu)
```

Takže teď máme ve stencil bufferu neviditelnou masku podlahy. Tak dlouho, jak bude stencilové testování zapnuté, budeme moci zobrazovat pixely pouze tam, kde je stencil buffer v jedničce (tam kde byla vykreslena podlahy). Zapneme hloubkové testování a nastavíme masku barev zpět do jedniček. To znamená, že se od teď vše vykreslované opravdu zobrazí.

```
glEnable(GL_DEPTH_TEST);// Zapne testování hloubky
glColorMask(1, 1, 1, 1);// Povolí zobrazování barev
```

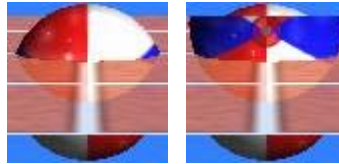
Namísto užití GL_ALWAYS pro stencilovou funkci, použijeme GL_EQUAL. Reference i maska zůstávají v jedničce. Pro stencilové operace nastavíme všechny parametry na GL_KEEP. Vykreslované pixely se zobrazí na obrazovku POUZE tehdy, když je na jejich souřadnicích hodnota stencilu v jedničce (reference ANDovaná s maskou (1), které jsou rovny (GL_EQUAL) hodnotě stencil bufferu ANDované s maskou (také 1)). GL_KEEP zajistí, že se hodnoty ve stencil bufferu nebudou modifikovat.

```

glStencilFunc(GL_EQUAL, 1, 1); // Zobrazí se pouze pixely na jedničkách ve stencil
bufferu (podlaha)
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); // Neměnit obsah stencil bufferu

```

Zapneme ořezávací plochu zrcadla, která je definována rovnicí uloženou v poli `eqr[]`. Umožňuje, aby byl odraz objektu vykreslen pouze směrem dolů od podlahy (v podlaze). Touto cestou nebude moci odraz míče vystoupit do "reálného světa". Pokud nechápete, co je tímto míněno zakomentárujte v kódu řádek `glEnable(GL_CLIP_PLANE0)`, zkompilujte program a zkuste projít reálným míčem skrz podlahu. Pokud clipping nebude zapnutý uvidíte, jak při vstupu míče do podlahy jeho odraz vystoupí nahoru nad podlahu. Vše vidíte na obrázku. Mimochodem, všimněte si, že vystoupivší obraz je pořád vidět jen tam, kde je ve stencil bufferu obraz podlahy.



Po zapnutí ořezávací plochy 0 (obvyčejně jich může být 0 až 5) jí předáme parametry rovnice uložené v `eqr[]`.

```

glEnable(GL_CLIP_PLANE0); // Zapne ořezávací testy pro odraz
glClipPlane(GL_CLIP_PLANE0, eqr); // Rovnice ořezávací roviny

```

Zálohujeme aktuální stav matice, aby jí změny trvale neovlivnily. Zadáním mínus jedničky do `glScalef()` obrátíme směr osy `y`. Do této chvíle procházela zezdola nahoru, nyní naopak. Stejný efekt by měla rotace o 180° . Vše je teď invertované jako v zrcadle. Pokud něco vykreslíme nahoře, zobrazí se to dole (zrcadlo je vodorovně ne svisle), rotujeme-li po směru, objekt se otočí proti směru hodinových ručiček a podobně. Tento stav se může zrušit buď opětovným voláním `glScalef()`, které provede opětovnou inverzi nebo POPnutím matice.

```

glPushMatrix(); // Zálaha matice
glScalef(1.0f, -1.0f, 1.0f); // Zrcadlení směru osy y

```

Nadefinujeme pozici světla podle pole `LightPos[]`. Na reálný míč svítí z pravé horní strany, ale protože se i poloha světla zrcadlí, tak na odraz bude zářit zezdola.

```

glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Umístění světla

```

Přesuneme se na ose `y` nahoru nebo dolů v závislosti na proměnné `height`. Opět je translace zrcadlena, takže pokud se přesuneme o pět jednotek nad podlahu budeme vlastně o pět jednotek pod podlahou. Stejným způsobem pracují i rotace. Nakonec nakreslíme objekt plážového míče a POPneme matici. Tím zrušíme všechny změny od volání `glPushMatrix()`.

```

glTranslatef(0.0f, height, 0.0f); // Umístění míče
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Rotace na ose x
glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Rotace na ose y

DrawObject(); // Vykreslí míč (odraz)

glPopMatrix(); // Obnoví matici

```

Vypneme ořezávací testy, takže se budou zobrazovat i objekty nad podlahou. Také vypneme stencil testy, abychom mohli vykreslovat i jinam než na pixely, které byly modifikovány podlahou.

```

glDisable(GL_CLIP_PLANE0); // Vypne ořezávací rovinu
glDisable(GL_STENCIL_TEST); // Už nebudeme potřebovat stencil testy

```

Připravíme program na vykreslení podlahy. Opět umístíme světlo, ale tak, aby už jeho pozice nebyla zrcadlena. Osa `y` je sice už v pořádku, ale světlo je stále vpravo dole.

```

glLightfv(GL_LIGHT0, GL_POSITION, LightPos); // Umístění světla

```

Zapneme blending, vypneme světla (globálně) a nastavíme 80% průhlednost bez změny barev textur (bílá nepřidává barevný nádech). Mód blendingu je nastaven pomocí `glBlendFunc()`. Poté vykreslíme částečně průhlednou podlahu. Asi nechápete, proč jsme napřed kreslili odraz a až poté zrcadlo. Je to proto, že chceme, aby byl odraz míče smíchan s barvami podlahy. Pokud se díváte do modrého zrcadla, tak také očekáváte trochu namodralý odraz. Vykreslení míče napřed způsobí zabarvení podlahou. Efekt je více reálný.

```

glEnable(GL_BLEND); // Zapne blending, jinak by se odraz míče nezobrazil
glDisable(GL_LIGHTING); // Kvůli blendingu vypneme světla

glColor4f(1.0f, 1.0f, 1.0f, 0.8f); // Bílá barva s 80% průhledností
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Funkce na bázi alfy zdroje a
jedna minus alfy cíle

```

```
DrawFloor();// Vykreslí podlahu
```

A konečně vykreslíme reálný míč. Napřed ale zapneme světla (pozice už je nastavená). Kdybychom nevyprnuli blending, míč by při průchodu podlahou vypadal jako odraz. To nechceme.

```
glEnable(GL_LIGHTING);// Zapne světla  
glDisable(GL_BLEND);// Vypne blending
```

Tento míč už narozdíl od jeho odrazu neořezáváme. Kdybychom používali clipping, nezobrazil by se pod podlahou. Docílili bychom toho definováním hodnoty +1.0f na ose y u rovnice ořezávací roviny. Pro toto demo není žádný důvod, abychom míč nemohli vidět pod podlahou. Všechny translace i rotace zůstávají stejné jako minule s tím rozdílem, že nyní už jde osa y klasickým směrem. Když posuneme reálný míč dolů, odraz jde nahoru a naopak.

```
glTranslatef(0.0f, height, 0.0f);// Umístění míče  
glRotatef(xrot, 1.0f, 0.0f, 0.0f);// Rotace na ose x  
glRotatef(yrot, 0.0f, 1.0f, 0.0f);// Rotace na ose y  
  
DrawObject();// Vykreslí míč
```

Zvětšíme hodnoty natočení míče a jeho odrazu o rychlost rotací. Před návratem z funkce zavoláme glFlush(), které počká na ukončení renderingu. Prevence mihotání na pomalejších grafických kartách.

```
xrot += xrotspeed;// Zvětší natočení  
yrot += yrotspeed;// Zvětší natočení  
  
glFlush();// Vyprázdni pipeline  
  
return TRUE;// Všechno v pořádku  
}
```

Následující funkce testuje stisk kláves. Voláme ji periodicky v hlavní smyčce WinMain(). Šípkami ovládáme rychlost rotace míče, klávesy A a Z přibližují/oddalují scénu, Page Up s Page Down umožňují změnit výšku plážového míče nad podlahou. Klávesa ESC plní stále svoji funkci, ale její umístění zůstalo ve WinMain().

```
void ProcessKeyboard();// Ovládání klávesnicí  
{  
    if (keys[VK_RIGHT]) yrotspeed += 0.08f;// Šipka vpravo zvýší rychlost y rotace  
    if (keys[VK_LEFT]) yrotspeed -= 0.08f;// Šipka vlevo sníží rychlost y rotace  
    if (keys[VK_DOWN]) xrotspeed += 0.08f;// Šipka dolů zvýší rychlost x rotace  
    if (keys[VK_UP]) xrotspeed -= 0.08f;// Šipka nahoru sníží rychlost x rotace  
  
    if (keys['A']) zoom +=0.05f;// A přiblíží scénu  
    if (keys['Z']) zoom -=0.05f;// Z oddálí scénu  
  
    if (keys[VK_PRIOR]) height += 0.03f;// Page Up zvětší vzdálenost míče nad podlahou  
    if (keys[VK_NEXT]) height -= 0.03f;// Page Down zmenší vzdálenost míče nad podlahou  
}
```

V CreateGLWindow() je úplně miniaturní změna, nicméně by bez ní program nefungoval. Ve struktuře PIXELFORMATDESCRIPTOR pfd nastavíme číslo, které vyjadřuje počet bitů stencil bufferu. Ve všech minulých lekcích jsme ho nepotřebovali, takže mu byla přiřazena nula. Při použití stencil bufferu MUSÍ být počet jeho bitů větší nebo roven jedné! Nám stačí jeden bit.

```
// Uprostřed funkce CreateGLWindow()  
  
static PIXELFORMATDESCRIPTOR pfd;// Oznamuje Windows jak chceme vše nastavit  
{  
    sizeof(PIXELFORMATDESCRIPTOR),// Velikost struktury  
    1,// Číslo verze  
    PFD_DRAW_TO_WINDOW |// Podpora okna  
    PFD_SUPPORT_OPENGL |// Podpora OpenGL  
    PFD_DOUBLEBUFFER, // Podpora double bufferingu  
    PFD_TYPE_RGBA, // RGBA formát  
    bits, // Barevná hloubka  
    0, 0, 0, 0, 0, 0, // Bity barev ignorovány  
    0, // Žádný alfa buffer  
    0, // Ignorován shift bit  
    0, // Žádný akumulární buffer  
    0, 0, 0, 0, // Akumulární bity ignorovány  
    16, // 16 bitový z-buffer  
    1, // Stencil buffer (DŮLEŽITÉ)  
    0, // Žádný auxiliary buffer
```



```
PFD_MAIN_PLANE, // Hlavní vykreslovací vrstva
0, // Rezervováno
0, 0, 0 // Maska vrstvy ignorována
};
```

Jak jsem se zmínil výše, test stisknutí kláves už nebudeme vykonávat přímo ve WinMain(), ale ve funkci ProcessKeyboard(), kterou voláme hned po vykreslení scény.

```
// Funkce WinMain()

    DrawGLScene(); // Vykreslí scénu
    SwapBuffers(hdc); // Prohodí buffery

    ProcessKeyboard(); // Vstup z klávesnice
```

Doufám, že jste si užili tuto lekci. Víím, že probírané téma nebylo zrovna nejjednodušší, ale co se dá dělat? Byl to jeden z nejtěžších tutoriálů, jak jsem kdy napsal. Pro mě je celkem snadné pochopit, co který řádek dělá a který příkaz se musí použít, aby vznikl požadovaný efekt. Ale sedněte si k počítači a pokuste se to vysvětlit lidem, kteří neví, co to je stencil buffer a možná o něm dokonce v životě neslyšeli (Překl.: Můj případ). Osobně si myslím, že i když mu napoprvé neporozumíte, po druhém přečtení by mělo být vše jasné...

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz> & Milan Turek <nalim.kerut (zavináč) email.cz>

Lekce 27 - Stíny

Představuje se vám velmi komplexní tutoriál na vrhání stínů. Efekt je doslova neuvěřitelný. Stíny se roztahují, ohýbají a zahalují i ostatní objekty ve scéně. Realisticky se pokroutí na stěnách nebo podlaze. Se vším lze pomocí klávesnice pohybovat ve 3D prostoru. Pokud ještě nejste se stencil bufferem a matematikou jako jedna rodina, nemáte nejmenší šanci.

Tento tutoriál má trochu jiný přístup - sumarizuje všechny vaše znalosti o OpenGL a přidává spoustu dalších. Každopádně byste měli stoprocentně chápat nastavování a práci se stencil bufferem. Pokud máte pocit, že v něčem existují mezery, zkuste se vrátit ke čtení dřívějších lekcí. Mimo jiné byste také měli mít alespoň malé znalosti o analytické geometrii (vektory, rovnice přímek a rovin, násobení matic...) - určitě mějte po ruce nějakou knihu. Já osobně používám zápisky z matematiky prvního semestru na univerzitě. Vždy jsem věděl, že se někdy budou hodit.

Nyní už ale ke kódu. Aby byl program přehledný, definujeme několik struktur. První z nich, `sPoint`, vyjadřuje bod nebo vektor v prostoru. Ukládá jeho `x`, `y`, `z` souřadnice.

```
struct sPoint// Souřadnice bodu nebo vektoru
{
    float x, y, z;
};
```

Struktura `sPlaneEq` ukládá hodnoty `a`, `b`, `c`, `d` obecné rovnice roviny, která je definována vzorcem $ax + by + cz + d = 0$.

```
struct sPlaneEq// Rovnice roviny
{
    float a, b, c, d;// Ve tvaru ax + by + cz + d = 0
};
```

Struktura `sPlane` obsahuje všechny informace potřebné k popsání trojúhelníku, který vrhá stín. Instance těchto struktur budou reprezentovat facy (čelo, stěna - nebudu překládat, protože je tento termín hodně používaný i v češtině) trojúhelníků. Facem se rozumí stěna trojúhelníku, která je přivrácená nebo odvrácená od pozorovatele. Jeden trojúhelník má vždy dva facy.

Pole `p[3]` definuje tři indexy v poli vertexů objektu, které dohromady tvoří tento trojúhelník. Druhé trojrozměrné pole, `normals[3]`, zastupuje normálový vektor každého rohu. Třetí pole specifikuje indexy sousedních faců. `PlaneEq` určuje rovnici roviny, ve které leží tento face a parametr `visible` oznamuje, jestli je face přivrácený (viditelný) ke zdroji světla nebo ne.

```
struct sPlane// Popisuje jeden face objektu
{
    unsigned int p[3];// Indexy 3 vertexů v objektu, které vytvářejí tento face
    sPoint normals[3];// Normálové vektory každého vertexu
    unsigned int neigh[3];// Indexy sousedních faců

    sPlaneEq PlaneEq;// Rovnice roviny facu
    bool visible;// Je face viditelný (přivrácený ke světlu)?
};
```

Poslední struktura, `glObject`, je mezi právě definovanými strukturami na nejvyšší úrovni. Proměnné `nPoints` a `nPlanes` určují počet prvků, které používáme v polích `points` a `planes`.

```
struct glObject// Struktura objektu
{
    GLuint nPoints;// Počet vertexů
    sPoint points[100];// Pole vertexů

    GLuint nPlanes;// Počet faců
    sPlane planes[200];// Pole faců
};
```

`GLvector4f` a `GLmatrix16f` jsou pomocné datové typy, které definujeme pro snadnější předávání parametrů funkci `VMatMult()`. Více později.

```
typedef float GLvector4f[4];// Nový datový typ
typedef float GLmatrix16f[16];// Nový datový typ
```

Nadefinujeme proměnné. `Obj` je objektem, který vrhá stín. Pole `ObjPos[]` definuje jeho polohu, roty jsou úhlem natočení

na osách x, y a speedy jsou rychlosti otáčení.

```
glObject obj;// Objekt, který vrhá stín

float ObjPos[] = { -2.0f, -2.0f, -5.0f };// Pozice objektu

GLfloat xrot = 0, xspeed = 0;// X rotace a x rychlost rotace objektu
GLfloat yrot = 0, yspeed = 0;// Y rotace a y rychlost rotace objektu
```

Následující čtyři pole definují světlo a další čtyři pole materiál. Použijeme je především v InitGL() při inicializaci scény.

```
float LightPos[] = { 0.0f, 5.0f, -4.0f, 1.0f };// Pozice světla
float LightAmb[] = { 0.2f, 0.2f, 0.2f, 1.0f };// Ambient světlo
float LightDif[] = { 0.6f, 0.6f, 0.6f, 1.0f };// Diffuse světlo
float LightSpc[] = { -0.2f, -0.2f, -0.2f, 1.0f };// Specular světlo

float MatAmb[] = { 0.4f, 0.4f, 0.4f, 1.0f };// Materiál - Ambient hodnoty (prostředí,
atmosféra)
float MatDif[] = { 0.2f, 0.6f, 0.9f, 1.0f };// Materiál - Diffuse hodnoty (rozptylování
světla)
float MatSpc[] = { 0.0f, 0.0f, 0.0f, 1.0f };// Materiál - Specular hodnoty (zrcadlivost)
float MatShn[] = { 0.0f };// Materiál - Shininess hodnoty (lesk)
```

Poslední dvě proměnné jsou pro kouli, na kterou dopadá stín objektu.

```
GLUQuadricObj *q;// Quadratic pro kreslení koule
float SpherePos[] = { -4.0f, -5.0f, -6.0f };// Pozice koule
```

Struktura datového souboru, který používáme pro definici objektu, není až tak složitá, jak na první pohled vypadá. Soubor se dělí do dvou částí: jedna část pro vertexy a druhá pro facy. První číslo první části určuje počet vertexů a po něm následují jejich definice. Druhá část začíná specifikací počtu faců. Na každém dalším řádku je celkem dvanáct čísel. První tři představují indexy do pole vertexů (každý face má tři vrcholy) a zbylých devět hodnot určuje tři normálové vektory (pro každý vrchol jeden). To je vše. Abych nezapomněl v adresáři Data můžete najít ještě tři podobné soubory.

```
24
-2 0.2 -0.2
2 0.2 -0.2
2 0.2 0.2
-2 0.2 0.2
-2 -0.2 -0.2
2 -0.2 -0.2
2 -0.2 0.2
-2 -0.2 0.2

-0.2 2 -0.2
0.2 2 -0.2
0.2 2 0.2
0.2 2 0.2
-0.2 -2 -0.2
0.2 -2 -0.2
0.2 -2 0.2
-0.2 -2 0.2

-0.2 0.2 -2
0.2 0.2 -2
0.2 0.2 2
-0.2 0.2 2
-0.2 -0.2 -2
0.2 -0.2 -2
0.2 -0.2 2
-0.2 -0.2 2

36
1 3 2 0 1 0 0 1 0 0 1 0
1 4 3 0 1 0 0 1 0 0 1 0
5 6 7 0 -1 0 0 -1 0 0 -1 0
5 7 8 0 -1 0 0 -1 0 0 -1 0
5 4 1 -1 0 0 -1 0 0 -1 0 0
5 8 4 -1 0 0 -1 0 0 -1 0 0
3 6 2 1 0 0 1 0 0 1 0 0
3 7 6 1 0 0 1 0 0 1 0 0
5 1 2 0 0 -1 0 0 -1 0 0 -1
```

```

5 2 6 0 0 -1 0 0 -1 0 0 -1
3 4 8 0 0 1 0 0 1 0 0 1
3 8 7 0 0 1 0 0 1 0 0 1

9 11 10 0 1 0 0 1 0 0 1 0
9 12 11 0 1 0 0 1 0 0 1 0
13 14 15 0 -1 0 0 -1 0 0 -1 0
13 15 16 0 -1 0 0 -1 0 0 -1 0
13 12 9 -1 0 0 -1 0 0 -1 0 0
13 16 12 -1 0 0 -1 0 0 -1 0 0
11 14 10 1 0 0 1 0 0 1 0 0
11 15 14 1 0 0 1 0 0 1 0 0
13 9 10 0 0 -1 0 0 -1 0 0 -1
13 10 14 0 0 -1 0 0 -1 0 0 -1
11 12 16 0 0 1 0 0 1 0 0 1
11 16 15 0 0 1 0 0 1 0 0 1

17 19 18 0 1 0 0 1 0 0 1 0
17 20 19 0 1 0 0 1 0 0 1 0
21 22 23 0 -1 0 0 -1 0 0 -1 0
21 23 24 0 -1 0 0 -1 0 0 -1 0
21 20 17 -1 0 0 -1 0 0 -1 0 0
21 24 20 -1 0 0 -1 0 0 -1 0 0
19 22 18 1 0 0 1 0 0 1 0 0
19 23 22 1 0 0 1 0 0 1 0 0
21 17 18 0 0 -1 0 0 -1 0 0 -1
21 18 22 0 0 -1 0 0 -1 0 0 -1
19 20 24 0 0 1 0 0 1 0 0 1
19 24 23 0 0 1 0 0 1 0 0 1

```

Právě představený soubor nahrává funkce ReadObject(). Pro pochopení podstaty by měly stačit komentáře.

```

inline int ReadObject(char *st, glObject *o)// Nahraje objekt
{
    FILE *file;// Handle souboru
    unsigned int i;// Řídící proměnná cyklů

    file = fopen(st, "r");// Otevře soubor pro čtení

    if (!file)// Podařilo se ho otevřít?
        return FALSE;// Pokud ne - konec funkce

    fscanf(file, "%d", &(o->nPoints));// Načtení počtu vertexů

    for (i = 1; i <= o->nPoints; i++)// Načítá vertexy
    {
        fscanf(file, "%f", &(o->points[i].x));// Jednotlivé x, y, z složky
        fscanf(file, "%f", &(o->points[i].y));
        fscanf(file, "%f", &(o->points[i].z));
    }

    fscanf(file, "%d", &(o->nPlanes));// Načtení počtu faců

    for (i = 0; i < o->nPlanes; i++)// Načítá facy
    {
        fscanf(file, "%d", &(o->planes[i].p[0]));// Načtení indexů vertexů
        fscanf(file, "%d", &(o->planes[i].p[1]));
        fscanf(file, "%d", &(o->planes[i].p[2]));

        fscanf(file, "%f", &(o->planes[i].normals[0].x));// Normálové vektory prvního
        vertexu
        fscanf(file, "%f", &(o->planes[i].normals[0].y));
        fscanf(file, "%f", &(o->planes[i].normals[0].z));

        fscanf(file, "%f", &(o->planes[i].normals[1].x));// Normálové vektory druhého
        vertexu
        fscanf(file, "%f", &(o->planes[i].normals[1].y));
        fscanf(file, "%f", &(o->planes[i].normals[1].z));

        fscanf(file, "%f", &(o->planes[i].normals[2].x));// Normálové vektory třetího
        vertexu
        fscanf(file, "%f", &(o->planes[i].normals[2].y));
    }
}

```

```

        fscanf(file, "%f", &(o->planes[i].normals[2].z));
    }

    return TRUE;// Vše v pořádku
}

```

Díky funkci SetConnectivity() začínají být věci zajímavé :-). Hledáme v ní ke každému facu tři sousední facy, se kterými má společnou hranu. Protože je zdrojový kód, abych tak řekl, trochu hůře pochopitelný, přidávám i pseudo kód, který by mohl situaci maličko objasnit.

Začátek funkce

```

{
    Postupně se prochází každý face (A) v objektu
    {
        V každém průchodu se znovu prochází všechny facy (B) objektu (zjišťuje se
        sousedství A s B)
        {
            Dále se projdou všechny hrany facu A
            {
                Pokud aktuální hrana ještě nemá přiřazeného souseda
                {
                    Projdou se všechny hrany facu B
                    {
                        Provedou se výpočty, kterými se zjistí, jestli je okraj A
                        stejný jako okraj B
                        Pokud ano
                        {
                            Nastaví se soused v A
                            Nastaví se soused v B
                        }
                    }
                }
            }
        }
    }
}

```

Konec funkce

Už chápete?

```

inline void SetConnectivity(glObject *o)// Nastavení sousedů jednotlivých faců
{
    unsigned int pli, p2i, plj, p2j;// Pomocné proměnné
    unsigned int P1i, P2i, P1j, P2j;// Pomocné proměnné
    unsigned int i, j, ki, kj;// Řídící proměnné cyklů

    for(i = 0; i < o->nPlanes-1; i++)// Každý face objektu (A)
    {
        for(j = i+1; j < o->nPlanes; j++)// Každý face objektu (B)
        {
            for(ki = 0; ki < 3; ki++)// Každý okraj facu (A)
            {
                if(!o->planes[i].neigh[ki])// Okraj ještě nemá souseda?
                {
                    for(kj = 0; kj < 3; kj++)// Každý okraj facu (B)
                    {

```

Nalezením dvou vertexů, které označují konce hrany a jejich porovnáním můžeme zjistit, jestli mají společný okraj. Část $(kj+1) \% 3$ označuje vertex umístěný vedle toho, o kterém uvažujeme. Ověříme, jestli jsou vertexy stejné. Protože může být jejich pořadí rozdílné musíme testovat obě možnosti.

```

// Výpočty pro zjištění sousedství
pli = ki;
plj = kj;

p2i = (ki+1) % 3;
p2j = (kj+1) % 3;

pli = o->planes[i].p[pli];
p2i = o->planes[i].p[p2i];
plj = o->planes[j].p[plj];

```



```

int InitGLObjects()// Inicializuje objekty
{
    if (!ReadObject("Data/Object2.txt", &obj))// Nahraje objekt
    {
        return FALSE;// Při chybě konec
    }

    SetConnectivity(&obj);// Pospojuje facy (najde sousedy)

    for (unsigned int i = 0; i < obj.nPlanes; i++)// Prochází facy
        CalcPlane(obj, &(obj.planes[i]));// Spočítá rovnici roviny facu

    return TRUE;// Vše v pořádku
}

```

Nyní přichází funkce, která renderuje stín. Na začátku nastavíme všechny potřebné parametry OpenGL a poté, ne na obrazovku, ale do stencil bufferu, vyrenderujeme stín. Dále vykreslíme vepředu před scénu velký šedý obdélník. Tam, kde byl stencil buffer modifikován se zobrazí šedé plochy - stín.

```

void CastShadow(glObject *o, float *lp)// Vržení stínu
{
    unsigned int i, j, k, jj;// Pomocné
    unsigned int p1, p2;// Dva body okraje vertexu, které vrhají stín
    sPoint v1, v2;// Vektor mezi světlem a předchozími body

```

Nejprve určíme, které povrchy jsou přivrácené ke světlu a to tak, že zjistíme, která strana facu je osvětlená. Provedeme to velice jednoduše: máme rovnici roviny ($ax + by + cz + d = 0$) i polohu světla, takže dosadíme x, y, z koordináty světla do rovnice. Nezajímá nás hodnota, ale znaménko výsledku. Pokud bude výsledek větší než nula, míří normálový vektor roviny na stranu ke světlu a rovina je osvětlená. Při záporném čísle míří vektor od světla, rovina je od něj odvrácená. Vyšel-li by výsledek nula, bude světlo ležet v rovině facu, ale tím se nebudeme zabývat.

```

    float side;// Pomocná proměnná

    for (i = 0; i < o->nPlanes; i++)// Projde všechny facy objektu
    {
        // Rozhodne jestli je face přivrácený nebo odvrácený od světla
        side = o->planes[i].PlaneEq.a * lp[0] + o->planes[i].PlaneEq.b * lp[1] + o->planes[i].PlaneEq.c * lp[2] + o->planes[i].PlaneEq.d * lp[3];

        if (side > 0)// Je přivrácený?
        {
            o->planes[i].visible = TRUE;
        }
        else// Není
        {
            o->planes[i].visible = FALSE;
        }
    }
}

```

Nastavíme parametry OpenGL, které jsou nutné pro vržení stínu. Vypneme světla, protože nebudeme renderovat do color bufferu (výstup na obrazovku), ale pouze do stencil bufferu. Ze stejného důvodu zakážeme pomocí glColorMask() vykreslování na obrazovku. Ačkoli je testování hloubky stále zapnuté, nechceme, aby stíny byly v depth bufferu reprezentovány pevnými objekty. Jako prevenci tedy nastavíme masku hloubky na GL_FALSE. Nakonec nastavíme stencil buffer tak, aby na místa v něm označená mohly být vykresleny stíny.

```

glDisable(GL_LIGHTING);// Vypne světla
glDepthMask(GL_FALSE);// Vypne zápis do depth bufferu
glDepthFunc(GL_EQUAL);// Funkce depth bufferu

glEnable(GL_STENCIL_TEST);// Zapne stencilové testy
glColorMask(0, 0, 0, 0);// Nekreslit na obrazovku
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);// Funkce stencilu

```

Protože máme zapnuté ořezávání zadních stran trojúhelníků (viz. InitGL()), specifikujeme, které strany jsou přední. Také nastavíme stencil buffer tak, aby se v něm při kreslení zvětšovaly hodnoty.

```

glFrontFace(GL_CCW);// Čelní stěna proti směru hodinových ručiček
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);// Zvyšování hodnoty stencilu

```

V cyklu projdeme každý face a pokud je označen jako viditelný (přivrácený ke světlu), zkontrolujeme všechny jeho okraje. Pokud vedle něj není žádný sousední face nebo sice má souseda, který ale není viditelný, našli jsme okraj objektu, který vrhá stín. Pokud se nad těmito dvěma podmínkami zamyslíme, zjistíte, že jsou pravdivé. Získali jsme první

dvě souřadnice čtyřúhelníku, který je stěnou stínu. V tomto případě si představte stín jako oblast, který je ohraničena na jedné straně objektem bránícím průchodu světelných paprsků, z druhé strany promítací rovinou (stěna místnosti) a na okrajích čtyřúhelníky, které se právě snažíme vykreslit. Už je to trochu jasnější?

```
for (i = 0; i < o->nPlanes; i++)// Každý face objektu
{
    if (o->planes[i].visible)// Je přivrácený ke světlu
    {
        for (j = 0; j < 3; j++)// Každý okraj facu
        {
            k = o->planes[i].neigh[j];// Index souseda (pomocný)
```

Nyní zjistíme, jestli je vedle aktuálního okraje face, který buď není viditelný nebo vůbec neexistuje (nemá souseda). Pokud podmínka platí, našli jsme okraj objektu, který vrhá stín.

```
// Pokud nemá souseda, který je přivrácený ke světlu
if ((!k) || (!o->planes[k-1].visible))
{
```

Rohy hrany právě ověřovaného trojúhelníku udávají první dva body stínu. Další dva získáme spočítáním směrového vektoru, který vychází ze světla, prochází bodem p1 popř. p2 a díky násobení stem pokračuje ve stejném směru někam do hlubin scény. Násobení stem bychom si mohli představit jako měřítko pro prodloužení vektoru a tudíž i polygonu, aby dosáhl až k promítací rovině a neskončil někde před ní.

Kreslení stínu hrubou silou použité zde, není zrovna nejvhodnější, protože má velmi velké nároky na grafickou kartu. Nekreslíme totiž pouze k promítací rovině, ale až za ni kód této lekce. (* 100). Pro větší účinnost by bylo vhodné modifikovat tento algoritmus tak, aby se polygony stínu ořezaly objektem, na který dopadá. Tento postup by ovšem byl mnohem náročnější na vymyšlení a asi by byl problematický sám o sobě.

```
// Našli jsme okraj objektu, který vrhá stín - nakreslíme polygon
p1 = o->planes[i].p[j];// První bod okraje
jj = (j+1) % 3;// Pro získání druhého okraje
p2 = o->planes[i].p[jj];// Druhý bod okraje

// Délka vektoru
v1.x = (o->points[p1].x - lp[0]) * 100;
v1.y = (o->points[p1].y - lp[1]) * 100;
v1.z = (o->points[p1].z - lp[2]) * 100;

v2.x = (o->points[p2].x - lp[0]) * 100;
v2.y = (o->points[p2].y - lp[1]) * 100;
v2.z = (o->points[p2].z - lp[2]) * 100;
```

Zbytek už je celkem snadný. Máme dva body s délkou a tak vykreslíme čtyřúhelník - jeden z mnoha okrajů stínu.

```
glBegin(GL_TRIANGLE_STRIP);// Nakreslí okrajový polygon stínu
glVertex3f(o->points[p1].x, o->points[p1].y, o->points[p1].z);
glVertex3f(o->points[p1].x + v1.x, o->points[p1].y + v1.y, o->points[p1].z + v1.z);
glVertex3f(o->points[p2].x, o->points[p2].y, o->points[p2].z);
glVertex3f(o->points[p2].x + v2.x, o->points[p2].y + v2.y, o->points[p2].z + v2.z);
glEnd();
```

V cyklech zůstaneme tak dlouho, dokud nenajdeme a nevykreslíme všechny okraje stínu.

```
    }
}
}
```

Nejjednodušší a nejpochopitelnější vysvětlení toho, proč vykreslujeme to samé ještě jednou, je obrázek - stíny budou pouze tam, kde být mají. Při vykreslování se nyní budou hodnoty ve stencil bufferu snižovat. Také si všimněte, že funkcí glFrontFace() budeme ořezávat opačné strany trojúhelníků.



```

glFrontFace(GL_CW); // Čelní stěna po směru hodinových ručiček
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR); // Snižování hodnoty stencilu

for (i=0; i < o->nPlanes; i++) // Každý face objektu
{
    if (o->planes[i].visible) // Je přivrácený ke světlu
    {
        for (j = 0; j < 3; j++) // Každý okraj facu
        {
            k = o->planes[i].neigh[j]; // Index souseda (pomocný)

            // Pokud nemá souseda, který je přivrácený ke světlu
            if (!(k || (!o->planes[k-1].visible)))
            {
                // Našli jsme okraj objektu, který vrhá stín - nakreslíme polygon
                p1 = o->planes[i].p[j]; // První bod okraje
                jj = (j+1) % 3; // Pro získání druhého okraje
                p2 = o->planes[i].p[jj]; // Druhý bod okraje

                // Délka vektoru
                v1.x = (o->points[p1].x - lp[0])*100;
                v1.y = (o->points[p1].y - lp[1])*100;
                v1.z = (o->points[p1].z - lp[2])*100;

                v2.x = (o->points[p2].x - lp[0])*100;
                v2.y = (o->points[p2].y - lp[1])*100;
                v2.z = (o->points[p2].z - lp[2])*100;

                glBegin(GL_TRIANGLE_STRIP); // Nakreslí okrajový polygon stínu
                glVertex3f(o->points[p1].x, o->points[p1].y, o->points[p1].z);
                glVertex3f(o->points[p1].x + v1.x, o->points[p1].y + v1.y, o->points[p1].z + v1.z);
                glVertex3f(o->points[p2].x, o->points[p2].y, o->points[p2].z);
                glVertex3f(o->points[p2].x + v2.x, o->points[p2].y + v2.y, o->points[p2].z + v2.z);
                glEnd();
            }
        }
    }
}

```

Až teď opravdu zobrazíme na scénu stíny. Na úrovni roviny obrazovky vykreslíme velký, šedý, poloprůhledný obdélník. Zobrazí se pouze ty pixely, které byly právě označeny ve stencil bufferu (na pozici stínu). Čím bude obdélník tmavší, tím tmavší bude i stín. Můžete zkusit jinou průhlednost nebo dokonce i barvu. Jak by se vám líbil červený, zelený nebo modrý stín? Žádný problém!

```

glFrontFace(GL_CCW); // Čelní stěna proti směru hodinových ručiček
glColorMask(1, 1, 1, 1); // Vykreslovat na obrazovku

// Vykreslení obdélníku přes celou scénu
glColor4f(0.0f, 0.0f, 0.0f, 0.4f); // Černá, 40% průhledná
glEnable(GL_BLEND); // Zapne blending
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Typ blendingu

glStencilFunc(GL_NOTEQUAL, 0, 0xffffffff); // Nastavení stencilu
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); // Neměnit hodnotu stencilu

glPushMatrix(); // Uloží matici
glLoadIdentity(); // Reset matice

```

```

glBegin(GL_TRIANGLE_STRIP); // Černý obdélník
glVertex3f(-0.1f, 0.1f, -0.10f);
glVertex3f(-0.1f, -0.1f, -0.10f);
glVertex3f( 0.1f, 0.1f, -0.10f);
glVertex3f( 0.1f, -0.1f, -0.10f);
glEnd();

```

```
glPopMatrix(); // Obnoví matici
```

Nakonec obnovíme změněné parametry OpenGL na výchozí hodnoty.

```

// Obnoví změněné parametry OpenGL
glDisable(GL_BLEND);
glDepthFunc(GL_LEQUAL);
glDepthMask(GL_TRUE);
glEnable(GL_LIGHTING);
glDisable(GL_STENCIL_TEST);
glShadeModel(GL_SMOOTH);
}

```

DrawGLScene(), ostatně jako vždycky, zajišťuje všechno vykreslování. Proměnná Minv bude reprezentovat OpenGL matici, wlp budou lokální koordináty a lp pomocná pozice světla.

```

int DrawGLScene(GLvoid) // Hlavní vykreslovací funkce
{
    GLmatrix16f Minv; // OpenGL matice
    GLvector4f wlp, lp; // Relativní pozice světla

```

Smažeme obrazovkový, hloubkový i stencil buffer. Resetujeme matici a přesuneme se o dvacet jednotek do obrazovky. Umístíme světlo, provedeme translaci na pozici koule a pomocí quadraticu ji vykreslíme.

```

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT); // Smaže buffery

glLoadIdentity(); // Reset matice
glTranslatef(0.0f, 0.0f, -20.0f); // Přesun 20 jednotek do hloubky

glLightfv(GL_LIGHT1, GL_POSITION, LightPos); // Umístění světla

glTranslatef(SpherePos[0], SpherePos[1], SpherePos[2]); // Umístění koule
gluSphere(q, 1.5f, 32, 16); // Vykreslení koule

```

Spočítáme relativní pozici světla vzhledem k lokálnímu souřadnicovému systému objektu, který vrhá stín. Do proměnné Min uložíme transformační matici objektu, ale obrácenou (vše se zápornými čísly a zadávané opačným pořadím), takže se stane invertovanou transformační maticí. Z lp vytvoříme kopii pozice světla a poté ho vynásobíme právě získanou OpenGL maticí. Jednoduše řečeno: na konci bude lp pozicí světla v souřadnicovém systému objektu.

```

glLoadIdentity(); // Reset matice
glRotatef(-yrot, 0.0f, 1.0f, 0.0f); // Rotace na ose y
glRotatef(-xrot, 1.0f, 0.0f, 0.0f); // Rotace na ose x

glGetFloatv(GL_MODELVIEW_MATRIX, Minv); // Uložení ModelView matice do Minv

lp[0] = LightPos[0]; // Uložení pozice světla
lp[1] = LightPos[1];
lp[2] = LightPos[2];
lp[3] = LightPos[3];

VMatMult(Minv, lp); // Vynásobení pozice světla OpenGL maticí

glTranslatef(-ObjPos[0], -ObjPos[1], -ObjPos[2]); // Posun záporně o pozici objektu

glGetFloatv(GL_MODELVIEW_MATRIX, Minv); // Uložení ModelView matice do Minv

wlp[0] = 0.0f; // Globální koordináty na nulu
wlp[1] = 0.0f;
wlp[2] = 0.0f;
wlp[3] = 1.0f;

VMatMult(Minv, wlp); // Originální globální souřadnicový systém relativně k lokálnímu

lp[0] += wlp[0]; // Pozice světla je relativní k lokálnímu souřadnicovému systému objektu

```

```
lp[1] += wlp[1];
lp[2] += wlp[2];
```

Vykreslíme místnost s objektem a potom zavoláme funkci CastShadow(), která vykreslí stín objektu. Předáváme jí referenci na objekt spolu s pozicí světla, která je nyní ve stejném souřadnicovém systému jako objekt.

```
glLoadIdentity();// Reset matice
glTranslatef(0.0f, 0.0f, -20.0f);// Přesun 20 jednotek do hloubky

DrawGLRoom();// Vykreslení místnosti

glTranslatef(ObjPos[0], ObjPos[1], ObjPos[2]);// Umístění objektu
glRotatef(xrot, 1.0f, 0.0f, 0.0f);// Rotace na ose x
glRotatef(yrot, 0.0f, 1.0f, 0.0f);// Rotace na ose y

DrawGLObject(obj);// Vykreslení objektu

CastShadow(&obj, lp);// Vržení stínu založené na siluetě
```

Abychom po spuštění dema viděli, kde se právě nachází světlo, vykreslíme na jeho pozici malý oranžový kruh (respektive kouli).

```
glColor4f(0.7f, 0.4f, 0.0f, 1.0f);// Oranžová barva

glDisable(GL_LIGHTING);// Vypne světlo
glDepthMask(GL_FALSE);// Vypne masku hloubky

glTranslatef(lp[0], lp[1], lp[2]);// Translace na pozici světla

// Pořád jsme v lokálním souřadnicovém systému objektu
gluSphere(q, 0.2f, 16, 8);// Vykreslení malé koule (reprezentuje světlo)

glEnable(GL_LIGHTING);// Zapne světlo
glDepthMask(GL_TRUE);// Zapne masku hloubky
```

Aktualizujeme rotaci objektu a ukončíme funkci.

```
xrot += xspeed;// Zvětšení úhlu rotace objektu
yrot += yspeed;

glFlush();
return TRUE;// Všechno v pořádku
}
```

Dále napíšeme speciální funkci DrawGLRoom(), která vykreslí místnost. Je jí obyčejná krychle.

```
void DrawGLRoom()// Vykreslí místnost (krychli)
{
    glBegin(GL_QUADS);// Začátek kreslení obdélníků
        // Podlaha
        glNormal3f(0.0f, 1.0f, 0.0f);// Normála směřuje nahoru
        glVertex3f(-10.0f,-10.0f,-20.0f);// Levý zadní
        glVertex3f(-10.0f,-10.0f, 20.0f);// Levý přední
        glVertex3f( 10.0f,-10.0f, 20.0f);// Pravý přední
        glVertex3f( 10.0f,-10.0f,-20.0f);// Pravý zadní

        // Strop
        glNormal3f(0.0f,-1.0f, 0.0f);// Normála směřuje dolů
        glVertex3f(-10.0f, 10.0f, 20.0f);// Levý přední
        glVertex3f(-10.0f, 10.0f,-20.0f);// Levý zadní
        glVertex3f( 10.0f, 10.0f,-20.0f);// Pravý zadní
        glVertex3f( 10.0f, 10.0f, 20.0f);// Pravý přední

        // Čelní stěna
        glNormal3f(0.0f, 0.0f, 1.0f);// Normála směřuje do hloubky
        glVertex3f(-10.0f, 10.0f,-20.0f);// Levý horní
        glVertex3f(-10.0f,-10.0f,-20.0f);// Levý dolní
        glVertex3f( 10.0f,-10.0f,-20.0f);// Pravý dolní
        glVertex3f( 10.0f, 10.0f,-20.0f);// Pravý horní

        // Zadní stěna
        glNormal3f(0.0f, 0.0f,-1.0f);// Normála směřuje k obrazovce
        glVertex3f( 10.0f, 10.0f, 20.0f);// Pravý horní
        glVertex3f( 10.0f,-10.0f, 20.0f);// Pravý spodní
```

```

glVertex3f(-10.0f,-10.0f, 20.0f);// Levý spodní
glVertex3f(-10.0f, 10.0f, 20.0f);// Levý zadní

// Levá stěna
glNormal3f(1.0f, 0.0f, 0.0f);// Normála směřuje doprava
glVertex3f(-10.0f, 10.0f, 20.0f);// Přední horní
glVertex3f(-10.0f,-10.0f, 20.0f);// Přední dolní
glVertex3f(-10.0f,-10.0f,-20.0f);// Zadní dolní
glVertex3f(-10.0f, 10.0f,-20.0f);// Zadní horní

// Pravá stěna
glNormal3f(-1.0f, 0.0f, 0.0f);// Normála směřuje doleva
glVertex3f( 10.0f, 10.0f,-20.0f);// Zadní horní
glVertex3f( 10.0f,-10.0f,-20.0f);// Zadní dolní
glVertex3f( 10.0f,-10.0f, 20.0f);// Přední dolní
glVertex3f( 10.0f, 10.0f, 20.0f);// Přední horní
glEnd();// Konec kreslení
}

```

Předtím než zapomenou... v DrawGLScene() jsme použili funkci VMatMult(), která násobí vektor maticí. Opět se jedná o implementaci vzorce z knížky o matematice.

```

void VMatMult(GLmatrix16f M, GLvector4f v)
{
    GLfloat res[4]; // Ukládá výsledky

    res[0] = M[ 0]*v[0] + M[ 4]*v[1] + M[ 8]*v[2] + M[12]*v[3];
    res[1] = M[ 1]*v[0] + M[ 5]*v[1] + M[ 9]*v[2] + M[13]*v[3];
    res[2] = M[ 2]*v[0] + M[ 6]*v[1] + M[10]*v[2] + M[14]*v[3];
    res[3] = M[ 3]*v[0] + M[ 7]*v[1] + M[11]*v[2] + M[15]*v[3];

    v[0] = res[0]; // Výsledek uloží zpět do v
    v[1] = res[1];
    v[2] = res[2];
    v[3] = res[3]; // Homogenní souřadnice
}

```

V Inicializaci OpenGL nejsou téměř žádné novinky. Na začátku nahrajeme a inicializujeme objekt, který vrhá stín, potom nastavíme obvyklé parametry a světla.

```

int InitGL(GLvoid) // Nastavení OpenGL
{
    if (!InitGLObjects()) // Nahraje objekt
        return FALSE;

    glShadeModel(GL_SMOOTH); // Jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glClearStencil(0); // Nastavení stencil bufferu
    glEnable(GL_DEPTH_TEST); // Povolí testování hloubky
    glDepthFunc(GL_LEQUAL); // Typ testování hloubky
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Perspektivní korekce

    glLightfv(GL_LIGHT1, GL_POSITION, LightPos); // Pozice světla
    glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmb); // Ambient světlo
    glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDif); // Diffuse světlo
    glLightfv(GL_LIGHT1, GL_SPECULAR, LightSp); // Specular světlo

    glEnable(GL_LIGHT1); // Zapne světlo 1
    glEnable(GL_LIGHTING); // Zapne světla
}

```

Materiály, které určují jak vypadají polygony při dopadu světla, jsou, myslím, novinkou. Nemusíme vepisovat žádné hodnoty, protože předávané pole jsou definovány na začátku tutoriálu. Materiály mimo jiné určují i barvu povrchu, takže při zapnutém světle nebude mít změna barvy pomocí glColor() žádný vliv (Překl. To jsem zjistil úplně náhodou. Nevím, jestli je to pravda obecně, ale minimálně v tomto demu ano.).

```

glMaterialfv(GL_FRONT, GL_AMBIENT, MatAmb); // Prostředí, atmosféra
glMaterialfv(GL_FRONT, GL_DIFFUSE, MatDif); // Rozptylování světla
glMaterialfv(GL_FRONT, GL_SPECULAR, MatSp); // Zrcadlivost
glMaterialfv(GL_FRONT, GL_SHININESS, MatShn); // Lesk

```

Abychom alespoň trochu zrychlili vykreslování, zapneme culling, takže se zadní strany trojúhelníků nebudou

vykreslovat. Která strana je odvrácená se určí podle pořadí zadávání vrcholů polygonů (po/proti směru hodinových ručiček).

```
glCullFace(GL_BACK); // Ořezávání zadních stran
glEnable(GL_CULL_FACE); // Zapne ořezávání
```

Budeme vykreslovat i nějaké koule, takže vytvoříme a inicializujeme quadratic.

```
q = gluNewQuadric(); // Nový quadratic
gluQuadricNormals(q, GL_SMOOTH); // Generování normálových vektorů pro světlo
gluQuadricTexture(q, GL_FALSE); // Nepotřebujeme texturovací koordináty

return TRUE; // V pořádku
}
```

Poslední funkcí tohoto tutoriálu je ProcessKeyboard(). Stejně jako vykreslování, tak i ona, se volá v každém průchodu hlavní smyčky programu. Ošetřuje uživatelské příkazy při stisku kláves. Jak se program zachová, popisují komentáře.

```
void ProcessKeyboard() // Ošetření klávesnice
{
    // Rotace objektu
    if (keys[VK_LEFT]) yspeed -= 0.1f; // Šipka vlevo - snižuje y rychlost
    if (keys[VK_RIGHT]) yspeed += 0.1f; // Šipka vpravo - zvyšuje y rychlost
    if (keys[VK_UP]) xspeed -= 0.1f; // Šipka nahoru - snižuje x rychlost
    if (keys[VK_DOWN]) xspeed += 0.1f; // Šipka dolů - zvyšuje x rychlost

    // Pozice objektu
    if (keys[VK_NUMPAD6]) ObjPos[0] += 0.05f; // '6' - pohybuje objektem doprava
    if (keys[VK_NUMPAD4]) ObjPos[0] -= 0.05f; // '4' - pohybuje objektem doleva

    if (keys[VK_NUMPAD8]) ObjPos[1] += 0.05f; // '8' - pohybuje objektem nahoru
    if (keys[VK_NUMPAD5]) ObjPos[1] -= 0.05f; // '5' - pohybuje objektem dolů

    if (keys[VK_NUMPAD9]) ObjPos[2] += 0.05f; // '9' - přibližuje objekt
    if (keys[VK_NUMPAD7]) ObjPos[2] -= 0.05f; // '7' oddaluje objekt

    // Pozice světla
    if (keys['L']) LightPos[0] += 0.05f; // 'L' - pohybuje světlem doprava
    if (keys['J']) LightPos[0] -= 0.05f; // 'J' - pohybuje světlem doleva

    if (keys['I']) LightPos[1] += 0.05f; // 'I' - pohybuje světlem nahoru
    if (keys['K']) LightPos[1] -= 0.05f; // 'K' - pohybuje světlem dolů

    if (keys['O']) LightPos[2] += 0.05f; // 'O' - přibližuje světlo
    if (keys['U']) LightPos[2] -= 0.05f; // 'U' - oddaluje světlo

    // Pozice koule
    if (keys['D']) SpherePos[0] += 0.05f; // 'D' - pohybuje koulí doprava
    if (keys['A']) SpherePos[0] -= 0.05f; // 'A' - pohybuje koulí doleva

    if (keys['W']) SpherePos[1] += 0.05f; // 'W' - pohybuje koulí nahoru
    if (keys['S']) SpherePos[1] -= 0.05f; // 'S' - pohybuje koulí dolů

    if (keys['E']) SpherePos[2] += 0.05f; // 'E' - přibližuje kouli
    if (keys['Q']) SpherePos[2] -= 0.05f; // 'Q' - oddaluje kouli
}
```

Několik poznámek ohledně tutoriálu

Na první pohled vypadá demo hyperefektně :-), ale má také své mouchy. Tak například koule nezastavuje projekci stínu na stěnu. V reálném prostředí by také vrhala stín, takže by se nic moc nestalo. Nicméně je zde pouze na ukázkou toho, co se se stínem stane na zakřiveném povrchu.

Pokud program běží extrémně pomalu, zkuste přepnout do fullscreenu nebo změnit barevnou hloubku na 32 bitů. Arseny L. napsal: "Pokud máte problémy s TNT2 v okenním módu, ujistěte se, že nemáte nastavenou 16bitovou barevnou hloubku. V tomto barevném módu je stencil buffer emulovaný, což ve výsledku znamená malý výkon. V 32bitovém módu je vše bez problémů."

napsal: Banu Cosmin - Choko & Brett Porter <brettporter (zavináč) yahoo.com>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 28 - Bezierovy křivky a povrchy, fullscreen fix

David Nikdel je osoba stojící za tímto skvělým tutoriálem, ve kterém se naučíte, jak se vytvářejí Bezierovy křivky. Díky nim lze velice jednoduše zakřivit povrch a provádět jeho plynulou animaci pouhou modifikací několika kontrolních bodů. Aby byl výsledný povrch modelu ještě zajímavější, je na něj namapována textura. Tutoriál také eliminuje problémy s fullscreenem, kdy se po návratu do systému neobnovilo původní rozlišení obrazovky.

Tento tutoriál je od začátku zamýšlen pouze jako úvod do Bezierových křivek, aby někdo mnohem šikovnější než já dokázal vytvořit něco opravdu skvělého. Neberte ho jako kompletní Bezier knihovnu, ale spíše jako koncept, jak tyto křivky pracují a co dokáží. Také prosím omluvte mou, v některých případech, ne až tak správnou terminologii. Doufám, že bude alespoň trochu srozumitelná. Abych tak řekl: Nikdo není dokonalý...

Pochopit Bezierovy křivky s nulovými znalostmi matematiky je nemožné. Proto bude následovat maličko delší sekce teorie, která by vás měla do problematiky alespoň trochu zasvětit. Pokud všechno už znáte, nic vám nebrání tuto (d) nou sekci přeskočit a věnovat se kódu.

Bezierovy křivky bývají primární metodou, jak v grafických editorech či obyčejných programech vykreslovat zakřivené linky. Jsou obvykle reprezentovány sérií bodů, z nich každé dva reprezentují tečnu ke grafu funkce.



Toto je nejjednodušší možná Bezierova křivka. Delší jsou tvořeny spojením několika dohromady. Je tvořena pouze čtyřmi body, dva konce a dva středové kontrolní body. Pro počítač jsou všechny úplně stejné, ale abychom si pomohli, spojujeme první a poslední dva. Linky budou vždy tečnami k ukončovacím bodům. Parametrické křivky jsou kresleny nalezením libovolného počtu bodů rovnoměrně rozprostřených po křivce, které se spojí čarami. Počtem bodů můžeme ovládat hranatost křivky a samozřejmě také dobu trvání výpočtů. Podaří-li se nám množství bodů správně regulovat, pozorovatel v každém okamžiku uvidí perfektně zakřivený povrch bez trhání animace.

Všechny Bezierovy křivky jsou založeny na základním vzorci funkce. Komplikovanější verze jsou z něj odvozeny.

$$t + (1 - t) = 1$$

Vypadá jednoduše? Ano, rovnice jednoduchá určitě je, ale nesmíme zapomenout na to, že je to pouze Bezierova křivka prvního stupně. Použijeme-li trochu terminologie: Bezierovy křivky jsou polynomiální (mnohočlenné). Jak si zajistíte pamatujete z algebry, první stupeň z polynomu je přímka - nic zajímavého. Základní funkce vychází, dosadíme-li libovolné číslo t . Rovnici můžeme ovšem také mocnit na druhou, na třetí, na jakékoli číslo, protože se obě strany rovnají jedné. Zkusíme ji tedy umocnit na třetí.

$$(t + (1 - t))^3 = 1^3$$
$$t^3 + 3t^2(1 - t) + 3t(1 - t)^2 + (1 - t)^3 = 1$$

Tuto rovnici použijeme k výpočtu mnohem více používanější křivky - Bezierovy křivky třetího stupně. Pro toto rozhodnutí existují dva důvody:

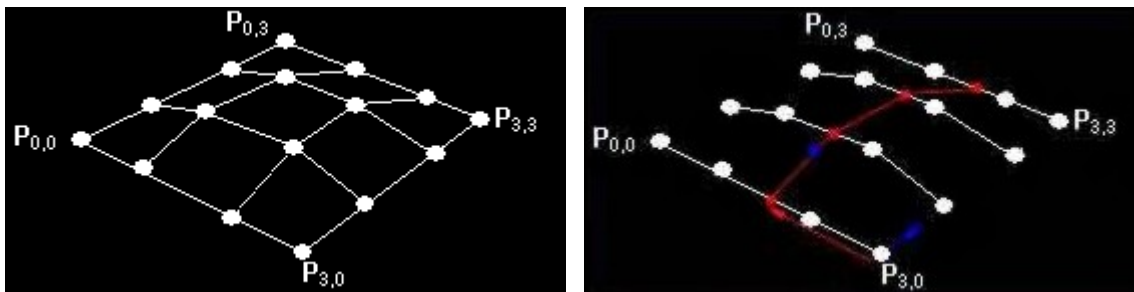
- Tento polynomiál je nejnižšího možného stupně, kdy už křivka nemusí ležet v rovině, ale i v prostoru.
- Tečny k funkci už nejsou závislé na jiných (křivky 2. stupně mohou mít pouze tři kontrolní body, my potřebujeme čtyři).

Zbývá ale dodat ještě jedna věc... Celá levá strana rovnice se rovná jedné, takže je bezpečné předpokládat, že pokud přidáme všechny složky měla by se stále rovnat jedné. Zní to, jako by to mohlo být použito k rozhodnutí kolik z každého kontrolního bodu lze použít při výpočtu bodu na křivce? (nápověda: Prostě řekni ano ;-) Ano. Správně! Pokud chceme spočítat hodnotu bodu v procentech vzdálenosti na křivce, jednoduše násobíme každou složku kontrolním bodem (stejně jako vektor) a nalezneme součet. Obecně budeme pracovat s hodnotami $0 \leq t \leq 1$, ale není to technicky nutné. Dokonale zmatení? Raději napíšu tu funkci.

$$P_1 * t^3 + P_2 * 3 * t^2 * (1-t) + P_3 * 3 * t * (1-t)^2 + P_4 * (1-t)^3 = P_{new}$$

Protože jsou polynomiály vždy spojité, jsou dobrou cestou k pohybu mezi čtyřmi body. Můžeme dosáhnout ale vždy pouze okrajových bodů (P1 a P4). Pokud tuto větu nechápete, podívejte se na první obrázek. V těchto případech se $t = 0$ popř. $t = 1$.

To je sice hezké, ale jak mám použít Bezierovy křivky ve 3D? Je to docela jednoduché. Potřebujeme 16 kontrolních bodů (4x4) a dvě proměnné t a v . Vytvoříme z nich čtyři paralelní křivky. Na každé z nich spočítáme jeden bod při určitém v a použijeme tyto čtyři body k vytvoření nové křivky a spočítáme t . Nalezením více bodů můžeme nakreslit triangle strip a tím zobrazit Bezierův povrch.



Přepokládám, že matematiky už bylo dost. Pojďme se vrhnout na kód této lekce. (Ze všeho nejdříve vytvoříme struktury. POINT_3D je obyčejný bod ve třírozměrném prostoru. Druhá struktura je už trochu zajímavější - představuje Bezierův povrch. Anchors[4][4] je dvourozměrné pole 16 řídicích bodů. Do display listu dlBPatch uložíme výsledný model a texture ukládá texturu, kterou na něj namapujeme.

```
typedef struct point_3d// Struktura bodu
{
    double x, y, z;
} POINT_3D;

typedef struct bpatch// Struktura Bezierova povrchu
{
    POINT_3D anchors[4][4];// Mřížka řídicích bodů (4x4)
    GLuint dlBPatch;// Display list
    GLuint texture;// Textura
} BEZIER_PATCH;
```

Mybezier je objektem právě vytvořené textury, rotz kontroluje úhel natočení scény. ShowCPoints indikuje, jestli vykreslujeme mřížku mezi řídicími body nebo ne. Divs určuje hladkost (hranatost) výsledného povrchu.

```
BEZIER_PATCH mybezier;// Bezierův povrch

GLfloat rotz = 0.0f;// Rotace na ose z
BOOL showCPoints = TRUE;// Flag pro zobrazení mřížky mezi kontrolními body
int divs = 7;// Počet interpolací (množství vykreslovaných polygonů)
```

Jestli si pamatujete, tak v úvodu jsem psal, že budeme maličko upravovat kód pro vytváření okna tak, aby se při návratu z fullscreenu obnovilo původní rozlišení obrazovky (některé grafické karty s tím mají problémy). DMsaved ukládá původní nastavení monitoru před vstupem do fullscreenu.

```
DEVMODE DMsaved;// Ukládá původní nastavení monitoru
```

Následuje několik pomocných funkcí pro jednoduchou vektorovou matematiku. Sčítání, násobení a vytváření 3D bodů. Nic složitého.

```
POINT_3D pointAdd(POINT_3D p, POINT_3D q)// Sčítání dvou bodů
{
    p.x += q.x;
    p.y += q.y;
    p.z += q.z;

    return p;
}

POINT_3D pointTimes(double c, POINT_3D p)// Násobení bodu konstantou
{
    p.x *= c;
    p.y *= c;
    p.z *= c;

    return p;
}
```

```

}
POINT_3D makePoint(double a, double b, double c)// Vytvoření bodu ze tří čísel
{
    POINT_3D p;

    p.x = a;
    p.y = b;
    p.z = c;

    return p;
}

```

Funkcí Bernstein() počítáme bod, který leží na Bezierově křivce. V parametrech jí předáváme proměnnou u, která specifikuje procentuální vzdálenost bodu od okraje křivky vzhledem k její délce a pole čtyř bodů, které jednoznačně definují křivku. Vícenásobným voláním a krokováním u vždy o stejný přírůstek můžeme získat aproximaci křivky.

```

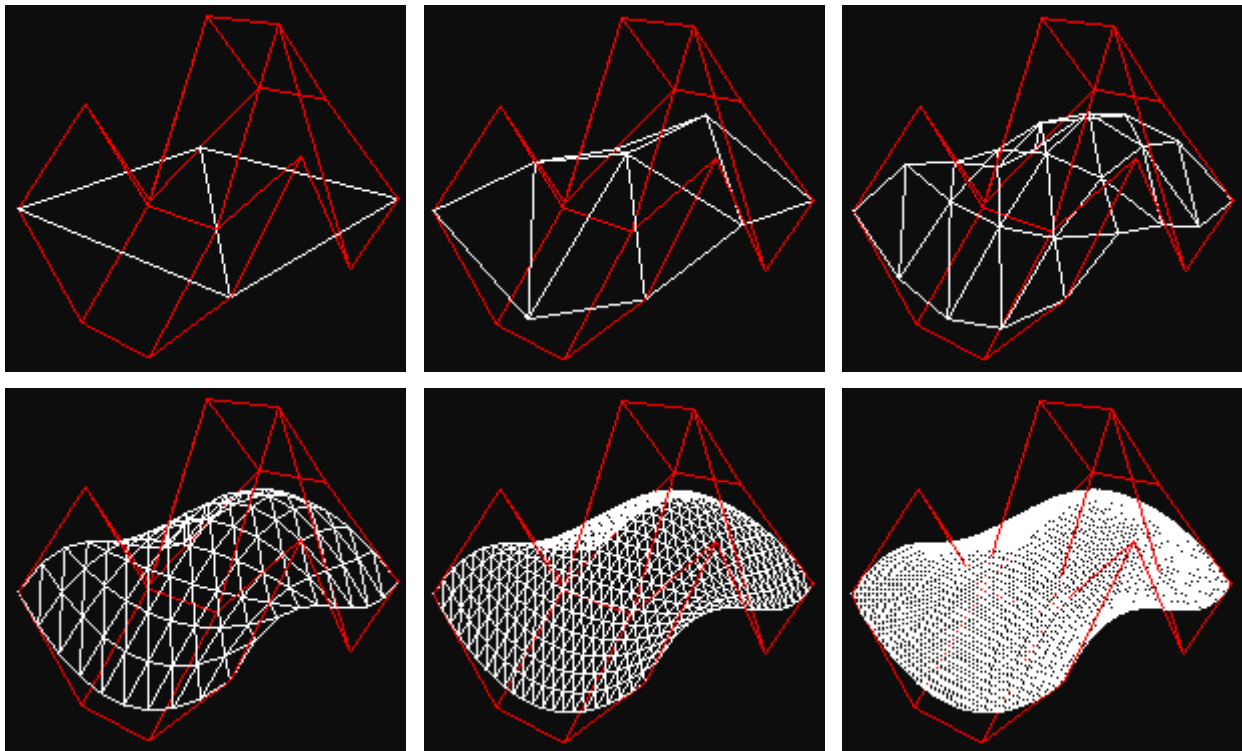
POINT_3D Bernstein(float u, POINT_3D *p)// Spočítá souřadnice bodu ležícího na křivce
{
    POINT_3D a, b, c, d, r;// Pomocné proměnné

    // Výpočet podle vzorce
    a = pointTimes(pow(u,3), p[0]);
    b = pointTimes(3 * pow(u,2) * (1-u), p[1]);
    c = pointTimes(3 * u * pow((1-u), 2), p[2]);
    d = pointTimes(pow((1-u), 3), p[3]);

    r = pointAdd(pointAdd(a, b), pointAdd(c, d));// Sečtení násobků a, b, c, d
    return r;// Vrácení výsledného bodu
}

```

Největší část práce odvádí funkce genBezier(). Spočítá křivky, vygeneruje triangle strip a výsledek uloží do display listu. Použití display listu je v tomto případě více než vhodné, protože nemusíme provádět složité výpočty při každém framu, ale pouze při změnách vyžádaných uživatelem. Odstraní se tím zbytečné zatížení procesoru. Funkci předáváme strukturu BEZIER_PATCH, v níž jsou uloženy všechny potřebné řídicí body. Divs určuje kolikrát budeme provádět výpočty - ovládá hranatost výsledného modelu. Následující obrázky jsou získány přepnutím do režimu vykreslování linek místo polygonů (glPolygonMode(GL_FRONT_AND_BACK, GL_LINES)) a zakázáním textur. Jasně je vidět, že čím je číslo v divs větší, tím je objekt zaoblenější.



```

GLuint genBezier(BEZIER_PATCH patch, int divs)// Generuje display list Bezierova povrchu
{

```

Proměnné u, v řídí cykly generující jednotlivé body na Bezierově křivce a py, px, pyold jsou jejich procentuální hodnoty,

kteří slouží k určení místa na křivce. Nabývají hodnot v intervalu od 0 do 1, takže je můžeme bez komplikací použít i jako texturovací koordináty. Drawlist je display list, do kterého kreslíme výsledný povrch. Do temp uložíme čtyři body pro získání pomocné Bezierovy křivky. Dynamické pole last ukládá minulý řádek bodů, protože pro triangle strip potřebujeme dva řádky.

```
int u = 0, v; // Řídící proměnné
float py, px, pyold; // Procentuální hodnoty

GLuint drawlist = glGenLists(1); // Display list

POINT_3D temp[4]; // Řídící body pomocné křivky
POINT_3D* last = (POINT_3D*) malloc(sizeof(POINT_3D) * (divs+1)); // První řada
polygonů

if (patch.dlBPatch != NULL) // Pokud existuje starý display list
    glDeleteLists(patch.dlBPatch, 1); // Smažeme ho

temp[0] = patch.anchors[0][3]; // První odvozená křivka (osa x)
temp[1] = patch.anchors[1][3];
temp[2] = patch.anchors[2][3];
temp[3] = patch.anchors[3][3];

for (v = 0; v <= divs; v++) // Vytvoří první řádek bodů
{
    px = ((float)v) / ((float)divs); // Px je procentuální hodnota v
    last[v] = Bernstein(px, temp); // Spočítá bod na křivce ve vzdálenosti px
}

glNewList(drawlist, GL_COMPILE); // Nový display list
glBindTexture(GL_TEXTURE_2D, patch.texture); // Zvolí texturu
```

Vnější cyklus prochází řádky a vnitřní jednotlivé sloupce. Nebo to může být i naopak. Záleží na tom, co si každý představí pod pojmy řádek a sloupec :-)

```
for (u = 1; u <= divs; u++) // Prochází body na křivce
{
    py = ((float)u) / ((float)divs); // Py je procentuální hodnota u
    pyold = ((float)u - 1.0f) / ((float)divs); // Pyold má hodnotu py při minulém
    průchodu cyklem
}
```

V každém prvku pole patch.anchors[] máme uloženy čtyři řídicí body (dvourozměrné pole). Celé pole dohromady tvoří čtyři paralelní křivky, které si označíme jako řádky. Nyní spočítáme body, které jsou umístěny na všech čtyřech křivkách ve stejné vzdálenosti py a uložíme je do pole temp[], které představuje sloupec v řádku a celkově tvoří čtyři řídicí body nové křivky pro sloupec.

Celou akci si představte jako trochu komplikovanější procházení dvourozměrného pole - vnější cyklus prochází řádky a vnitřní sloupce. Z upravených řídicích proměnných si vybíráme pozice bodů a texturovací koordináty. Py s pyold představuje dva "rovnoběžné" řádky a px sloupec. (Překl.: Než jsem tohle pochopil... v originále o tom nebyla ani zmínka).

```
temp[0] = Bernstein(py, patch.anchors[0]); // Spočítá Bezierovy body pro
křivku
temp[1] = Bernstein(py, patch.anchors[1]);
temp[2] = Bernstein(py, patch.anchors[2]);
temp[3] = Bernstein(py, patch.anchors[3]);

glBegin(GL_TRIANGLE_STRIP); // Začátek kreslení triangle stripu
for (v = 0; v <= divs; v++) // Prochází body na křivce
{
    px = ((float)v) / ((float)divs); // Px je procentuální hodnota v

    glTexCoord2f(pyold, px); // Texturovací koordináty z minulého
    průchodu
    glVertex3d(last[v].x, last[v].y, last[v].z); // Bod z minulého
    průchodu
}
```

Do pole last nyní uložíme nové hodnoty, které se při dalším průchodu cyklem stanou opět starými.

```
last[v] = Bernstein(px, temp); // Generuje nový bod

glTexCoord2f(py, px); // Nové texturové koordináty
glVertex3d(last[v].x, last[v].y, last[v].z); // Nový bod
}
```

```

        glEnd();// Konec triangle stripu
    }

    glEndList();// Konec display listu

    free(last);// Uvolní dynamické pole vertexů
    return drawlist;// Vrátí právě vytvořený display list
}

```

Jediná věc, kterou neděláme, ale která by se určitě mohla hodit, jsou normálové vektory pro světlo. Když na ně přijde, máme dvě možnosti. V první nalezneme střed každého trojúhelníku, aplikujeme na něj několik výpočtu k získání tečen k Bezierově křivce na osách x a y , vektorově je vynásobíme a tím získáme vektor kolmý současně k oběma tečnám. Po normalizování ho můžeme použít jako normálu. Druhý způsob je rychlejší a jednodušší, ale méně přesný. Můžeme cheatovat a použít normálový vektor trojúhelníku (spočítaný libovolným způsobem). Tím získáme docela dobrou aproximaci. Osobně preferuji druhou, jednodušší cestu, která ovšem nevypadá tak realisticky.

Ve funkci `initBezier()` inicializujeme matici kontrolních bodů na výchozí hodnoty. Pohrajte si s nimi, ať vidíte, jak jednoduše se dají měnit tvary povrchů.

```

void initBezier(void) // Počáteční nastavení kontrolních bodů
{
    mybezier.anchors[0][0] = makePoint(-0.75,-0.75,-0.5);
    mybezier.anchors[0][1] = makePoint(-0.25,-0.75, 0.0);
    mybezier.anchors[0][2] = makePoint( 0.25,-0.75, 0.0);
    mybezier.anchors[0][3] = makePoint( 0.75,-0.75,-0.5);
    mybezier.anchors[1][0] = makePoint(-0.75,-0.25,-0.75);
    mybezier.anchors[1][1] = makePoint(-0.25,-0.25, 0.5);
    mybezier.anchors[1][2] = makePoint( 0.25,-0.25, 0.5);
    mybezier.anchors[1][3] = makePoint( 0.75,-0.25,-0.75);
    mybezier.anchors[2][0] = makePoint(-0.75, 0.25, 0.0);
    mybezier.anchors[2][1] = makePoint(-0.25, 0.25,-0.5);
    mybezier.anchors[2][2] = makePoint( 0.25, 0.25,-0.5);
    mybezier.anchors[2][3] = makePoint( 0.75, 0.25, 0.0);
    mybezier.anchors[3][0] = makePoint(-0.75, 0.75,-0.5);
    mybezier.anchors[3][1] = makePoint(-0.25, 0.75,-1.0);
    mybezier.anchors[3][2] = makePoint( 0.25, 0.75,-1.0);
    mybezier.anchors[3][3] = makePoint( 0.75, 0.75,-0.5);

    mybezier.dlBPatch = NULL;// Display list ještě neexistuje
}

```

`InitGL()` je celkem standardní. Na jejím konci zavoláme funkce pro inicializaci kontrolních bodů, nahrání textury a vygenerování display listu Bezierova povrchu.

```

int InitGL(GLvoid) // Inicializace
{
    glEnable(GL_TEXTURE_2D);// Zapne texturování
    glShadeModel(GL_SMOOTH);// Jemné stínování
    glClearDepth(1.0f);// Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST);// Zapne testování hloubky
    glDepthFunc(GL_EQUAL);// Typ testování hloubky
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Černé pozadí
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);// Perspektivní korekce

    initBezier();// Inicializace kontrolních bodů
    LoadGLTexture(&(mybezier.texture), "./data/NeHe.bmp");// Loading textury
    mybezier.dlBPatch = genBezier(mybezier, divs);// Generuje display list Bezierova
    povrchu

    return TRUE;// Inicializace v pořádku
}

```

Vykreslování není oproti minulým tutoriálům vůbec složité. Po všech translacích a rotacích zavoláme display list a potom případně propojíme řídicí body červenými čarami. Chcete-li linky zapnout nebo vypnout stiskněte mezerník.

```

int DrawGLScene(GLvoid) // Všechno kreslení
{
    int i, j;// Řídicí proměnné cyklů

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smaže obrazovku a hloubkový
    buffer
    glLoadIdentity();// Reset matice

```

```

glTranslatef(0.0f, 0.0f, -4.0f); // Přesun do hloubky
glRotatef(-75.0f, 1.0f, 0.0f, 0.0f); // Rotace na ose x
glRotatef(rotz, 0.0f, 0.0f, 1.0f); // Rotace na ose z

glCallList(mybezier.dlBPatch); // Vykreslí display list Bezierova povrchu

if (showCPoints) // Pokud je zapnuté vykreslování mřížky
{
    glDisable(GL_TEXTURE_2D); // Vypne texturování
    glColor3f(1.0f, 0.0f, 0.0f); // Červená barva

    for(i = 0; i < 4; i++) // Horizontální linky
    {
        glBegin(GL_LINE_STRIP); // Kreslení linek
        for(j = 0; j < 4; j++) // Čtyři linky
        {
            glVertex3d(mybezier.anchors[i][j].x, mybezier.anchors[i][j].y,
                mybezier.anchors[i][j].z);
        }
        glEnd(); // Konec kreslení
    }

    for(i = 0; i < 4; i++) // Vertikální linky
    {
        glBegin(GL_LINE_STRIP); // Kreslení linek
        for(j = 0; j < 4; j++) // Čtyři linky
        {
            glVertex3d(mybezier.anchors[j][i].x, mybezier.anchors[j][i].y,
                mybezier.anchors[j][i].z);
        }
        glEnd(); // Konec kreslení
    }

    glColor3f(1.0f, 1.0f, 1.0f); // Bílá barva
    glEnable(GL_TEXTURE_2D); // Zapne texturování
}

return TRUE; // V pořádku
}

```

Práci s Bezierovými křivkami jsme úspěšně dokončili, ale ještě nesmíme zapomenout na fullscreen fix. Odstraňuje problém s přepínáním z fullscreenu do okenního módu, kdy některé grafické karty správně neobnovují původní rozlišení obrazovky (např. moje staříčká ATI Rage PRO a několik dalších). Doufám, že budete používat tento pozměněný kód, aby si každý mohl bez komplikací vychutnat vaše skvělá OpenGL dema. V tutoriálu jsme provedli celkem tři změny. První při deklaraci proměnných, kdy jsme vytvořili proměnnou DEVMODE DMSaved. Druhou najdete v CreateGLWindow(), kde jsme tuto pomocnou strukturu naplnili informacemi o aktuálním nastavení. Třetí změna je v KillGLWindow(), kde se obnovuje původní uložené nastavení.

```

BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag) //
Vytváření okna
{
    // Deklarace proměnných

    EnumDisplaySettings(NULL, ENUM_CURRENT_SETTINGS, &DMSaved); // Uloží aktuální
nastavení obrazovky

    // Vše ostatní zůstává stejné
}

GLvoid KillGLWindow(GLvoid) // Zavření okna
{
    if (fullscreen) // Jsme ve fullscreenu?
    {
        if (!ChangeDisplaySettings(NULL, CDS_TEST)) // Pokud pokusná změna nefunguje
        {
            ChangeDisplaySettings(NULL, CDS_RESET); // Odstraní hodnoty z registrů
            ChangeDisplaySettings(&DMSaved, CDS_RESET); // Použije uložené nastavení
        }
        else
        {
            ChangeDisplaySettings(NULL, CDS_RESET);
        }
    }
}

```

```

    }
    ShowCursor(TRUE); // Zobrazí ukazatel myši
}
// Vše ostatní zůstává stejné
}

```

Poslední věci jsou už standardní testy stisku kláves.

```

// Funkce WinMain()
if (keys[VK_LEFT]) // Šipka doleva
{
    rotz -= 0.8f; // Rotace doleva
}

if (keys[VK_RIGHT]) // Šipka doprava
{
    rotz += 0.8f; // Rotace doprava
}

if (keys[VK_UP]) // Šipka nahoru
{
    divs++; // Menší hranatost povrchu
    mybezier.dlBPatch = genBezier(mybezier, divs); // Aktualizace display listu
    keys[VK_UP] = FALSE;
}

if (keys[VK_DOWN] && divs > 1) // Šipka dolů
{
    divs--; // Větší hranatost povrchu
    mybezier.dlBPatch = genBezier(mybezier, divs); // Aktualizace display listu
    keys[VK_DOWN] = FALSE;
}

if (keys[VK_SPACE]) // Mezerník
{
    showCPoints = !showCPoints; // Zobrazí/skryje linky mezi řídícími body
    keys[VK_SPACE] = FALSE;
}

```

Doufám, že pro vás byl tento tutoriál poučný a že od nynějška miluje Bezierovy křivky stejně jako já ;-). Ještě jsem se o tom nezmínil, ale mnohé z vás jistě napadlo, že se s nimi dá vytvořit perfektní morfovací efekt. A velmi jednoduše! Nezapomeňte, se mění poloha pouze šestnácti bodů. Zkuste o tom popřemýšlet...

**napsal: David Nikdel <ogapo (zavináč) ithink.net>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>**

Lekce 29 - Blitter, nahrávání .RAW textur

V této lekci se naučíte, jak se nahrávají .RAW obrázky a konvertují se do textur. Dozvíte se také o blitteru, grafické metodě přenášení dat, která umožňuje modifikovat textury poté, co už byly nahrány do programu. Můžete jím zkopírovat část jedné textury do druhé, blendingem je smíchat dohromady a také roztahovat. Maličko upravíme program tak, aby v době, kdy není aktivní, vůbec nezatěžoval procesor.

Blitting... v počítačové grafice je toto slovo hodně používané. Označuje se jím zkopírování části jedné textury a vložení do druhé. Pokud programujete ve Win API nebo MFC, jistě jste slyšeli o funkcích BitBlt() nebo StretchBlt(). Přesně toto se pokusíme vytvořit.

Chcete-li napsat funkci, která implementuje blitting, měli byste něco vědět o lineární grafické paměti. Když se podíváte na monitor, vidíte spousty bodů reprezentujících nějaký obrázek, ovládací prvky nebo třeba kurzor myši. Vše je prostě složeno z matice pixelů. Ale jak ví grafická karta nebo BIOS, jak nakreslit bod například na souřadnicích [64; 64]? Jednoduše! Všechno, co je na obrazovce není v matici, ale v lineární paměti (v jednorozměrném poli). Pozici bodu v paměti můžeme získat následující rovnicí:

```
adresa_v_paměti = (pozice_y * rozlišení_obrazovky_x) + pozice_x
```

Pokud máme rozlišení obrazovky 640x480, bude bod [64; 64] umístěn na paměťové adrese $(64 \cdot 640) + 64 = 41024$. Protože paměť, do které budeme ukládat bitmapy je také lineární, můžeme této vlastnosti využít při přenášení bloků grafických dat. Výslednou adresu ještě budeme násobit barevnou hloubkou obrázku, protože nepoužíváme jedno-bytové pixely (256 barev), ale RGBA obrázky. Pokud jste tento výklad nepochopili, nemá cenu jít dál...

Vytvoříme strukturu TEXTURE_IMAGE, která bude obsahovat informace o nahrávaném obrázku - šířku, výšku, barevnou hloubku. Pointer data bude ukazovat do dynamické paměti, kam nahrajeme ze souboru data obrázku.

```
typedef struct Texture_Image// Struktura obrázku
{
    int width;// Šířka v pixelech
    int height;// Výška v pixelech
    int format;// Barevná hloubka v bytech na pixel
    unsigned char *data;// Data obrázku
} TEXTURE_IMAGE;
```

Další datový typ je ukazatelem na právě vytvořenou strukturu. Po něm následují dvě proměnné t1 a t2. Do nich budeme nahrávat obrázky, které potom blittingem sloučíme do jednoho a vytvoříme z něj texturu.

```
typedef TEXTURE_IMAGE *P_TEXTURE_IMAGE;// Datový typ ukazatele na obrázek

P_TEXTURE_IMAGE t1;// Dva obrázky
P_TEXTURE_IMAGE t2;

GLuint texture[1];// Jedna textura
```

Rot proměnné určují úhel rotace výsledného objektu. Nic nového.

```
GLfloat xrot;// X rotace
GLfloat yrot;// Y rotace
GLfloat zrot;// Z rotace
```

Funkcí AllocateTextureBuffer(), alokujeme dynamickou paměť pro obrázek a vrátíme ukazatel. Při neúspěchu se vrací NULL. Funkci předává program celkem tři parametry: šířku, výšku a barevnou hloubku v bytech na pixel.

```
P_TEXTURE_IMAGE AllocateTextureBuffer(GLint w, GLint h, GLint f)// Alokuje paměť pro
obrázek
{
```

Ukazatel na obrázek ti vrátíme na konci funkce volajícím kódu. Na začátku ho inicializujeme na NULL. Proměnné c, přiřadíme také NULL. Představuje úložiště nahrávaných dat.

```
    P_TEXTURE_IMAGE ti = NULL;// Ukazatel na strukturu obrázku
    unsigned char *c = NULL;// Ukazatel na data obrázku
```

Pomocí standardní funkce malloc() se pokusíme alokovat dynamickou paměť pro strukturu obrázku. Pokud se operace podaří, program pokračuje dále. Při jakékoli chybě vrátí malloc() NULL. Vypíšeme chybovou zprávu a oznámíme

volajícím kódu neúspěch.

```
ti = (P_TEXTURE_IMAGE)malloc(sizeof(TEXTURE_IMAGE)); // Alokace paměti pro strukturu
if(ti != NULL) // Podařila se alokace paměti?
{
```

Po úspěšné alokaci paměti vyplníme strukturu atributy obrázku. Barevná hloubka není v obvyklém formátu bit na pixel, ale kvůli jednodušší manipulaci s pamětí v bytech na pixel.

```
ti->width = w; // Nastaví atribut šířky
ti->height = h; // Nastaví atribut výšky
ti->format = f; // Nastaví atribut barevné hloubky
```

Stejným způsobem jako pro strukturu alokujeme paměť i pro data obrázku. Její velikost získáme násobením šířky, výšky a barevné hloubky. Při úspěchu nastavíme atribut data struktury na právě získanou dynamickou paměť, neúspěch ošetříme stejně jako minule.

```
c = (unsigned char *)malloc(w * h * f); // Alokace paměti pro strukturu
if (c != NULL) // Podařila se alokace paměti?
{
    ti->data = c; // Nastaví ukazatel na data
}
else // Alokace paměti pro data se nepodařila
{
    MessageBox(NULL, "Could Not Allocate Memory For A Texture Buffer", "BUFFER
    ERROR", MB_OK | MB_ICONINFORMATION);
}
```

Překl.: Tady by správně měla funkce vrátit namísto NULL proměnnou ti nebo ještě lépe před opuštěním funkce dealokovat dynamickou paměť struktury ti. Bez vrácení ukazatele nemůžeme z venku paměť uvolnit. Pokud operační systém nepracuje tak, jak má (Toto není narážka na MS Windows :-), čili po skončení neuvolní poskytne zdroje programu, vznikají paměťové úniky.

```
    // Uvolnění paměti struktury (Překl.)
    // free(ti);
    // ti = NULL;

    return NULL;
}

else // Alokace paměti pro strukturu se nepodařila
{
    MessageBox(NULL, "Could Not Allocate An Image Structure", "IMAGE STRUCTURE
    ERROR", MB_OK | MB_ICONINFORMATION);
    return NULL;
}
```

Pokud dosud nebyly žádné problémy, vrátíme ukazatel na strukturu ti.

```
return ti; // Vrátí ukazatel na dynamickou paměť
}
```

Ve funkci DeallocateTexture() děláme pravý opak - uvolňujeme paměť obrázku, na kterou ukazuje předaný parametr t.

```
void DeallocateTexture(P_TEXTURE_IMAGE t) // Uvolní dynamicky alokovanou paměť obrázku
{
    if(t) // Pokud struktura obrázku existuje
    {
        if(t->data) // Pokud existují data obrázku
        {
            free(t->data); // Uvolní data obrázku
        }

        free(t); // Uvolní strukturu obrázku
    }
}
```

Všechno už máme připravené, zbývá jenom nahrát .RAW obrázek. RAW formát je nejjednodušší a nejrychlejší způsob, jak nahrát do programu texturu (samozřejmě kromě funkce auxDIBImageLoad()). Proč je to tak jednoduché? Protože .RAW formát obsahuje pouze samotná data bitmapy bez hlaviček nebo něčeho dalšího. Jediné, co musíme udělat, je otevřít soubor a načíst data tak, jak jsou. Téměř... bohužel tento formát má dvě nevýhody. První je to, že ho

neotevřete v některých grafických editorech, o druhé později. Pochopíte sami :-)

Funkci předáváme název souboru a ukazatel na strukturu.

```
int ReadTextureData(char *filename, P_TEXTURE_IMAGE buffer)// Načte data obrázku
{
```

Deklarujeme handle souboru, řídicí proměnné cyklů a proměnnou done, která indikuje úspěch/neúspěch operace volajícímu kódu. Na začátku jí přiřadíme nulu, protože obrázek ještě není nahraný. Proměnnou stride, která určuje velikost řádku, hned na začátku inicializujeme na hodnotu získanou vynásobením šířky řádku v pixelech s barevnou hloubkou. Pokud bude obrázek široký 256 pixelů a barevná hloubka 4 byty (32 bitů, RGBA), velikost řádku bude celkem 1024 bytů. Pointer p ukazuje do paměti dat obrázku.

```
FILE *f;// Handle souboru
int i, j, k;// Řídicí proměnné cyklů
int done = 0;// Počet načtených bytů ze souboru (návrátová hodnota)

int stride = buffer->width * buffer->format;// Velikost řádku
unsigned char *p = NULL;// Ukazatel na aktuální byte paměti
```

Otevřeme soubor pro čtení v binárním módu.

```
f = fopen(filename, "rb");// Otevře soubor

if(f != NULL)// Podařilo se ho otevřít?
{
```

Pokud soubor existuje a šel otevřít, začneme se postupně vnořovat do cyklů. Vše by bylo velice jednoduché, kdyby .RAW formát byl trochu jinak uspořádán. Řádky vedou, jak je obvyklé, zleva doprava, ale jejich pořadí je invertované. To znamená, že první řádek je poslední, druhý předposlední atd. Vnější cyklus tedy nastavíme tak, aby řídicí proměnná ukazovala dolů na začátek obrázku. Soubor načítáme od začátku, ale hodnoty ukládáme od konce paměti vzhůru. Výsledkem je převrácení obrázku.

```
for(i = buffer->height-1; i >= 0 ; i--)// Od zdola nahoru po řádcích
{
```

Nastavíme ukazatel, kam se právě ukládá, na správný řádek paměti. Jejím začátkem je samozřejmě buffer->data. Sečteme ho s umístěním od začátku i * velikost řádku. Představte si, že buffer->data je stránka v paměti a i * stride představuje offset. Je to úplně stejné. Offsetem se pohybujeme po přidělené stránce. Na začátku je maximální a postupně klesá. Výsledkem je, že v paměti postupujeme vzhůru. Myslím, že je to pochopitelné.

```
p = buffer->data + (i * stride);// P ukazuje na požadovaný řádek
```

Druhým cyklem se pohybujeme zleva doprava po pixelech obrázku (ne bytech!).

```
for (j = 0; j < buffer->width; j++)// Zleva doprava po pixelech
{
```

Třetí cyklus prochází jednotlivé byty v pixelu. Pokud barevná hloubka (= byty na pixel) bude 4, cyklus projde celkem 3x (od 0 do 2; format-1). Důvodem odečtení jedničky je, že většina .RAW obrázků neobsahuje alfa hodnotu, ale pouze RGB složky. Alfa nastavíme ručně.

Všimněte si také, že každým průchodem inkrementujeme tři proměnné: k, p a done. Řídicí proměnná k je jasná. P ukazovalo před vstupem do všech cyklů na začátek posledního řádku v paměti. Postupně ho inkrementujeme až dosáhne úplného konce. Potom ho nastavíme na předposlední řádek atd. Done na konci funkce vrátíme, označuje celkový počet načtených bytů.

```
for (k = 0; k < buffer->format-1; k++, p++, done++)// Jednotlivé byty v
pixelu
{
```

Funkce fgetc() načte ze souboru f jeden znak a vrátí ho. Tento znak má velikost 1 byte (Už víte proč zrovna unsigned char?). Považujeme ho za složku barvy. Protože se cyklus po třetím průchodu zastaví, načteme a uložíme složky R, G a B.

```
*p = fgetc(f);// Načte R, G a B složku barvy
}
```

Po opuštění cyklu přiřadíme alfu a opět inkrementujeme ukazatel, aby se posunul na další byte.

Překl.: Tady se hodí poznamenat, že alfa nemusí být zrovna 255 (neprůhledná), ale můžeme ji nastavit na polovinu (122) a tak vytvořit poloprůhlednou texturu. Nebo si říct, že pixel o určitých složkách RGB bude průhledný. Většinou se vezme černá nebo bílá barva, ale nic nebrání např. načtení levého horního pixelu obrázku a zprůhlednění všech ostatních pixelů se stejným RGB. Nebo postupně, jak načítáme jednotlivé pixely v řádku, snižovat alfu od 255 do 0.

Textura bude vlevo neprůhledná a vpravo průhledná - plynulý přechod. S průhledností se dělají hodně kvalitní efekty. Maličké upozornění na konec: Efekty s alfa hodnotou jsou možné nejen u .RAW textur. Nezapomeňte, že už v 6. lekcí !!! jsme měli přístup k datům textury. Funkci glTexImage2D() jsme na konci LoadGLTextures() předávali parametr data!

```

        *p = 255; // Alfa neprůhledná (ruční nastavení)
        p++; // Ukazatel na další byte
    }
}

```

Poté, co projdeme všechny byty v pixelu, pixely v řádku a řádky v souboru se všechny cykly ukončí. Uf, Konečně! :-). Po ukončení cyklů zavřeme soubor.

```

    fclose(f); // Zavře soubor
}

```

Pokud byly problémy s otevřením souboru (neexistuje ap.) zobrazíme chybovou zprávu.

```

else // Soubor se nepodařilo otevřít
{
    MessageBox(NULL, "Unable To Open Image File", "IMAGE ERROR", MB_OK |
        MB_ICONINFORMATION);
}

```

Nakonec vrátíme done. Pokud se soubor nepodařilo otevřít a my nic nenačetli, obsahuje nulu. Pokud bylo vše v pořádku done se rovná počtu načtených bytů.

```

return done; // Vrátí počet načtených bytů
}

```

Máme loadovaná data obrázku, takže vytvoříme texturu. Funkci předáváme ukazatel na obrázek. Vygenerujeme texturu, nastavíme ji jako aktuální, zvolíme lineární filtrování pro zvětšení i zmenšení a nakonec vytvoříme mipmapovanou texturu. Vše je úplně stejné jako s knihovnou glaux, ale s tím rozdílem, že jsme si obrázek tentokrát nahráli sami.

```

void BuildTexture(P_TEXTURE_IMAGE tex) // Vytvoří texturu
{
    glGenTextures(1, &texture[0]); // Generuje texturu
    glBindTexture(GL_TEXTURE_2D, texture[0]); // Vybere texturu za aktuální

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Lineární
    filtrování
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    // Mipmapovaná textura
    gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, tex->width, tex->height, GL_RGBA,
        GL_UNSIGNED_BYTE, tex->data);
}

```

Funkci Blit(), která implementuje blitting, je předávána spousta parametrů. Co vyjadřují? Vezmeme to hezky popořadě. Src je zdrojovým obrázkem, jehož data vkládáme do cílového obrázku dst. Ostatní parametry vyznačují, která data se zkopírují (obdélník určený čtyřmi src_* čísly), kam se mají do cílového obrázku umístit (dst_*) a jakým způsobem (blending, popř. alfa hodnota).

```

// Blitting obrázků
void Blit(P_TEXTURE_IMAGE src, // Zdrojový obrázek
    P_TEXTURE_IMAGE dst, // Cílový obrázek
    int src_xstart, // Levý horní bod kopírované oblasti
    int src_ystart, // Levý horní bod kopírované oblasti
    int src_width, // Šířka kopírované oblasti
    int src_height, // Výška kopírované oblasti
    int dst_xstart, // Kam kopírovat (levý horní bod)
    int dst_ystart, // Kam kopírovat (levý horní bod)
    int blend, // Použít blending?
    int alpha) // Hodnota alfy při blendingu
{

```

Po řídicích proměnných cyklů deklaruje pomocné proměnné s a d, které ukazují do paměti obrázků. Dále ošetříme předávané parametry tak, aby alfa hodnota byla v rozmezí 0 až 255 a blend 0 nebo 1.

```

    int i, j, k; // Řídící proměnné cyklů
    unsigned char *s, *d; // Pomocné ukazatele na data zdroje a cíle

    if(alpha > 255) // Je alfa mimo rozsah?
        alpha = 255;

```



```

if(alpha < 0)
    alpha = 0;

if(blend < 0)// Je blending mimo rozsah?
    blend = 0;
if(blend > 1)
    blend = 1;

```

Překl.: Celé kopírování raději vysvětlím na příkladu, bude snáze pochopitelné. Máme obrázek 256 pixelů široký a chceme zkopírovat např. oblast od 50. do 200. pixelu o určité výšce. Před vstupem do cyklu se přesuneme na první kopírovaný řádek. Potom skočíme na 50. pixel zleva, zkopírujeme 150 pixelů a skočíme na konec řádku přes zbývajících 56 pixelů. Vše opakujeme pro další řádek, dokud nezkopírujeme celý požadovaný obdélník dat zdrojového obrázku do cílového.



Nyní nastavíme ukazatele d a s. Cílový ukazatel získáme sečtením adresy, kde začínají data cílového obrázku s offsetem, který je výsledkem násobení y pozice, kam začneme kopírovat, šířkou obrázku v pixelech a barevnou hloubkou obrázku. Tímto získáme řádek, na kterém začínáme kopírovat. Zdrojový ukazatel určíme analogicky.

```

// Ukazatele na první kopírovaný řádek
d = dst->data + (dst_ystart * dst->width * dst->format);
s = src->data + (src_ystart * src->width * src->format);

```

Vnější cyklus prochází kopírované řádky od shora dolů.

```

for (i = 0; i < src_height; i++)// Řádky, ve kterých se kopírují data
{

```

Už máme ukazatel nastaven na správný řádek, ale ještě musíme přičíst x-ovou pozici, která se opět násobí barevnou hloubkou. Akci provedeme pro zdrojový i cílový ukazatel.

```

// Posun na první kopírovaný pixel v řádku
s = s + (src_xstart * src->format);
d = d + (dst_xstart * dst->format);

```

Pointery nyní ukazují na první kopírovaný pixel. Začneme cyklus, který v řádku prochází jednotlivé pixely.

```

for (j = 0; j < src_width; j++)// Pixely v řádku, které se mají kopírovat
{

```

Nejvnitřnější cyklus prochází jednotlivé byty v pixelu. Všimněte si, že se také inkrementují pozice ve zdrojovém i cílovém obrázku.

```

for(k = 0; k < src->format; k++, d++, s++)// Byty v kopírovaném pixelu
{

```

Přichází nejzajímavější část - vytvoření alfablendingu. Představte si, že máte dva pixely: červený (zdroj) a zelený (cíl). Oba leží na stejných souřadnicích. Pokud je nezprůhledníte, půjde vidět pouze jeden z nich, protože původní pixel bude nahrazen novým. Jak jistě víte, každý pixel se skládá ze tří barevných kanálů RGB. Chceme-li vytvořit alfa blending, musíme nejdříve spočítat opačnou hodnotu alfa kanálu a to tak, že odečteme tuto hodnotu od maxima (255 - alpha). Násobíme jí cílový (zelený) pixel a sečteme ho se zdrojovým (červeným), který jsme násobili neupravenou alfou. Jsme skoro hotovi. Konečnou barvu vypočítáme dělením výsledku maximální hodnotou průhlednosti (255). tuto operaci z důvodu větší rychlosti vykonává bitový posun doprava o osm bitů. A je to! Máme pixel složený z obou předcházejících pixelů. Všimněte si, že se výpočty postupně provádějí se všemi kanály RGBA. Víte, co jsme právě implementovali? OpenGL techniku `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.

```

if (blend)// Je požadován blending?
{
    *d = ((*s * alpha) + (*d * (255-alpha))) >> 8;// Sloučení dvou
    pixelů do jednoho
}

```

Pokud nebudeme chtít blending, jednoduše zkopírujeme data ze zdrojové bitmapy do cílové. Žádná matematika, alfa se

ignoruje.

```
        else// Bez blendingu
        {
            *d = *s;// Obyčejné kopírování
        }
    }
}
```

Dojdeme-li až na konec kopírované oblasti, zvětšíme ukazatel tak, aby se dostal na konec řádku. Pokud dobře rozumíme ukazatelům a pamětovým operacím, je blitting hračkou.

```
    // Skočí ukazatelem na konec řádku
    d = d + (dst->width - (src_width + dst_xstart)) * dst->format;
    s = s + (src->width - (src_width + src_xstart)) * src->format;
}
}
```

Inicializace je tentokrát změněna od základů. Alokujeme paměť pro dva obrázky veliké 256 pixelů, které mají barevnou hloubku 4 byty (RGBA). Poté se je pokusíme nahrát. Pokud něco nevyjde vypíšeme chybovou zprávu a ukončíme program.

```
int InitGL(GLvoid)// Inicializace
{
    t1 = AllocateTextureBuffer(256, 256, 4);// Alokace paměti pro první obrázek

    if (ReadTextureData("Data/Monitor.raw", t1) == 0)// Nahraje data obrázku
    {
        // Nic se nenahrálo
        MessageBox(NULL, "Could Not Read 'Monitor.raw' Image Data", "TEXTURE ERROR",
            MB_OK | MB_ICONINFORMATION);
        return FALSE;
    }

    t2 = AllocateTextureBuffer(256, 256, 4);// Alokace paměti pro druhý obrázek

    if (ReadTextureData("Data/GL.raw", t2) == 0)// Nahraje data obrázku
    {
        // Nic se nenahrálo
        MessageBox(NULL, "Could Not Read 'GL.raw' Image Data", "TEXTURE ERROR", MB_OK |
            MB_ICONINFORMATION);
        return FALSE;
    }
}
```

Pokud jsme se dostali až tak daleko, je bezpečné předpokládat, že můžeme pracovat s daty obrázků, které se pokusíme blittingem sloučit do jednoho. Předáme je funkci - obrázek t2 jako zdrojový, t1 jako cílový. Výsledný obrázek získaný sloučením se uloží do t1. Vytvoříme z něj texturu.

```
    // Blitting obrázků
    Blit(t2,// Zdrojový obrázek
        t1,// Cílový obrázek
        127,// Levý horní bod kopírované oblasti
        127,// Levý horní bod kopírované oblasti
        128,// Šířka kopírované oblasti
        128,// Výška kopírované oblasti
        64,// Kam kopírovat (levý horní bod)
        64,// Kam kopírovat (levý horní bod)
        1,// Použít blending?
        128)// Hodnota alfy při blendingu

    BuildTexture(t1);// Vytvoří texturu
```

Překl.: Původně jsem chtěl vložit obrázky, abyste věděli, jak vypadají, ale bohužel ani jeden grafický editor, který mám zrovna doma .RAW formát nepodporuje. V anglickém tutoriálu je zmíněno, že Adobe Photoshop to svede. Ale poradil jsem si... víte jak? OpenGL.



Potom, co je vytvořena textura, můžeme uvolnit paměť obou obrázků.

```
DeallocateTexture(t1); // Uvolní paměť obrázků
DeallocateTexture(t2);
```

Následují běžná nastavení OpenGL.

```
glEnable(GL_TEXTURE_2D); // Zapne texturování

glShadeModel(GL_SMOOTH); // Jemné stínování
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Černé pozadí

glClearDepth(1.0); // Povolí mazání depth bufferu
glEnable(GL_DEPTH_TEST); // Zapne testování hloubky
glDepthFunc(GL_LESS); // Typ testování hloubky

return TRUE;
}
```

DrawGLScene() renderuje obyčejnou krychli - to už určitě znáte.

```
GLvoid DrawGLScene(GLvoid) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže buffery
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f, 0.0f, -10.0f); // Přesun do hloubky

    glRotatef(xrot, 1.0f, 0.0f, 0.0f); // Rotace
    glRotatef(yrot, 0.0f, 1.0f, 0.0f);
    glRotatef(zrot, 0.0f, 0.0f, 1.0f);

    glBindTexture(GL_TEXTURE_2D, texture[0]); // Zvolí texturu

    glBegin(GL_QUADS); // Začátek kreslení obdélníků
        // Čelní stěna
        glNormal3f(0.0f, 0.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
        // Zadní stěna
        glNormal3f(0.0f, 0.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        // Horní stěna
        glNormal3f(0.0f, 1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
        // Dolní stěna
        glNormal3f(0.0f, -1.0f, 0.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
```

```

glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,-1.0f,-1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,-1.0f,-1.0f);
// Pravá stěna
glNormal3f(1.0f, 0.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,-1.0f,-1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f,-1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f,-1.0f, 1.0f);
// Levá stěna
glNormal3f(-1.0f, 0.0f, 0.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,-1.0f,-1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f,-1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f,-1.0f);
glEnd();// Konec kreslení

xrot += 0.3f;// Zvětši úhly rotace
yrot += 0.2f;
zrot += 0.4f;
}

```

Maličko upravíme kód WinMain(). Pokud není program aktivní (např. minimalizovaný), zavoláme WaitMessage(). Všechno se zastaví, dokud program neobdrží nějakou zprávu (obvyčně o maximalizaci okna). Ve výsledku dosáhneme toho, že pokud program není aktivní nebude vůbec zatěžovat procesor.

```

// Funkce WinMain() - v hlavní smyčce programu
if (!active)// Je program neaktivní?
{
    WaitMessage();// Čekej na zprávu a zatím nic nedělej
}

```

Takže to bychom měli. Nyní máte ve svých hrách, enginech, demech nebo jakýchkoli programech všechny dveře otevřené pro vytváření velmi efektních blending efektů. S texturovými buffery můžete vytvářet věci jako například real-time plazmu nebo vodu. Vzájemnou kombinací více obrázků (i několikrát za sebou) je možné dosáhnout téměř fotorealistického terénu. Hodně štěstí.

napsal: Andreas Löffler & Rob Fletcher
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz> & Václav Slováček - Wessan <horizont (zavináč) host.sk>

Lekce 30 - Detekce kolizí

Na podobný tutoriál jste už jistě netrpělivě čekali. Naučíte se základy o detekcích kolizí, jak na ně reagovat a na fyzice založené modelovací efekty (nárazy, působení gravitace ap.). Tutoriál se více zaměřuje na obecnou funkci kolizí než zdrojovým kódům. Nicméně důležité části kódu jsou také popsány. Neočekávejte, že po prvním přečtení úplně všemu z kolizí porozumíte. Je to komplexní námět, se kterým vám pomohu začít.

Zdrojový kód, na němž tato lekce staví, pochází z mého dřívějšího příspěvku do jedné soutěže (najdete na OGLchallenge.dhs.org). Tématem byly Bláznivé kolize a můj příspěvek (mimořádně, získal první místo :-)) se jmenoval Magická místnost.

Detekce kolizí jsou obtížné, dodnes nebylo nalezeno žádné snadné řešení. Samozřejmě existují velice obecné algoritmy, které umí pracovat s jakýmkoli druhem objektů, ale očekávejte od nich také patřičnou cenu. My budeme zkoumat postupy, které jsou velmi rychlé, relativně snadné k pochopení a v rámci mezí celkem flexibilní. Důraz musí být vložen nejen na detekci kolize, ale i na reakci objektů na náraz. Je důležité, aby se vše dělo podle fyzikálních zákonů. Máme mnoho věcí na práci! Pojdme se tedy podívat, co všechno se v této lekci naučíme:

Detekce kolizí mezi

- Pohybující se koulí a rovinou
- Pohybující se koulí a válcem
- Dvěma pohybujícími se koulemi

Fyzikálně založené modelování

- Reakce na kolize - odrazy
- Pohyb v gravitaci za použití Eulerových rovnic

Speciální efekty

- Modelování explozí za použití metody Fin-Tree Billboard
- Zvuky pomocí Windows Multimedia Library (pouze Windows)

Zdrojový kód se dělí na pět částí

- Lesson30.cpp - Základní kód tutoriálu
- Image.cpp, Image.h - Nahrávání bitmap
- Tmatrix.cpp, Tmatrix.h - Třída pro práci s maticemi
- Tray.cpp, Tray.h - Třída pro práci s polopřímkami
- Tvector.cpp, Tvector.h - Třída pro práci s vektory

Třídy Vektor, Ray a Matrix jsou velmi užitečné. Dají se použít v jakémkoli projektu. Doporučuji je pečlivě prostudovat, třeba se vám budou někdy hodit.

Detekce kolizí

Polopřímka

Při detekci kolizí využijeme algoritmus, který se většinou používá v trasování polopřímek (ray tracing). Vektorová reprezentace polopřímky je tvořena bodem, který označuje začátek a vektorem (obvykle normalizovaným) určujícím směr polopřímky. Rovnice polopřímky:

```
bod_na_poloprimce = začátek + t * směr
```

Číslo t nabývá hodnot od nuly do nekonečna. Dosadíme-li nulu získáme počáteční bod. U větších čísel dostaneme odpovídající body na polopřímce. Bod, počátek i směr jsou 3D vektory se složkami x , y , a z . Nyní můžeme použít tuto

reprezentaci polopřímky k výpočtu průsečíku s rovinou nebo válcem.

Průsečík polopřímky a roviny

Vektorová reprezentace roviny vypadá takto:

$$X_n \cdot X = d$$

X_n je normála roviny, X je bod na jejím povrchu a d je číslo, které určuje vzdálenost roviny od počátku souřadnicového systému.

Překl.: Pod operací "dot" se skrývá skalární součin dvou vektorů (dot product), který se vypočte součtem násobků jednotlivých x , y a z složek. Nechávám ho v původním znění, protože i ve zdrojových kódech budeme volat metodu dot ().

$$\text{Překl.: } P \cdot Q = P_x * Q_x + P_y * Q_y + P_z * Q_z$$

Abychom definovali rovinu, potřebujeme 3D bod a vektor, který je kolmý k rovině. Pokud vezmeme za 3D bod vektor (0, 0, 0) a pro normálu vektor (0, 1, 0), protíná rovina osy x a z . Pokud známe bod a normálu, dá se chybějící číslo d snadno dopočítat.

Pozn.: Vektorová reprezentace roviny je ekvivalentní více známé formě $Ax + By + Cz + D = 0$ (obecná rovnice roviny). Pro přepočítání dosadte za A , B , C složky normály a přiřadte $D = -d$.

Nyní máme dvě rovnice

$$\text{bod_na_polopřímce} = \text{začátek} + t * \text{směr}$$
$$X_n \cdot X = d$$

Pokud polopřímka protne rovinu v nějakém bodě, musí souřadnice průsečíků vyhovovat oběma rovnicím

$$X_n \cdot \text{bod_na_polopřímce} = d$$

Nebo

$$(X_n \cdot \text{začátek}) + t * (X_n \cdot \text{směr}) = d$$

Vyjádříme t

$$t = (d - X_n \cdot \text{začátek}) / (X_n \cdot \text{směr})$$

Dosadíme d

$$t = (X_n \cdot \text{bod_na_polopřímce} - X_n \cdot \text{začátek}) / (X_n \cdot \text{směr})$$

Vytkneme X_n

$$t = (X_n \cdot (\text{bod_na_polopřímce} - \text{začátek})) / (X_n \cdot \text{směr})$$

Číslo t reprezentuje vzdálenost od začátku polopřímky k průsečíku s rovinou ve směru polopřímky (ne na kolmici). Dosazením t do rovnice polopřímky získáme kolizní bod. Existuje několik speciálních situací. Pokud $X_n \cdot \text{směr} = 0$, budou tyto dva vektory navzájem kolmé. Polopřímka prochází rovnoběžně s rovinou a tudíž neexistuje kolizní bod. Pokud je t záporné, průsečík leží před počátečním bodem. Polopřímka nesměřuje k rovině, ale od ní. Opět žádný průsečík.

```
int TestIntersectionPlane(const Plane& plane, const TVector& position, const TVector&
direction, double& lamda, TVector& pNormal)
{
    double DotProduct = direction.dot(plane._Normal); // Skalární součin vektorů
    double l2; // Určuje kolizní bod

    if ((DotProduct < ZERO) && (DotProduct > -ZERO)) // Je polopřímka rovnoběžná s
        rovinou?
        return 0; // Bez průsečíku

    l2 = (plane._Normal.dot(plane._Position - position)) / DotProduct; // Dosazení do
        vzorce

    if (l2 < -ZERO) // Směřuje polopřímka od roviny?
        return 0; // Bez průsečíku

    pNormal = plane._Normal; // Normála roviny
    lamda = l2; // Kolizní bod

    return 1; // Průsečík existuje
```

```
}
```

Průsečík polopřímky a válce

Výpočet průsečíku polopřímky s nekonečně dlouhým válcem je mnohem komplikovanější než vysvětlení toho, proč se tím zde nebudeme zabývat. Na pozadí je příliš mnoho matematiky. Mým primárním záměrem je poskytnout a vysvětlit nástroje bez zabíhání do zbytečných detailů, které by stejně někteří nepochopili. Válec je tvořen polopřímkou, která reprezentuje jeho osu, a poloměrem podstavy. Pro detekci kolize se v tomto tutoriálu používá funkce `TestIntersectionCylinder()`, která vrací jedničku, pokud byl nalezen průsečík, jinak nulu.

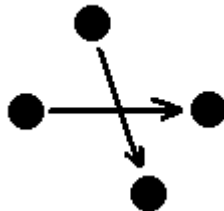
```
int TestIntersectionCylinder(const Cylinder& cylinder, const TVector& position, const TVector& direction, double& lambda, TVector& pNormal, TVector& newPosition)
```

V parametrech se předává struktura válce, začátek a směrový vektor polopřímky. Kromě návratové hodnoty získáme z funkce vzdálenost od průsečíku (na polopřímce), normálu vycházející z průsečíku a bod průsečíku.

Kolize mezi dvěma pohybujícími se koulemi

Koule je v geometrii reprezentována středem a poloměrem. Zjištění, jestli do sebe dvě koule narazily je banální. Vypočteme vzdálenost mezi dvěma středy a porovnáme ji se součtem poloměrů. Jak snadné!

Problémy nastanou při hledání kolizního bodu dvou POHYBUJÍCÍCH se koulí. Na obrázku je příklad, kdy se dvě koule přesunou z jednoho místa do druhého za určitý časový úsek. Jejich dráhy se protínají, ale to není dostatečný důvod k tvrzení, že do sebe opravdu narazí. Mohou se například pohybovat rozdílnou rychlostí. Jedna téměř stojí a druhá je za okamžik úplně někde jinde.



V předchozích metodách jsme řešili rovnice dvou geometrických objektů. Pokud podobná rovnice pro daný objekt neexistuje nebo nemůže být použita (pohyb po složité křivce), používá se jiná metoda. V našem případě známe počáteční i koncové body přesunu, časový krok, rychlost (+směr) a metodu zjištění nárazu statických koulí. Rozkouskujeme časový úsek na malé části. Koule budeme v závislosti na nich postupně posunovat a pokaždé testovat kolizi. Pokud najdeme některý bod, kdy je vzdálenost koulí menší než součet jejich poloměrů, vezmeme minulou pozici a označíme ji jako kolizní bod. Může se ještě začít interpolovat mezi těmito dvěma body na rozhraní, kdy kolize ještě nebyla a už je, abychom našli úplně přesnou pozici, ale většinou to není potřeba.

Čím menší bude časový úsek, tím budou části vzniklé rozsekáním menší a metoda bude více přesná (a více náročná na hardware počítače). Například, pokud bude časový úsek 1 a části 3, budeme zjišťovat kolizi v časech 0, 0,33, 0,66 a 1. V následujícím výpisu kódu hledáme koule, které během následujícího časového kroku narazí do kterékoli z ostatních. Funkce vrátí indexy obou koulí, bod a čas nárazu.

```
int FindBallCol(TVector& point, double& TimePoint, double Time2, int& BallNr1, int& BallNr2)
```

```
{
    TVector RelativeV;// Relativní rychlost mezi koulemi
    TRay rays;// Polopřímka

    double MyTime = 0.0;// Hledání přesné pozice nárazu
    double Add = Time2 / 150.0;// Rozkouskuje časový úsek na 150 částí
    double Timedummy = 10000;// Čas nárazu

    TVector posi;// Pozice na polopřímce

    // Test všech koulí proti všem ostatním po 150 krocích
    for (int i = 0; i < NrOfBalls - 1; i++)// Všechny koule
    {
        for (int j = i + 1; j < NrOfBalls; j++)// Všechny zbývající koule
        {
            // Výpočet vzdálenosti
            RelativeV = ArrayVel[i] - ArrayVel[j];// Relativní rychlost mezi koulemi
            rays = TRay(OldPos[i], TVector::unit(RelativeV));// Polopřímka
```

```

if ((rays.dist(OldPos[j])) > 40)// Je vzdálenost větší než 2 poloměry?
{
    continue;// Další
}

// Náráz

MyTime = 0.0;// Inicializace před vstupem do cyklu

while (MyTime < Time2)// Přesný bod nárazu
{
    MyTime += Add;// Zvětší čas
    posi = OldPos[i] + RelativeV * MyTime;//Přesun na další bod (pohyb na
    polopřímce)

    if (posi.dist(OldPos[j]) <= 40)// Náráz
    {
        point = posi;// Bod nárazu

        if (Timedummy > (MyTime - Add))// Bližší náráz, než který jsme už
        našli (v čase)?
        {
            Timedummy = MyTime - Add;// Přiřadit čas nárazu
        }

        BallNr1 = i;// Označení koulí, které narazily
        BallNr2 = j;
        break;// Ukončí vnitřní cyklus
    }
}

}

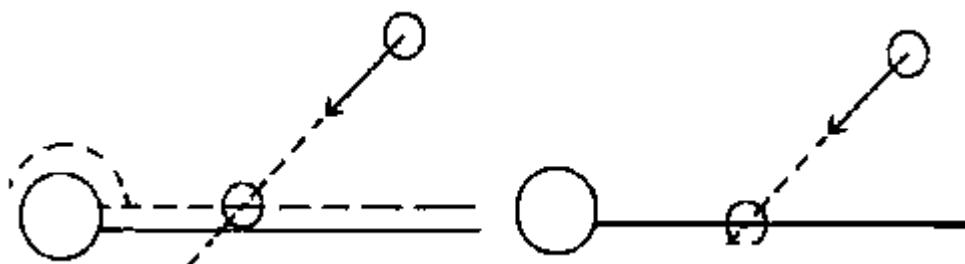
if (Timedummy != 10000)// Našli jsme kolizi?
{
    TimePoint = Timedummy;// Čas nárazu
    return 1;// Úspěch
}

return 0;// Bez kolize
}

```

Kolize mezi koulí a rovinou nebo válcem

Nyní už umíme zjistit průsečík polopřímky a roviny/válce. Tyto znalosti použijeme pro hledání kolizí mezi koulí a jedním z těchto objektů. Potřebujeme najít přesný bod nárazu. Převod znalostí z polopřímky na pohybující se kouli je relativně snadný. Podívejte se na levý obrázek, možná, že podstatu pochopíte sami.



Každá koule sice má poloměr, ale my ji budeme brát jako bezrozměrnou část, která má pouze pozici. K povrchu tělesa přičteme ve směru normálového vektoru offset určený poloměrem koule. Neboli k poloměru válce přičteme průměr koule (2 poloměry; z každé strany jeden). Operací jsme se vrátili k detekci kolize polopřímka - válec. Rovina je ještě jednodušší. Posuneme ji směrem ke kouli o její poloměr. Na obrázku jsou čárkovaně nakresleny "virtuální" objekty pro testy kolizí a plně objekty, které program vykreslí. Kdybychom k objektům při testech nepřipočítávali offset, koule by před odrazem z poloviny pronikaly do objektů (obrázek vpravo).

Máme-li určit místo nárazu, je vhodné nejdříve zjistit, jestli kolize nastane při aktuálním časovém úseku. Protože polopřímka má nekonečnou délku, je vždy možné, že se kolizní bod nachází až někde za novou pozici koule. Abychom to zjistili, spočítáme novou pozici a určíme vzdálenost mezi počátečním a koncovým bodem. Pokud je tato vzdálenost

kratší než vzdálenost, o kterou se objekt posune, tak máme jistotu, že kolize nastane v tomto časovém úseku. Abychom spočítali přesný čas kolize použijeme následující jednoduchou rovnici. Dst představuje vzdálenost mezi počátečním a koncovým bodem, Dsc vzdálenost mezi počátečním a kolizním bodem a časový krok je definován jako T. Řešením získáme čas kolize Tc.

$$T_c = D_{sc} * T / D_{st}$$

Výpočet se provede samozřejmě jenom tehdy, když má kolize nastat v tomto časovém kroku. Vracený čas je zlomkem (částí) celého časového kroku. Pokud bude časový krok 1 s a my nalezneme kolizní bod přesně uprostřed vzdálenosti, čas kolize se bude rovnat 0,5 s. Je interpretován jako: V časovém okamžiku 0,5 sekund po začátku přesunu do sebe objekty narazí. Kolizní bod se vypočte násobením času Tc aktuální rychlostí a přičtením počátečního bodu.

$$\text{bod_kolize} = \text{start} + \text{rychlost} * T_c$$

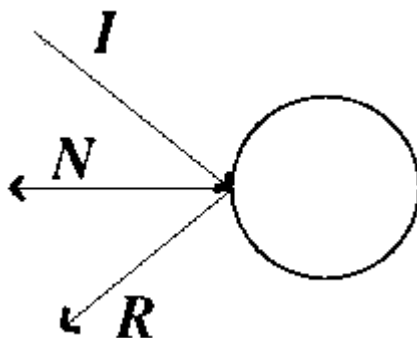
Tento kolizní bod je však na objektu s offsetem (pomocným). Abychom našli bod nárazu na reálném objektu, přičteme k bodu kolize invertovaný normálový vektor z bodu kolize, který má velikost poloměru koule. Normálový vektor získáme z funkce pro kolize. Všimněte si, že funkce pro kolizi s válcem vrací bod nárazu, takže nemusí být znovu počítán.

Modelování založené na fyzice

Reakce na náraz

Ošetření toho, jak se koule zachová po nárazu je stejně důležité jako samotné nalezení kolizního bodu. Použité algoritmy a funkce popisují přesný bod nárazu, normálový vektor vycházející z objektů v místě nárazu a časový úsek, ve kterém kolize nastala.

Při odrazech nám pomohou fyzikální zákony. Implementujeme poučku: "Úhel dopadu se rovná úhlu odrazu". Oba úhly se vztahují k normálovému vektoru, který vychází z objektu v kolizním bodě. Následující obrázek ukazuje odraz polopřímky od koule.



I je směrový vektor před nárazem, N je normálový vektor v bodě kolize a R je směrový vektor po odrazu, který se vypočte podle následující rovnice:

$$R = 2 * (-I \cdot N) * N + I$$

Omezení spočívá v tom, že I i N musí být jednotkové vektory. U nás však délka vektoru reprezentuje rychlost a směr koule, a proto nemůže být bez transformace dosazen do rovnice. Potřebujeme z něj vyjmout rychlost. Nalezneme jeho velikost a vydělíme jí jednotlivé x, y, z složky. Získaný jednotkový vektor dosadíme do rovnice a vypočteme R. Jsme skoro u konce. Vektor nyní míří ve směru odražené polopřímky, ale nemá původní délku. Minule jsme dělili, takže teď budeme násobit.

Následující výpis kódu se používá pro výpočet odrazu po kolizi koule s rovinou nebo válcem. Uvedený algoritmus pracuje i s jinými povrchy, nezáleží na jejich tvaru. Pokud nalezneme bod kolize a normálu, je odraz vždy stejný.

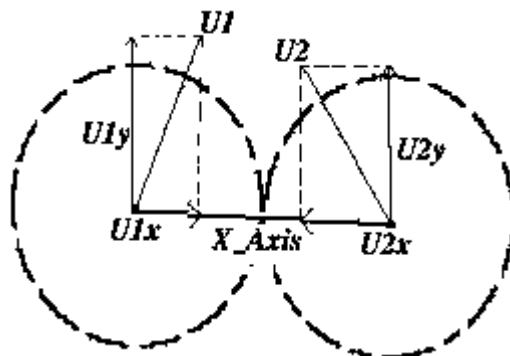
```
rt2 = ArrayVel[BallNr].mag();// Uloží délku vektoru
ArrayVel[BallNr].unit();// Normalizace vektoru

// Výpočet odrazu
ArrayVel[BallNr] = TVector::unit((normal * (2 * normal.dot(-ArrayVel[BallNr]))) +
ArrayVel[BallNr]);
ArrayVel[BallNr] = ArrayVel[BallNr] * rt2;// Nastavení původní délky
```

Když se koule srazí s jinou

Ošetření vzájemného nárazu dvou pohybujících se koulí je mnohem obtížnější. Musí být vyřešeny složité rovnice.

Nebudeme nic odvozovat, pouze vysvětlím výsledek. Situace při kolizi dvou koulí vypadá přibližně takto:



Vektory U_1 a U_2 představují rychlost koulí v čase nárazu. Středy dohromady spojuje osa X_Axis , na které leží vektory U_{1x} a U_{2x} , což jsou vlastně průměty rychlosti. U_{1y} a U_{2y} jsou projekce rychlosti na osu, která je kolmá k X_Axis . K jejich výpočtu postačí jednoduchý skalární součin.

Do následujících rovnic dosazujeme ještě čísla M_1 a M_2 , která vyjadřují hmotnost koulí. Snažíme se vypočítat orientaci vektorů rychlosti U_1 a U_2 po odrazu. Budou je vyjadřovat nové vektory V_1 a V_2 . Čísla V_{1x} , V_{1y} , V_{2x} , V_{2y} jsou opět průměty.

a) najít X_Axis

```
X_Axis = (střed2 - střed1)
Jednotkový vektor, X_Axis.unit();
```

b) najít projekce

```
U1x = X_Axis * (X_Axis dot U1)
U1y = U1 - U1x
U2x = -X_Axis * (-X_Axis dot U2)
U2y = U2 - U2x
```

c) najít nové rychlosti

```
V1x = ((U1x * M1) + (U2x * M2) - (U1x - U2x) * M2) / (M1 + M2)
V2x = ((U1x * M1) + (U2x * M2) - (U2x - U1x) * M1) / (M1 + M2)
```

V naší aplikaci nastavujeme jednotkovou hmotnost ($M_1 = M_2 = 1$), a proto se výpočet výsledných vektorů velmi zjednoduší.

d) najít konečné rychlosti

```
V1y = U1y
V2y = U2y

V1 = V1x + V1y
V2 = V2x + V2y
```

Odvození rovnic stálo hodně práce, ale jakmile se nacházejí v této formě, je jejich použití docela snadné. Kód, které vykonává srážky dvou koulí vypadá takto:

```
TVector pb1, pb2, xaxis, U1x, U1y, U2x, U2y, V1x, V1y, V2x, V2y; // Deklarace proměnných
double a, b;

pb1 = OldPos[BallColNr1] + ArrayVel[BallColNr1] * BallTime; // Nalezení pozice koule 1
pb2 = OldPos[BallColNr2] + ArrayVel[BallColNr2] * BallTime; // Nalezení pozice koule 2

xaxis = (pb2 - pb1).unit(); // Nalezení X_Axis

a = xaxis.dot(ArrayVel[BallColNr1]); // Nalezení projekce
U1x = xaxis * a; // Nalezení průmětů vektorů
U1y = ArrayVel[BallColNr1] - U1x;

xaxis = (pb1 - pb2).unit();

b = xaxis.dot(ArrayVel[BallColNr2]); // To samé pro druhou kouli
U2x = xaxis * b;
U2y = ArrayVel[BallColNr2] - U2x;
```

```

V1x = (U1x + U2x - (U1x - U2x)) * 0.5; // Nalezení nových rychlostí
V2x = (U1x + U2x - (U2x - U1x)) * 0.5;
V1y = U1y;
V2y = U2y;

for (j = 0; j < NrOfBalls; j++) // Posun všech koulí do času nárazu
{
    ArrayPos[j] = OldPos[j] + ArrayVel[j] * BallTime;
}

ArrayVel[BallColNr1] = V1x + V1y; // Nastavení právě vypočítaných vektorů koulím, které
do sebe narazily
ArrayVel[BallColNr2] = V2x + V2y;

```

Pohyb v gravitaci za použití Eulerových rovnic

Pro simulaci realistických pohybů nejsou nárazy, hledání kolizních bodů a odrazy dostatečné. Musí být přidán ještě pohyb podle fyzikálních zákonů. Asi nejpoužívanější metodou jsou Eulerovy rovnice. Všechny výpočty se vykonávají pro určitý časový úsek. To znamená, že se celá simulace neposouvá vpřed plynule, ale po určitých skocích. Představte si, že máte fotoaparát a každou vteřinu výslednou scénu vyfotíte. Během této vteřiny se provedou všechny pohyby, testy kolizí a odrazy. Výsledný obrázek se zobrazí na monitoru a zůstane tam až do další vteřiny. Opět stejné výpočty a další zobrazení. Takto pracují všechny počítačové animace, ale mnohem rychleji. Oko, stejně jako u filmu, vidí plynulý pohyb. V závislosti na Eulerových rovnicích se rychlost a pozice v každém časovém kroku změní takto:

```

nová_rychlost = stará_rychlost + zrychlení * časový úsek
nová_pozice = stará_pozice + nová_rychlost * časový úsek

```

Nyní se objekty pohybují a testují na kolize s použitím nové rychlosti. Zrychlení objektu je získáno vydělením síly, která na něj působí, jeho hmotností.

```

zrychlení = síla / hmotnost

```

V tomto demu je gravitace jediná síla, která působí na objekt. Může být reprezentována vektorem, který udává gravitační zrychlení. U nás se bude tento vektor rovnat (0; -0,5; 0). To znamená, že na začátku každého časového úseku spočítáme novou rychlost koule a s testováním kolizí ji posuneme. Pokud během časového úseku narazí (např. po 0,5 s), posuneme ji na pozici kolize, vypočteme odraz (nový vektor rychlosti) a přesuneme ji o zbývající čas (0,5 s). V něm opět testujeme kolize atd. Opakujeme tak dlouho, dokud zbývá nějaký čas.

Pokud je přítomno více pohybujících se objektů, musí být nejprve testován každý z nich na nárazy do statických objektů. Uloží se časově nejbližší z nich. Potom se provedou testy nárazů mezi pohybujícími se objekty - každý s každým. Vrácený čas je porovnán s časem u testů se statickými objekty a v úvahu je brán nejbližší náraz. Celá simulace se posune do tohoto času. Vypočte se odraz objektu a opět se provedou detekce nárazů do statických objektů atd. atd. - dokud zbývá nějaký čas. Překreslí se scéna a vše se opakuje nanovo.

Speciální efekty

Exploze

Kdykoli, když se objekty srazí, nastane exploze, která se zobrazí na souřadnicích průsečíku. Velmi jednoduchou cestou je alfablending dvou polygonů, které jsou navzájem kolmé a jejich střed je na souřadnicích kolizního bodu. Oba polygony se postupně zvětšují a zprůhledňují. Alfa hodnota se zmenšuje z počáteční jedničky až na nulu. Díky Z bufferu může spousta alfablendovaných polygonů způsobovat problémy - navzájem se překrývají, a proto si půjčíme techniku používanou při renderingu částic. Abychom vše dělali správně, musíme polygony řadit od zadních po přední podle vzdálenosti od pozorovatele. Také vypneme zápis do Depth bufferu (ne čtení). Všimněte si, že omezujeme počet explozí na maximálně dvacet na jeden snímek. Nastane-li jich najednou více, pole se zaplní a další se nebudou brát v úvahu. Následuje kód, který aktualizuje a renderuje exploze.

```

glEnable(GL_BLEND); // Blending
glDepthMask(GL_FALSE); // Vypne zápis do depth bufferu
glBindTexture(GL_TEXTURE_2D, texture[1]); // Textura exploze

for(i = 0; i < 20; i++) // Prochází výbuchy
{
    if(ExplosionArray[i]._Alpha >= 0) // Je exploze vidět?
    {
        glPushMatrix(); // Záloha matice
        ExplosionArray[i]._Alpha -= 0.01f; // Aktualizace alfa hodnoty
        ExplosionArray[i]._Scale += 0.03f; // Aktualizace měřítko
    }
}

```

```

        glColor4f(1, 1, 0, ExplosionArray[i]._Alpha); // Žlutá barva s průhledností
        glScalef(ExplosionArray[i]._Scale, ExplosionArray[i]._Scale, ExplosionArray
        [i]._Scale); // Změna měřítka

        glTranslatef((float)ExplosionArray[i]._Position.X() / ExplosionArray
        [i]._Scale, (float)ExplosionArray[i]._Position.Y() / explosionArray
        [i]._Scale, (float)ExplosionArray[i]._Position.Z() / ExplosionArray
        [i]._Scale); // Přesun na pozici kolizního bodu, měřítko je offsetem

        glCallList(dlist); // Zavolá display list
        glPopMatrix(); // Obnova původní matice
    }
}

glDepthMask(GL_TRUE); // Obnova původních parametrů OpenGL
glDisable(GL_BLEND);
glDisable(GL_TEXTURE_2D);

```

Zvuky

Pro přehrávání zvuků se používá funkce PlaySound() z multimediální knihovny Windows - rychlá cesta, jak bez problémů přehrát .wav zvuk.

Vysvětlení kódu

Gratuluji... pokud stále čtete, úspěšně jste se prokousali dlouhou a náročnou teoretickou sekcí. Předtím, než si začnete hrát s demem, měl by být ještě vysvětlen zdrojový kód. Ze všeho nejdříve se ale půjdeme podívat na globální proměnné.

Vektory dir a pos reprezentují pozici a směr kamery, kterou v programu pohybujeme funkcí gluLookAt(). Pokud scéna není vykreslována v módu "sledování koule", otáčí se kolem osy y.

```

TVector dir; // Směr kamery
TVector pos(0, -50, 1000); // Pozice kamery
float camera_rotation = 0; // Rotace scény na ose y

```

Gravitace, která působí na koule.

```

TVector accel(0, -0.05, 0); // Gravitační zrychlení aplikované na koule

```

Pole, která ukládají novou a starou pozici všech koulí a jejich směr. Počet koulí je natvrdo nastaven na deset.

```

TVector ArrayVel[10]; // Rychlost koulí
TVector ArrayPos[10]; // Pozice koulí
TVector OldPos[10]; // Staré pozice koulí

```

Časový úsek pro simulaci.

```

double Time = 0.6; // Časový krok simulace

```

Pokud je tato proměnná v jedničce, změní se mód kamery tak, aby sledovala pohyby koule. Pro její umístění a nasměrování se použije pozice a směr koule s indexem 1, která tedy bude vždy v záběru.

```

int hook_toball1 = 0; // Sledovat kamerou kouli?

```

Následující struktury se popisují samy svým jménem. Budou ukládat data o rovinách, válcích a explozích.

```

struct Plane // Struktura roviny
{
    TVector _Position;
    TVector _Normal;
};

struct Cylinder // Struktura válce
{
    TVector _Position;
    TVector _Axis;
    double _Radius;
};

```

```

struct Explosion// Struktura exploze
{
    TVector _Position;
    float _Alpha;
    float _Scale;
};

```

Objekty struktur.

```

Plane pl1, pl2, pl3, pl4, pl5;// Pět rovin místnosti (bez stropu)
Cylinder cyl1, cyl2, cyl3;// Tři válce
Explosion ExplosionArray[20];// Dvacet explozí

```

Textury, display list, quadratic.

```

GLuint texture[4];// Čtyři textury
GLuint dlist;// Display list výbuchu
GLUQuadricObj *cylinder_obj;// Quadratic pro kreslení koulí a válců

```

Funkce pro kolize koulí se statickými objekty a mezi koulemi navzájem.

```

int TestIntersionPlane(const Plane& plane, const TVector& position, const TVector&
direction, double& lamda, TVector& pNormal);

int TestIntersionCylinder(const Cylinder& cylinder, const TVector& position, const
TVector& direction, double& lamda, TVector& pNormal, TVector& newposition);

int FindBallCol(TVector& point, double& TimePoint, double Time2, int& BallNr1, int&
BallNr2);

```

Loading textur, inicializace proměnných, logika simulace, renderování scény a inicializace OpenGL.

```

void LoadGLTextures();
void InitVars();
void idle();

int DrawGLScene(GLvoid);
int InitGL(GLvoid)

```

Pro informace o geometrických třídách vektoru, polopřímky a matice nahlédněte do zdrojových kódů. Jsou velmi užitečné a mohou být bez problémů využity ve vašich vlastních programech.

Nejdůležitější kroky simulace nejprve popíše pseudokódem.

```

while (časový úsek != 0)
{
    for (každá koule)
    {
        Výpočet nejbližší kolize s rovinami;
        Výpočet nejbližší kolize s válci;

        Uložit/nahradit "záznam" o kolizi, pokud je to do teď nejbližší kolize v čase;
    }

    Testy kolizí mezi pohybujícími se koulemi;

    Uložit/nahradit "záznam" o kolizi, pokud je to do teď nejbližší kolize v čase;

    if (nastala kolize?)
    {
        Přesun všech koulí do času nejbližší kolize;
        (Už máme vypočten bod, normálu a čas kolize.)
        Výpočet odrazu;

        časový úsek -= čas kolize;
    }
    else
    {
        Přesun všech koulí na konec časového úseku;
    }
}

```

Zdrojový kód založený na pseudokódu je na první pohled mnohem více náročný na čtení a hlavně pochopení, nicméně

v základu je jeho přesnou implementací.

```
void idle()// Simulační logika - kolize
{
    // Deklarace proměnných
    double rt, rt2, rt4, lamda = 10000;

    TVector norm, uveloc;
    TVector normal, point, time;

    double RestTime, BallTime;

    TVector Pos2;

    int BallNr = 0, dummy = 0, BallColNr1, BallColNr2;
    TVector Nc;

    if (!hook_toball1)// Pokud kamera nesleduje kouli
    {
        camera_rotation += 0.1f;// Pootočení scény

        if (camera_rotation > 360)// Ošetření přetečení
        {
            camera_rotation = 0;
        }
    }

    RestTime = Time;
    lamda = 1000;

    // Výpočet rychlostí všech koulí pro následující časový úsek (Eulerovy rovnice)
    for (int j = 0; j < NrOfBalls; j++)
    {
        ArrayVel[j] += accel * RestTime;
    }

    while (RestTime > ZERO)// Dokud neskončil časový úsek
    {
        lamda = 10000;// Inicializace na velmi vysokou hodnotu

        // Kolize všech koulí s rovinami a válci
        for (int i = 0; i < NrOfBalls; i++)// Všechny koule
        {
            // Výpočet nové pozice a vzdálenosti
            OldPos[i] = ArrayPos[i];
            TVector::unit(ArrayVel[i], uveloc);
            ArrayPos[i] = ArrayPos[i] + ArrayVel[i] * RestTime;
            rt2 = OldPos[i].dist(ArrayPos[i]);

            // Kolize koule s rovinou
            if (TestIntersionPlane(pl1, OldPos[i], uveloc, rt, norm))
            {
                // Čas nárazu
                rt4 = rt * RestTime / rt2;

                // Pokud je menší než některý z dříve nalezených nahradit ho
                if (rt4 <= lamda)
                {
                    if (rt4 <= RestTime + ZERO)
                    {
                        if (!(rt <= ZERO) && (uveloc.dot(norm) > ZERO))
                        {
                            normal = norm;
                            point = OldPos[i] + uveloc * rt;
                            lamda = rt4;
                            BallNr = i;
                        }
                    }
                }
            }

            if (TestIntersionPlane(pl2, OldPos[i], uveloc, rt, norm))
```

```

{
    // To samé jako minule, ale s jinou rovinou
}
if (TestIntersionPlane(pl3, OldPos[i], uveloc, rt, norm))
{
    // To samé jako minule, ale s jinou rovinou
}
if (TestIntersionPlane(pl4, OldPos[i], uveloc, rt, norm))
{
    // To samé jako minule, ale s jinou rovinou
}
if (TestIntersionPlane(pl5, OldPos[i], uveloc, rt, norm))
{
    // To samé jako minule, ale s jinou rovinou
}
// Kolize koule s válcem
if (TestIntersionCylinder(cyl1, OldPos[i], uveloc, rt, norm, Nc))
{
    rt4 = rt * RestTime / rt2;
    if (rt4 <= lamda)
    {
        if (rt4 <= RestTime + ZERO)
        {
            if (!(rt <= ZERO) && (uveloc.dot(norm) > ZERO))
            {
                normal = norm;
                point = Nc;
                lamda = rt4;
                BallNr = i;
            }
        }
    }
}
if (TestIntersionCylinder(cyl2, OldPos[i], uveloc, rt, norm, Nc))
{
    // To samé jako minule, ale s jiným válcem
}
if (TestIntersionCylinder(cyl3, OldPos[i], uveloc, rt, norm, Nc))
{
    // To samé jako minule, ale s jiným válcem
}
}
// Kolize mezi koulemi
if (FindBallCol(Pos2, BallTime, RestTime, BallColNr1, BallColNr2))
{
    if (sounds)// Jsou zapnuté zvuky?
    {
        PlaySound("Data/Explode.wav", NULL, SND_FILENAME | SND_ASYNC);
    }
    if ((lamda == 10000) || (lamda > BallTime))
    {
        RestTime = RestTime - BallTime;
        TVector pb1, pb2, xaxis, U1x, U1y, U2x, U2y, V1x, V1y, V2x, V2y;//
        Deklarace proměnných
        double a, b;
        pb1 = OldPos[BallColNr1] + ArrayVel[BallColNr1] * BallTime;// Nalezení
        pozice koule 1
        pb2 = OldPos[BallColNr2] + ArrayVel[BallColNr2] * BallTime;// Nalezení
        pozice koule 2
    }
}

```

```

xaxis = (pb2 - pb1).unit();// Nalezení X_Axis

a = xaxis.dot(ArrayVel[BallColNr1]);// Nalezení projekce
U1x = xaxis * a;// Nalezení průmětů vektorů
U1y = ArrayVel[BallColNr1] - U1x;

xaxis = (pb1 - pb2).unit();

b = xaxis.dot(ArrayVel[BallColNr2]);// To samé pro druhou kouli
U2x = xaxis * b;
U2y = ArrayVel[BallColNr2] - U2x;

V1x = (U1x + U2x - (U1x - U2x)) * 0.5;// Nalezení nových rychlostí
V2x = (U1x + U2x - (U2x - U1x)) * 0.5;
V1y = U1y;
V2y = U2y;

for (j = 0; j < NrOfBalls; j++)// Aktualizace pozic všech koulí
{
    ArrayPos[j] = OldPos[j] + ArrayVel[j] * BallTime;
}

ArrayVel[BallColNr1] = V1x + V1y;// Nastavení právě vypočítaných vektorů
koulím, které do sebe narazily
ArrayVel[BallColNr2] = V2x + V2y;

// Aktualizace pole explozí
for(j = 0; j < 20; j++)// Všechny exploze
{
    if (ExplosionArray[j]._Alpha <= 0)// Hledá volné místo
    {
        ExplosionArray[j]._Alpha = 1;// Neprůhledná
        ExplosionArray[j]._Position = ArrayPos[BallColNr1];// Pozice
        ExplosionArray[j]._Scale = 1;// Měřítko

        break;// Ukončit prohledávání
    }
}

continue;// Opakovat cyklus
}
}

// Konec testů kolizí
// Pokud se prošel celý časový úsek a byly vypočteny reakce koulí, které
narazily
if (lamda != 10000)
{
    RestTime -= lamda;// Odečtení času kolize od časového úseku

    for (j = 0; j < NrOfBalls; j++)
    {
        ArrayPos[j] = OldPos[j] + ArrayVel[j] * lamda;
    }

    rt2 = ArrayVel[BallNr].mag();
    ArrayVel[BallNr].unit();
    ArrayVel[BallNr] = TVector::unit((normal * (2 * normal.dot(-ArrayVel
[BallNr]))) + ArrayVel[BallNr]);
    ArrayVel[BallNr] = ArrayVel[BallNr] * rt2;

    // Aktualizace pole explozí
    for(j = 0; j < 20; j++)// Všechny exploze
    {
        if (ExplosionArray[j]._Alpha <= 0)// Hledá volné místo
        {
            ExplosionArray[j]._Alpha = 1;// Neprůhledná
            ExplosionArray[j]._Position = ArrayPos[BallColNr1];// Pozice
            ExplosionArray[j]._Scale = 1;// Měřítko

            break;// Ukončit prohledávání
        }
    }
}

```



```
        }
    }
}
else
{
    RestTime = 0; // Ukončení hlavního cyklu a vlastně i funkce
}
}
}
```

Jak jsem už napsal na začátku, předmět kolizí je velmi těžký a rozsáhlý, aby se dal popsat jen v jednom tutoriálu, přesto jste se naučili spoustu nových věcí. Můžete začít vytvářet vlastní působivá demo. Nyní, když chápete základy, budete lépe rozumět i cizím zdrojovým kódům, které vás zase posunou o kousek dál. Přeji hodně štěstí.

napsal: Dimitrios Christopoulos
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Informace o autorovi

V současné době pracuje jako softwarový inženýr virtuální reality Helénskému světa v Aténách/Řecko (www.fhw.gr). Ačkoli se narodil v Německu, studoval řeckou univerzitu Patras na bakaláře přírodních věd v počítačovém inženýrství a informatice. Je také držitelem MSc degree (titul Magistra přírodních věd) z univerzity Hull (Anglie) v počítačové grafice a virtuálním prostředí.

První krůčky s programováním podnikl v jazyce Basic na Commodoru 64. Po začátku studia se přeorientoval na C/C++/Assembler na platformě PC. Během několika minulých let si jako grafické API zvolil OpenGL.

Lekce 31 - Nahrávání a renderování modelů

Další skvělý tutoriál! Naučíte se, jak nahrát a zobrazit otexturovaný Milkshape3D model. Nezdá se to, ale asi nejvíce se budou hodit znalosti o práci s dynamickou pamětí a jejím kopírování z jednoho místa na druhé.

Zdrojový kód tohoto projektu byl vyjmut z PortaLib3D, knihovny, kterou jsem napsal, abych lidem umožnil zobrazovat modely za použití velmi malého množství dalšího kódu. Abyste se na ni mohli opravdu spolehnout musíte nejdříve vědět, co dělá a jak pracuje.

Část PortaLib3D, uvedená zde, si stále zachovává můj copyright. To neznamená, že ji nesmíte používat, ale že při vložení kódu do svého projektu musíte uvést náležitý credit. To je vše - žádné velké nároky. Pokud byste chtěli číst, pochopit a re-implementovat celý kód (žádné kopírovat vložit!), budete uvolněni ze své povinnosti. Pak je to váš výtvar. Pojdme se ale podívat na něco zajímavějšího.

Model, který používáme v tomto projektu, pochází z Milkshape3D. Je to opravdu kvalitní balík pro modelování, který zahrnuje vlastní file-formát. Mým dalším plánem je implementovat Anim8or (<http://www.anim8or.com/>), souborový reader. Je free a umí číst samozřejmě i 3DS. Nicméně formát souboru není tím hlavním pro loading modelů. Nejdříve se musí vytvořit vlastní struktury, které jsou schopny pojmout data.

První ze všeho deklarujeme obecnou třídu Model, která je kontejnerem pro všechna data.

```
class Model// Obecné úložiště dat (abstraktní třída)
{
public:
```

Ze všeho nejdůležitější jsou samozřejmě vertexy. Pole tří desetinných hodnot m_location reprezentuje jednotlivé x, y, z souřadnice. Proměnnou m_boneID budeme v tomto tutoriálu ignorovat. Její čas přijde až v dalším při kosterní animaci.

```
    struct Vertex// Struktura vertexu
    {
        float m_location[3];// X, y, z souřadnice
        char m_boneID;// Pro skeletální animaci
    };
```

Všechny vertexy potřebujeme seskupit do trojúhelníků. Pole m_vertexIndices obsahuje tři indexy do pole vertexů. Touto cestou bude každý vertex uložen v paměti pouze jednou. V polích m_s a m_t jsou texturové koordináty každého vrcholu. Poslední atribut definuje tři normálové vektory pro světlo.

```
    struct Triangle// Struktura trojúhelníku
    {
        int m_vertexIndices[3];// Tři indexy do pole vertexů
        float m_s[3], m_t[3];// Texturové koordináty
        float m_vertexNormals[3][3];// Tři normálové vektory
    };
```

Další struktura popisuje mesh modelu. Mesh je skupina trojúhelníků, na které je aplikován stejný materiál a textura. Skupiny meshů dohromady tvoří celý model. Stejně jako trojúhelníky obsahovaly pouze indexy na vertexy, budou i meshe obsahovat pouze indexy na trojúhelníky. Protože neznáme jejich přesný počet, musí být pole dynamické. Třetí proměnná je opět indexem, tentokrát do materiálů (textura, osvětlení).

```
    struct Mesh//Mesh modelu
    {
        int *m_pTriangleIndices;// Indexy do trojúhelníků
        int m_numTriangles;// Počet trojúhelníků
        int m_materialIndex;// Index do materiálů
    };
```

Ve struktuře Material jsou uloženy standardní koeficienty světla, ve stejném formátu jako používá OpenGL: okolní (ambient), rozptýlené (diffuse), odražené (specular), vyzářující (emissive) a lesklost (shininess). dále obsahuje objekt textury a souborovou cestu k textuře, aby mohla být znovu nahrána, když je ukončen kontext OpenGL.

```
    struct Material// Vlastnosti materiálů
    {
        float m_ambient[4], m_diffuse[4], m_specular[4], m_emissive[4];// Reakce
        materiálu na světlo
        float m_shininess;// Lesk materiálu
    };
```

```

        GLuint m_texture; // Textura
        char *m_pTextureFilename; // Souborová cesta k textuře
    };

```

Vytvoříme proměnné právě napsaných struktur ve formě ukazatelů na dynamická pole, jejichž paměť alokuje funkce pro loading objektů. Musíme samozřejmě ukládat i velikost polí.

```

protected:
    int m_numVertices; // Počet vertexů
    Vertex *m_pVertices; // Dynamické pole vertexů

    int m_numTriangles; // Počet trojúhelníků
    Triangle *m_pTriangles; // Dynamické pole trojúhelníků

    int m_numMeshes; // Počet meshů
    Mesh *m_pMeshes; // Dynamické pole meshů

    int m_numMaterials; // Počet materiálů
    Material *m_pMaterials; // Dynamické pole materiálů

```

A konečně metody třídy. Virtuální členská funkce loadModelData() má za úkol nahrát data ze souboru. Přidáme jí nulu, aby nemohl být vytvořen objekt třídy (abstraktní třída). Tato třída je zamýšlena pouze jako úložiště dat. Všechny operace pro nahrávání mají na starosti odvozené třídy, kdy každá z nich umí svůj vlastní formát souboru. Celá hierarchie je více obecná.

```

public:
    Model(); // Konstruktor
    virtual ~Model(); // Destruktor

    virtual bool loadModelData(const char *filename) = 0; // Loading objektu ze souboru

```

Metoda reloadTextures() slouží pro loading textur a jejich znovunahrávání, když se ztratí kontext OpenGL (např. při přepnutí z/do fullscreenu). Draw() vykresluje objekt. Tato funkce nemusí být virtuální, protože defakto známe všechny potřebné informace o struktuře objektu (vertexy, trojúhelníky...).

```

    void reloadTextures(); // Znovunahrání textur
    void draw(); // Vykreslení objektu
};

```

Od třídy Model podědíme třídu MilkshapeModel. Přepíšeme v ní metodu loadModelData().

```

class MilkshapeModel : public Model
{
public:
    MilkshapeModel(); // Konstruktor
    virtual ~MilkshapeModel(); // Destruktor

    virtual bool loadModelData(const char *filename); // Loading objektu ze souboru
};

```

Nyní nahrávání objektů. Přepíšeme virtuální funkci loadModelData() abstraktní třídy Model tak, aby ve třídě MilkShapeModel nahrávala data ze souboru ve formátu Milkshape3D. Předáváme jí řetězec se jménem souboru. Pokud vše proběhne v pořádku, funkce nastaví datové struktury a vrátí true.

```

bool MilkshapeModel::loadModelData(const char *filename)
{

```

Soubor otevřeme jako vstupní (ios::in), binární (ios::binary) a nebudeme ho vytvářet (ios::nocreate). Pokud nebyl nalezen vrátí funkce false, aby indikovala error.

```

    ifstream inputFile(filename, ios::in | ios::binary | ios::nocreate); // Otevření
    souboru

    if (inputFile.fail()) // Podařilo se ho otevřít?
        return false;

```

Zjistíme velikost souboru v bytech a potom ho celý načteme do pomocného bufferu pBuffer.

```

    // Velikost souboru
    inputFile.seekg(0, ios::end);
    long fileSize = inputFile.tellg();
    inputFile.seekg(0, ios::beg);

    byte *pBuffer = new byte[fileSize]; // Alokace paměti pro kopii souboru

```

```
inputFile.read(pBuffer, fileSize); // Vytvoření paměťové kopie souboru
inputFile.close(); // Zavření souboru
```

Deklarujeme pomocný ukazatel pPtr, který ihned inicializujeme tak, aby ukazoval na stejné místo jako pBuffer, tedy na začátek paměti. Do hlavičky souboru pHeader uložíme adresu hlavičky a zvětšíme adresu v pPtr o velikost hlavičky.

Pozn.: Strukturu hlavičky a jí podobné jsem na začátku tutoriálu neuváděl, protože je budeme používat jenom zde, v této funkci. Pokud vás přeci zajímají, stáhněte si zdrojový kód. Jsou deklarované nahoře v souboru MilkshapeModel.cpp.

```
const byte *pPtr = pBuffer; // Pomocný ukazatel na kopii souboru

MS3DHeader *pHeader = (MS3DHeader*)pPtr; // Ukazatel na hlavičku
pPtr += sizeof(MS3DHeader); // Posun za hlavičku
```

Hlavička přímo specifikuje formát souboru. Ujistíme se, že se jedná o platný formát, který umíme nahrát.

```
// Není Milkshape3D souborem
if (strncmp(pHeader->m_ID, "MS3D000000", 10) != 0)
{
    delete [] pBuffer; // Překl.: Smaže kopii souboru !!!!!
    return false;
}

// Špatná verze souboru, třída podporuje pouze verze 1.3 a 1.4
if (pHeader->m_version < 3 || pHeader->m_version > 4)
{
    delete [] pBuffer; // Překl.: Smaže kopii souboru !!!!!
    return false;
}
```

Načteme všechny vertexy. Nejdříve zjistíme jejich počet, alokujeme potřebnou paměť a přesuneme pPtr na další pozici. V cyklu procházíme jednotlivé vertexy. Nastavíme ukazatel pVertex na přetypovaný pPtr a definujeme m_boneID. Nakonec zavoláme memcpy() pro zkopírování hodnot a zvětšíme pPtr.

```
int nVertices = *(word*)pPtr; // Počet vertexů

m_numVertices = nVertices; // Nastaví atribut třídy
m_pVertices = new Vertex[nVertices]; // Alokace paměti pro vertexy

pPtr += sizeof(word); // Posun za počet vertexů

int i; // Pomocná proměnná

for (i = 0; i < nVertices; i++) // Nahrává vertexy
{
    MS3DVertex *pVertex = (MS3DVertex*)pPtr; // Ukazatel na vertex

    // Načtení vertexu
    m_pVertices[i].m_boneID = pVertex->m_boneID;
    memcpy(m_pVertices[i].m_location, pVertex->m_vertex, sizeof(float) * 3);

    pPtr += sizeof(MS3DVertex); // Posun za tento vertex
}
```

Stejně jako u vertexů, tak i trojúhelníků nejdříve provedeme potřebné operace pro alokaci paměti. V cyklu procházíme jednotlivé trojúhelníky a inicializujeme je. Všimněte si, že v souboru jsou indexy vertexů uloženy v poli word hodnot, ale v modelu kvůli konzistentnosti a jednoduchosti používáme datový typ int. Číslo se implicitně přetypuje.

```
int nTriangles = *(word*)pPtr; // Počet trojúhelníků
m_numTriangles = nTriangles; // Nastaví atribut třídy
m_pTriangles = new Triangle[nTriangles]; // Alokace paměti pro trojúhelníky

pPtr += sizeof(word); // Posun za počet trojúhelníků

for (i = 0; i < nTriangles; i++) // Načítá trojúhelníky
{
    MS3DTriangle *pTriangle = (MS3DTriangle*)pPtr; // Ukazatel na trojúhelník

    // Načtení trojúhelníku
    int vertexIndices[3] = { pTriangle->m_vertexIndices[0], pTriangle->
        >m_vertexIndices[1], pTriangle->m_vertexIndices[2] };
}
```

Všechna čísla v poli t jsou nastavena na 1.0 mínus originál. To proto, že OpenGL používá počátek texturovacího souřadnicového systému vlevo dole, narozdíl od Milkshape, které ho má vlevo nahoře. Odečtením od jedničky, y

souřadnici invertujeme. Vše ostatní by mělo být bez problémů.

```
float t[3] = { 1.0f-pTriangle->m_t[0], 1.0f-pTriangle->m_t[1], 1.0f-pTriangle->m_t[2] };

memcpy(m_pTriangles[i].m_vertexNormals, pTriangle->m_vertexNormals, sizeof
(float)*3*3);
memcpy(m_pTriangles[i].m_s, pTriangle->m_s, sizeof(float)*3);
memcpy(m_pTriangles[i].m_t, t, sizeof(float)*3);
memcpy(m_pTriangles[i].m_vertexIndices, vertexIndices, sizeof(int)*3);

pPtr += sizeof(MS3DTriangle); // Posun za tento trojúhelník
}
```

Nahrajeme struktury mesh. V Milkshape3D jsou také nazývány groups - skupiny. V každé se liší počet trojúhelníků, takže nemůžeme načíst žádnou standardní strukturu. Namísto toho budeme dynamicky alokovat paměť pro indexy trojúhelníků a v každém průchodu je načítat.

```
int nGroups = *(word*)pPtr; // Počet meshů
m_numMeshes = nGroups; // Nastaví atribut třídy
m_pMeshes = new Mesh[nGroups]; // Alokace paměti pro meshe

pPtr += sizeof(word); // Posun za počet meshů

for (i = 0; i < nGroups; i++) // Načítá meshe
{
    pPtr += sizeof(byte); // Posun za flagy
    pPtr += 32; // Posun za jméno

    word nTriangles = *(word*)pPtr; // Počet trojúhelníků v meshi
    pPtr += sizeof(word); // Posun za počet trojúhelníků

    int *pTriangleIndices = new int[nTriangles]; // Alokace paměti pro indexy
    trojúhelníků

    for (int j = 0; j < nTriangles; j++) // Načítá indexy trojúhelníků
    {
        pTriangleIndices[j] = *(word*)pPtr; // Přiřadí index trojúhelníku
        pPtr += sizeof(word); // Posun za index trojúhelníku
    }

    char materialIndex = *(char*)pPtr; // Načte index materiálu

    pPtr += sizeof(char); // Posun za index materiálu

    m_pMeshes[i].m_materialIndex = materialIndex; // Index materiálu
    m_pMeshes[i].m_numTriangles = nTriangles; // Počet trojúhelníků
    m_pMeshes[i].m_pTriangleIndices = pTriangleIndices; // Indexy trojúhelníků
}
```

Poslední, co načítáme jsou informace o materiálech.

```
int nMaterials = *(word*)pPtr; // Počet materiálů
m_numMaterials = nMaterials; // Nastaví atribut třídy
m_pMaterials = new Material[nMaterials]; // Alokace paměti pro materiály

pPtr += sizeof(word); // Posun za počet materiálů

for (i = 0; i < nMaterials; i++) // Prochází materiály
{
    MS3DMaterial *pMaterial = (MS3DMaterial*)pPtr; // Ukazatel na materiál

    // Načte materiál
    memcpy(m_pMaterials[i].m_ambient, pMaterial->m_ambient, sizeof(float)*4);
    memcpy(m_pMaterials[i].m_diffuse, pMaterial->m_diffuse, sizeof(float)*4);
    memcpy(m_pMaterials[i].m_specular, pMaterial->m_specular, sizeof(float)*4);
    memcpy(m_pMaterials[i].m_emissive, pMaterial->m_emissive, sizeof(float)*4);
    m_pMaterials[i].m_shininess = pMaterial->m_shininess;
}
```

Alokujeme paměť pro řetězec jména souboru textury a zkopírujeme ho.

```
// Alokace pro jméno souboru textury
m_pMaterials[i].m_pTextureFilename = new char[strlen(pMaterial->m_texture)+1];
```

```

    // Zkopírování jména souboru
    strcpy(m_pMaterials[i].m_pTextureFilename, pMaterial->m_texture);

    // Posun za materiál
    pPtr += sizeof(MS3DMaterial);
}

```

Nakonec loadujeme textury objektu, uvolníme paměť kopie souboru a vrátíme true, abychom oznámili úspěch celé akce.

```

    reloadTextures();// Nahraje textury
    delete [] pBuffer;// Smaže kopii souboru
    return true;// Model byl nahrán
}

```

Nyní jsou členské proměnné třídy Model vyplněné. Zbývá ještě nahrát textury. V cyklu procházíme všechny materiály a testujeme, jestli je řetězec se jménem textury delší než nula. Pokud ano nahrajeme texturu pomocí standardní NeHe funkce. Pokud ne přiřadíme textuře nulu jako indikaci, že neexistuje.

```

void Model::reloadTextures()// Nahrání textur
{
    for (int i = 0; i < m_numMaterials; i++)// Jednotlivé materiály
    {
        if (strlen(m_pMaterials[i].m_pTextureFilename) > 0)// Existuje řetězec s cestou
        {
            // Nahraje texturu
            m_pMaterials[i].m_texture = LoadGLTexture(m_pMaterials
                [i].m_pTextureFilename);
        }
        else
        {
            // Nulou indikuje, že materiál nemá texturu
            m_pMaterials[i].m_texture = 0;
        }
    }
}

```

Můžeme začít vykreslovat model. Díky uspořádání do struktur to není nic složitého. Ze všeho nejdříve uložíme atribut, jestli je zapnuté nebo vypnuté texturování. Na konci funkce ho budeme moci obnovit.

```

void Model::draw()
{
    GLboolean texEnabled = glIsEnabled(GL_TEXTURE_2D);// Uloží atribut
}

```

Každý mesh renderujeme samostatně, protože mesh seskupuje všechny trojúhelníky se stejnými vlastnostmi. Stačí jedno hromadné nastavení OpenGL pro velkou skupinu polygonů, namísto mnohem méně efektivnímu: nastavit vlastnosti pro trojúhelník - vykreslit trojúhelník. S meshi postupujeme takto: nastavit vlastnosti - vykreslit všechny trojúhelníky s těmito vlastnostmi.

```

    for (int i = 0; i < m_numMeshes; i++)// Meshe
    {

```

M_pMeshes[i] použijeme jako referenci na aktuální mesh. Každý z nich má vlastní materiálové vlastnosti, podle kterých nastavíme OpenGL. Pokud se materialIndex rovná -1, znamená to, že mesh není definován. V takovém případě zůstaneme u implicitních nastavení OpenGL. Texturu zvolíme a zapneme pouze tehdy, pokud je větší než nula. Při jejím loadingu jsme nadefinovali, že pokud neexistuje nastavíme ji na nulu. Vypnutí texturingu je tedy logickým krokem. Pokud materiál meshe neexistuje, texturování také vypneme, protože nemáme kde vzít texturu.

```

        int materialIndex = m_pMeshes[i].m_materialIndex;// Index

        if (materialIndex >= 0)// Obsahuje mesh index materiálu?
        {
            // Nastaví OpenGL
            glMaterialfv(GL_FRONT, GL_AMBIENT, m_pMaterials[materialIndex].m_ambient);
            glMaterialfv(GL_FRONT, GL_DIFFUSE, m_pMaterials[materialIndex].m_diffuse);
            glMaterialfv(GL_FRONT, GL_SPECULAR, m_pMaterials[materialIndex].m_specular);
            glMaterialfv(GL_FRONT, GL_EMISSION, m_pMaterials[materialIndex].m_emissive);
            glMaterialf(GL_FRONT, GL_SHININESS, m_pMaterials
                [materialIndex].m_shininess);

            if (m_pMaterials[materialIndex].m_texture > 0)// Obsahuje materiál texturu?

```

```

    {
        glBindTexture(GL_TEXTURE_2D, m_pMaterials[materialIndex].m_texture);
        glEnable(GL_TEXTURE_2D);
    }
    else// Bez textury
    {
        glDisable(GL_TEXTURE_2D);
    }
}
else// Bez materiálu nemůže být ani textura
{
    glDisable(GL_TEXTURE_2D);
}
}

```

Při vykreslování procházíme nejdříve všechny trojúhelníky meshe a potom každý z jeho vrcholů. Specifikujeme normálový vektor a texturové koordináty.

```

glBegin(GL_TRIANGLES); // Začátek trojúhelníků
{
    for (int j = 0; j < m_pMeshes[i].m_numTriangles; j++) // Trojúhelníky v meshi
    {
        int triangleIndex = m_pMeshes[i].m_pTriangleIndices[j]; // Index
        const Triangle* pTri = &m_pTriangles[triangleIndex]; // Trojúhelník
        for (int k = 0; k < 3; k++) // Vertexy v trojúhelníku
        {
            int index = pTri->m_vertexIndices[k]; // Index vertexu

            glNormal3fv(pTri->m_vertexNormals[k]); // Normála
            glTexCoord2f(pTri->m_s[k], pTri->m_t[k]); // Texturovací souřadnice
            glVertex3fv(m_pVertices[index].m_location); // Souřadnice vertexu
        }
    }
}
glEnd(); // Konec kreslení
}

```

Obnovíme atribut OpenGL.

```

// Obnovení nastavení OpenGL
if (texEnabled)
{
    glEnable(GL_TEXTURE_2D);
}
else
{
    glDisable(GL_TEXTURE_2D);
}
}

```

Jediným dalším kódem ve třídě Model, který stojí za pozornost je konstruktor a destruktory. Konstruktor inicializuje všechny členské proměnné na nulu nebo v případě ukazatelů na NULL. Mějte na paměti, že pokud zavoláte funkci loadModelData() dvakrát pro jeden objekt, nastanou úniky paměti! Paměť se totiž uvolňuje až v destruktory.

```

Model::Model() // Konstruktor
{
    m_numMeshes = 0;
    m_pMeshes = NULL;

    m_numMaterials = 0;
    m_pMaterials = NULL;

    m_numTriangles = 0;
    m_pTriangles = NULL;

    m_numVertices = 0;
    m_pVertices = NULL;
}

Model::~Model() // Destruktor
{

```

```

int i;

for (i = 0; i < m_numMeshes; i++)
{
    delete[] m_pMeshes[i].m_pTriangleIndices;
}
for (i = 0; i < m_numMaterials; i++)
{
    delete[] m_pMaterials[i].m_pTextureFilename;
}

m_numMeshes = 0;

if (m_pMeshes != NULL)
{
    delete[] m_pMeshes;
    m_pMeshes = NULL;
}

m_numMaterials = 0;

if (m_pMaterials != NULL)
{
    delete[] m_pMaterials;
    m_pMaterials = NULL;
}

m_numTriangles = 0;

if (m_pTriangles != NULL)
{
    delete[] m_pTriangles;
    m_pTriangles = NULL;
}

m_numVertices = 0;

if (m_pVertices != NULL)
{
    delete[] m_pVertices;
    m_pVertices = NULL;
}
}

```

Vysvětlili jsme si třídu Model, zbytek už bude velice jednoduchý. Nahoře v souboru Lesson32.cpp deklarujeme ukazatel na model a inicializujeme ho na NULL.

```
Model *pModel = NULL; // Ukazatel na model
```

Jeho data nahrajeme až ve funkci WinMain(). Loading NIKDY nevkládejte do InitGL(), protože se volá vždycky, když uživatel změní mód fullscreen/okno. Při této akci se ztrácí a znovu vytváří OpenGL kontext, ale data modelu se nemusí (a kvůli unikům paměti dokonce nesmí) reloadovat. Zůstávají nedotčená. Stačí znovu nahrát textury, které jsou na OpenGL závislé. Je-li ve scéně více modelů, musí se reloadTextures() volat zvlášť pro každý objekt třídy. Pokud se stane, že budou modely najednou bílé, znamená to, že se textury nenahrály správně.

```

// Začátek funkce WinMain()
pModel = new MilkshapeModel(); // Alokace paměti pro model

if (pModel->loadModelData("data/model.ms3d") == false) // Pokusí se nahrát model
{
    MessageBox(NULL, "Couldn't load the model data\model.ms3d", "Error", MB_OK |
        MB_ICONERROR);
    return 0; // Model se nepodařilo nahrát - program se ukončí
}

// Začátek funkce InitGL()
pModel->reloadTextures(); // Nahrání textur modelu

```

Poslední, co popíšeme je DrawGLScene(). Namísto klasických glTranslatef() a glRotatef() použijeme funkci gluLookAt(). Prvními třemi parametry umísťuje kameru na pozici, prostřední tři souřadnice určují střed scény a poslední tři definují vektor směřující vzhůru. V našem případě se díváme z bodu (75, 75, 75) na bod (0, 0, 0). Model tedy bude vykreslen kolem souřadnic (0, 0, 0), pokud před kreslením neprovedeme translaci. Osa y směřuje vzhůru. Aby se gluLookAt() chovala tímto způsobem, musí být volána jako první po glLoadIdentity().


```
int DrawGLScene(GLvoid) // Rendering scény
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže buffery
    glLoadIdentity(); // Reset matice

    gluLookAt(75,75,75, 0,0,0, 0,1,0); // Přesun kamery
```

Aby byl výsledek trochu zajímavější rotujeme modelem kolem osy y.

```
    glRotatef(yrot, 0.0f, 1.0f, 0.0f); // Rotace na ose y
```

Pro rendering modelu použijeme jeho vlastní funkce. Vykreslí se vycentrovaný okolo středu, ale pouze tehdy, že i v Milkshape 3D byl modelován okolo středu. Pokus s ním budete chtít rotovat, posunovat nebo měnit velikost, zavolejte odpovídající OpenGL funkce. Pro otestování si zkuste vytvořit vlastní model a nahrajte ho do programu. Funguje?

```
    pModel->draw(); // Rendering modelu

    yrot += 1.0f; // Otáčení scény
    return TRUE;
}
```

A co dál? Plánuji další tutoriál pro NeHe, ve kterém rozšíříme třídu tak, aby umožňovala animaci objektu pomocí jeho kostry (skeletal animation). Možná také naprogramuji další třídy loaderů - program bude schopen nahrát více různých formátů. Krok ke skeletální animaci není až zase tak velký, jak se může zdát, ačkoli matematika bude o stupeň složitější. Pokud ještě nerozumíte maticím a vektorům, je čas se na ně trochu podívat.

napsal: Brett Porter
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Informace o autorovi

Brett Porter se narodil v Austrálii, studoval na Wollongongské Univerzitě. Nedávno absolvoval na BCompSc A BMath (BSc - bakalář přírodních věd). Programovat začal před dvanácti lety v Basicu na "klonu" Commodore 64 zvaném VZ300, ale brzy přešel na Pascal, Intel Assembler, C++ a Javu. Před několika lety začal používat OpenGL.

Lekce 32 - Picking, alfa blending, alfa testing, sorting

V tomto tutoriálu se pokusím zodpovědět několik otázek, na které jsem denně dotazován. Chcete vědět, jak při kliknutí tlačítkem myši identifikovat OpenGL objekt nacházející se pod kurzorem (picking). Dále byste se chtěli dozvědět, jak vykreslit objekt bez zobrazení určité barvy (alfa blending, alfa testing). Třetí věcí, se kterou si nevíte rady, je, jak řadit objekty, aby se při blendingu správně zobrazily (sorting). Naprogramujeme hru, na které si vše vysvětlíme.

Vítejte do 32. lekce. Je asi nejdelší, jakou jsem kdy napsal - přes 1000 řádků kódu a více než 1500 řádků HTML. Také je prvním, který používá nový NeHeGL základní kód. Tutoriál zabral hodně času, ale myslím si, že stojí za to. Probírá se v něm především: alfa blending, alfa testing, čtení zpráv myši, současné používání perspektivní i pravouhlé projekce, zobrazování kurzoru myši pomocí OpenGL, ruční řazení objektů podle hloubky, snímky animace z jedné textury a to nejdůležitější: naučíte se vše o pickingu.

V první verzi program zobrazoval tři polygony, které po kliknutí měnily barvu. Jak vzrušující! Tak, jako vždycky, chci zapůsobit super cool tutoriálem. Nejen, že jsou v něm zahrnuty všechny informace k probíranému tématu, ale samozřejmě musí být také hezký na pohled. Dokonce i tehdy, pokud neprogramujete, vás může zaujmout - kompletní hra. Objekty se sestřelují tak dlouho, dokud vám neochabne ruka držící myš, takže už nejste schopni stisknout tlačítko.

Poznámka ohledně kódu: Budu vysvětlovat pouze lesson33.cpp. V NeHeGL jsou změny především v podpoře myši ve funkci WindowProc(). Také už nebudu vysvětlovat loading textur, vytváření display listů fontu a výstup textu. Vše bylo vysvětleno v minulých tutoriálech.

Textury, používané v tomto programu, byly nakresleny v Adobe Photoshopu. Každý z .TGA obrázků má barevnou hloubku 32 bitů na pixel, obsahuje tedy alfa kanál. Pokud si nejste jisti, jak ho přidat, kupte si nějakou knihu, prozkoumejte internet nebo zkuste help. Postup je podobný vytváření masky v tutoriálu o maskingu, nahrajte svůj obrázek do Adobe Photoshopu nebo jakéhokoli grafického editoru s podporou alfa kanálu. Proveďte výběr barvy, abyste označili oblast okolo objektu, zkopírujte výběr a vložte ho do nového obrázku. Negujte obrázek, takže oblast, kde by měl být, bude černá. Změňte okolí na bílé, vyberte celý obrázek a zkopírujte ho. Vraťte se na originál a vytvořte alfa kanál, do kterého vložte masku. Uložte obrázek jako 32 bitový .TGA soubor. Ujistěte se, že je zaškrtnuto Uchovat průhlednost a ukládejte bez komprese.

Zjistíme, jestli je definovaná symbolická konstanta CDS_FULLSCREEN a pokud ne, nadefinujeme ji na hodnotu 4. Pro ty z vás, kteří se úplně ztratili... některé kompilátory nepřijímají CDS_FULLSCREEN hodnotu. Pokud ji pak v programu použijeme, kompilace skončí s chybovou zprávou. Abychom tomuto předešli, tak ji v případě potřeby nadefinujeme ručně.

```
#ifndef CDS_FULLSCREEN// Některé kompilátory nedefinují CDS_FULLSCREEN
#define CDS_FULLSCREEN 4// Ruční nadefinování
#endif
```

Deklarujeme funkci DrawTargets(), potom proměnnou okna a kláves.

```
void DrawTargets();// Deklarace funkce

GL_Window* g_window;// Okno
Keys* g_keys;// Klávesy
```

Každý program potřebuje proměnné. Base ukládá display listy fontu, roll slouží k pohybu země a rolování mraků. Jako ve všech hrách i my začínáme prvním levelem. Miss vede záznam, do kolika objektů se v daném levelu střelec nestrefil, kill je jeho pravý opak. Score zahrnuje součty zasažených objektů z jednotlivých levelů. Game signalizuje konec hry.

```
GLuint base;// Display listy fontu
GLfloat roll;// Rolování mraků

GLint level = 1;// Aktuální level
GLint miss;// Počet nesestřelených objektů
GLint kills;// Počet sestřelených objektů v daném levelu
GLint score;// Aktuální skóre

bool game;// Konec hry?
```

Nadefinujeme nový datový typ, díky kterému budeme moci předat struktury porovnávací funkci. Qsort() totiž očekává v posledním parametru ukazatel na funkci s parametry (const* void, const* void).

```
typedef int (*compfn)(const void*, const void*);// Ukazatel na porovnávací funkci
```

Struktura objects bude ukládat všechny informace popisující sestřelovaný objekt. Rychlý průzkum proměnných: rot určuje směr rotace na ose z. Pokud ještě nebyl objekt sestřelen, hit bude obsahovat false. Frame definuje snímek animace při explozi, dir určuje směr pohybu. Texid je indexem do pole textur, nabývá hodnot nula až čtyři, z čehož plyne, že máme celkem pět druhů objektů. X a y definuje aktuální pozici, spin úhel rotace na ose z. Distance je hodně důležitá proměnná, určuje hloubku ve scéně. Právě podle ní budeme při blendingu řadit objekty, aby se nejdříve vykreslovali vzdálenější a až po nich bližší.

```
struct objects// Struktura objektu
{
    GLuint rot;// Rotace (0 - žádná, 1 - po směru hodinových ručiček, 2 - proti směru)
    bool hit;// Byl objekt zasažen?

    GLuint frame;// Aktuální snímek exploze
    GLuint dir;// Směr pohybu (0 - vlevo, 1 - vpravo, 2 - nahoru, 3 - dolů)
    GLuint texid;// Index do pole textur

    GLfloat x;// X pozice
    GLfloat y;// Y pozice
    GLfloat spin;// Směr rotace na ose z
    GLfloat distance;// Hloubka ve scéně
};
```

Následující pole vedou záznamy o deseti texturách a třiceti objektech.

```
TextureImage textures[10];// Deset textur
objects object[30];// 30 Objektů
```

Nebudeme limitovat velikost objektů. Váza by měla být vyšší než plechovka coly a kýbl naopak širší než váza. Abychom si ulehčili život, vytvoříme strukturu obsahující výšku a šířku. Definujeme a ihned inicializujeme pole těchto struktur o pěti prvcích. Na každém indexu se nachází jeden z pěti typů objektů.

```
struct dimensions// Rozměr objektu
{
    GLfloat w;// Šířka
    GLfloat h;// Výška
};

// Velikost každého objektu: Modrá tvář, kýbl, terč, Coca-cola, Váza
dimensions size[5] = {{1.0f,1.0f}, {1.0f,1.0f}, {1.0f,1.0f}, {0.5f,1.0f}, {0.75f,1.5f}};
```

Tento kód bude volán funkcí qsort(). Porovnává hloubku dvou objektů ve scéně a vrací -1, pokud je první objekt dále, bude-li ale vzdálenější druhý objekt vrátí funkce 1. Získáme-li 0, znamená to, že jsou oba ve stejné vzdálenosti od pozorovatele.

```
// *** Modifikovaný MSDN kód pro tento tutoriál ***
int Compare(struct objects *elem1, struct objects *elem2)// Porovnávací funkce
{
    if (elem1->distance < elem2->distance)// První je vzdálenější
    {
        return -1;
    }
    else if (elem1->distance > elem2->distance)// První je bližší
    {
        return 1;
    }
    else// Vzdálenosti jsou stejné
    {
        return 0;
    }
}
```

Ve funkci InitObject() nastavujeme objekt na výchozí hodnoty. Přiřadíme mu rotaci po směru hodinových ručiček. Animace exploze samozřejmě začíná na prvním (nultém) snímku. Objekt ještě nebyl zasažen, takže nastavíme hit na false. Randomem zvolíme jednu z pěti dostupných textur.

```
GLvoid InitObject(int num)// Inicializace objektu
{
    object[num].rot = 1;// Rotace po směru hodinových ručiček
    object[num].frame = 0;// První snímek exploze
    object[num].hit = FALSE;// Ještě nebyl zasažen
    object[num].texid = rand() % 5;// Náhodný index textury
}
```

Vzdálenost od pozorovatele nastavíme opět náhodně na hodnotu 0.0f až -40.0f ($4000/100 = 40$). Před renderingem objektu však scénu ještě posouváme do hloubky o dalších deset jednotek, takže se objekt defakto zobrazí v rozmezí od -10.0f do -50.0f. Ani příliš blízko ani příliš daleko.

```
object[num].distance = -(float(rand() % 4001) / 100.0f); // Náhodná hloubka
```

Po definování hloubky určíme výšku nad zemí. Nechceme, aby se objekt nacházel níže než -1.5f, protože by byl pod zemí. Také by neměl být výše než 3.0f. Abychom zůstali v tomto rozmezí, výsledek randomu nesmí být vyšší než $4.5f (-1.5f + 4.5f = 3.0f)$.

```
object[num].y = -1.5f + (float(rand() % 451) / 100.0f); // Náhodná y pozice
```

Výpočet počáteční x pozice je maličko složitější. Vezmeme pozici objektu v hloubce a odečteme od ní 15.0f. Výsledek operace vydělíme dvěma a odečteme od něj $5 * \text{level}$. Následuje další odčítání. Tentokrát odečteme náhodné číslo od 0 do 5 násobené aktuálním levelem. Předpokládám, že nechápete :-). Objekty se nyní ve vyšších levelech zobrazují dále od viditelné části scény (vlevo nebo vpravo). Kdybychom toto neudělali, zobrazovaly by se rychle jeden za druhým, takže by bylo velmi obtížné všechny zasáhnout a dostat se tak do dalšího levelu.

Abyste lépe pochopili určování x pozice, uvedu příklad. Řekněme, že se objekt nachází -30.0f jednotek hluboko ve scéně a aktuální level je 1.

```
object[num].x = ((-30.0f - 15.0f) / 2.0f) - (5*1) - float(rand() % (5*1));
object[num].x = (-45.0f / 2.0f) - 5 - float(rand() % 5);
object[num].x = (-22.5f) - 5 - { řekněme 3.0f };
object[num].x = (-22.5f) - 5 - { 3.0f };
object[num].x = -27.5f - { 3.0f };
object[num].x = -30.5f;
```

Před renderingem objektu provádíme translaci o deset jednotek do scény na ose z a hloubka v našem příkladu je -30.0f. Celková hloubka ve scéně je tedy -40.0f. Používáním perspektivního kódu z NeHeGL můžeme předpokládat, že levý okraj viditelné scény je -20.0f a pravý okraj se nachází na +20.0f. Před odečítáním randomů se rovná x-ová pozice -22.5f, což je PRÁVĚ okraj viditelné scény. Po těchto operacích to už je ale -30.0f a to znamená, že než se poprvé objeví, musí nejdříve urazit celých 8 jednotek doprava. Už je to jasnější?

```
// Náhodná x pozice založená na hloubce v obrazovce a s náhodným zpožděním před vstupem na scénu
object[num].x = ((object[num].distance - 15.0f) / 2.0f) - (5*level) - float(rand() % (5*level));
```

Nakonec zvolíme náhodný směr pohybu: 0 vlevo nebo 1 vpravo.

```
object[num].dir = (rand() % 2); // Náhodný směr pohybu
```

Nyní se podíváme, kterým směrem se bude objekt posouvat. Pokud půjde doleva ($\text{dir} == 0$), změním rotaci na proti směru hodinových ručiček ($\text{rot} = 2$). Pozice na ose x je defaultně záporná. Nicméně, pokud se máme pohybovat vpravo, musíme se na začátku nacházet vpravo. Negujeme tedy hodnotu x.

```
if (object[num].dir == 0) // Pohybuje se doleva?
{
    object[num].rot = 2; // Rotace proti směru hodinových ručiček
    object[num].x = -object[num].x; // Výchozí pozice vpravo
}
```

Zjistíme, který druh objektu počítač vybral. Pokud se index textury rovná nule, zvolil texturu modré tváře a ty se vždy pohybují těsně nad zemí. Ručně nastavíme y pozici na -2.0f.

```
if (object[num].texid == 0) // Modrá tvář
{
    object[num].y = -2.0f; // Vždy těsně nad zemí
}
```

Práce s objektem kýblu bude složitější. Padají totiž z nebe ($\text{dir} = 3$). Z toho také plyne, že bychom měli nastavit novou x-ovou pozici, protože by nikdy nebyl vidět (objekty jsou na začátku vždy vlevo nebo vpravo od scény). Namísto odečítání 15 z minulého příkladu odečteme pouze 10. Tímto dosáhneme menšího rozmezí hodnot, které udrží objekt viditelně na scéně. Předpokládáme-li, že se hloubka rovná -30.0f, skončíme s náhodnou hodnotou od 0.0f do +40.0f. Horní hodnota je kladná a ne záporná, jak by se mohlo zdát, protože $\text{rand}()$ vždy vrací kladné číslo. Získali jsme tedy číslo od 0.0f do 40.0f, k němu přičteme hloubku (záporné číslo) minus 10.0f a to celé dělené dvěma. Opět příklad: předpokládáme, že vrácená náhodná hodnota je 15 a objekt se nachází ve vzdálenosti -30.0f jednotek.

```
object[num].x = float(rand() % int(-30.0f - 10.0f)) + ((-30.0f - 10.0f) / 2.0f);
object[num].x = float(rand() % int(-40.0f) + (-40.0f) / 2.0f);
object[num].x = { předpokládejme 15 } + (-20.0f);
```

```
object[num].x = 15.0f - 20.0f;
object[num].x = -5.0f;
```

Nakonec určíme umístění na ose y. Chceme, aby padal z oblohy, ale nevystupoval z mraků. Číslo 4.5f odpovídá pozici maličko níže pod mraky.

```
if (object[num].texid == 1) // Kýbl
{
    object[num].dir = 3; // Padá dolů
    object[num].x = float(rand() % int(object[num].distance - 10.0f)) + ((object
[num].distance - 10.0f) / 2.0f);
    object[num].y = 4.5f; // Těsně pod mraky
}
```

Objekt terče by měl vystoupit nahoru ze země (dir = 2). Pro umístění na ose x použijeme stejný postup jako před chvílí. Nechceme, aby jeho počáteční poloha začínala nad zemí, takže nastavíme y na -3.0f (pod zemí). Od něj odečteme náhodné číslo od nuly do 5*level, aby se neobjevil hned, ale se zpožděním až po chvíli. Čím vyšší level, tím déle trvá, než se objeví. To dává hráči trochu času na vzpomínání se - bez této operace by terče vyskakovaly rychle jeden za druhým.

```
if (object[num].texid == 2) // Terč
{
    object[num].dir = 2; // Vyletí vzhůru
    object[num].x = float(rand() % int(object[num].distance - 10.0f)) + ((object
[num].distance - 10.0f) / 2.0f);
    object[num].y = -3.0f - float(rand() % (5*level)); // Pod zemí
}
```

Všechny ostatní objekty se pohybují zleva doprava, a proto není nutné, abychom jejich nastavení nějakým způsobem měnili.

Mohli bychom už skončit, ale zbývá ještě udělat jednu velice důležitou věc. Aby alfa blending pracoval správně, musí být průhledné polygony vykreslovány od nejvzdálenějších po nejbližší a nesmí se protínat. Z buffer totiž vyřazuje vzdálenější polygony, jsou-li již nějaké před nimi. Kdyby ty přední nebyly průhledné, ničemu by to nevadilo a navíc by se rendering urychlil, nicméně, když jsou objekty vepředu průhledné, tak by objekty za nimi měly být vidět. Nyní se buď nezobrazí nebo je kolem předních vykreslen čtvercový tvar, reprezentující původní polygon bez průhlednosti... nic hezkého.

Známe hloubku všech objektů, takže není žádný problém, abychom je po inicializaci nového seřadili, jak potřebujeme. Použijeme standardní funkci qsort() (quick sort - rychlé řazení). Při následném renderingu vezmeme první prvek pole a vykreslíme ho. Nebudeme se muset o nic starat, protože víme, že je ve scéně nejhluběji.

Tento kód jsme našli v MSDN, ale úspěchu předcházelo dlouhé hledání na internetu. Funkce qsort() pracuje dobře a dovoluje řadit celé struktury. Předáváme jí čtyři parametry. První ukazuje na pole objektů, které mají být seřazeny, druhý určuje jejich počet (odpovídá aktuálnímu levelu). Třetí parametr definuje velikost jedné struktury a čtvrtý je ukazatelem na porovnávací funkci Compare(). S největší pravděpodobností existuje nějaká lepší metoda pro řazení struktur, ale qsort() vyhovuje. Je rychlá a snadno se používá.

Důležitá poznámka: Pokud používáte glEnable(GL_ALPHA_TEST) namísto "klasického" blendingu, není řazení nutné. Používáním alpha funkcí jste ale omezeni na úplnou průhlednost nebo úplnou neprůhlednost, nic mezi tím. Používání BlendFunc() a řazení objektů stojí sice trochu práce navíc, ale dovoluje mít objekty poloprůhledné.

```
// *** Modifikovaný MSDN kód pro tento tutoriál ***
qsort((void *) &object, level, sizeof(struct objects), (compfn)Compare); // Řazení
objektů podle hloubky
}
```

První dva příkazy v inicializačním kódu nagražují informace o okně a indikátoru stisknutých kláves. Funkcí srand() inicializujeme generátor náhodných čísel, potom loadujeme textury a vytvoříme display listy fontu.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Inicializace OpenGL
{
    g_window = window;
    g_keys = keys;

    srand((unsigned)time(NULL)); // Inicializace generátoru náhodných čísel

    if ((!LoadTGA(&textures[0], "Data/BlueFace.tga")) || // Modrá tvář
        (!LoadTGA(&textures[1], "Data/Bucket.tga")) || // Kbelík
        (!LoadTGA(&textures[2], "Data/Target.tga")) || // Terč
        (!LoadTGA(&textures[3], "Data/Coke.tga")) || // Coca-Cola
        (!LoadTGA(&textures[4], "Data/Vase.tga")) || // Váza
```

```

(!LoadTGA(&textures[5], "Data/Explode.tga")) ||// Exploze
(!LoadTGA(&textures[6], "Data/Ground.tga")) ||// Země
(!LoadTGA(&textures[7], "Data/Sky.tga")) ||// Obloha
(!LoadTGA(&textures[8], "Data/Crosshair.tga")) ||// Kurzor
(!LoadTGA(&textures[9], "Data/Font.tga")) // Font
{
    return FALSE;// Inicializace se nezdařila
}

BuildFont();// Vytvoří display listy fontu

```

Nastavíme černé pozadí. Depth bufferem testujeme na méně nebo rovno (GL_LEQUAL).

```

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);// Černé pozadí
glClearDepth(1.0f);// Nastavení depth bufferu
glDepthFunc(GL_LEQUAL);// Typ testování hloubky
glEnable(GL_DEPTH_TEST);// Zapne testování hloubky

```

Příkaz `glBlendFunc()` je VELMI důležitý. Parametry `GL_SRC_ALPHA`, `GL_ONE_MINUS_SRC_ALPHA` oznamují OpenGL, aby při renderingu používalo alfa hodnoty uložené v textuře. Aby se blending mohl projevit, musíme ho zapnout. Dále zapínáme i mapování 2D textur a ořezávání zadních stran polygonů. Při kreslení zadáváme souřadnice polygonů proti směru hodinových ručiček, takže odstranění zadních stran polygonů ničemu nevádí. Navíc se program urychlí, protože má s kreslením pouze polovinu práce.

Výše v tutoriálu jsem psal o použití `glAlphaFunc()` namísto blendingu. Pokud chcete používat raději alfa funkci, zakomentárujte dva řádky důležité pro blending a odkomentárujte dva řádky alfy. Zakomentářovat můžete také řazení objektů pomocí `qsort()` a vše s ním spojené. Při alfa testingu není pořadí renderingu důležité.

Program půjde v pořádku, ale obloha se nezobrazí. Příčinou je její textura, která má alfa hodnotu 0.5f. Alfa, narozdíl od blendingu, však může být buď nula nebo jedna, nic mezi. Problém lze vyřešit modifikací alfa kanálu textury. Obě metody přinášejí velmi dobré výsledky.

```

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);// Nastavení alfa blendingu
glEnable(GL_BLEND);// Zapne alfa blending

// glAlphaFunc(GL_GREATER, 0.1f);// Nastavení alfa testingu
// glEnable(GL_ALPHA_TEST);// Zapne alfa testing

glEnable(GL_TEXTURE_2D);// Zapne mapování textur
glEnable(GL_CULL_FACE);// Ořezávání zadních stran polygonů

```

Na tomto místě inicializujeme všechny objekty, které program používá a potom ukončíme funkci.

```

for (int loop = 0; loop < 30; loop++)// Prochází všechny objekty
{
    InitObject(loop);// Inicializace každého z nich
}

return TRUE;// Inicializace úspěšná
}

```

Naprogramujeme detekci zásahů do objektů. Ze všeho nejdříve deklarujeme buffer, který použijeme k uložení informací o vybraných objektech. Proměnná `hits` slouží k počítání zásahů.

```

void Selection(void) // Detekce zasažení objektů
{
    GLuint buffer[512]; // Deklarace selection bufferu
    GLint hits; // Počet zasažených objektů

```

Skončila-li hra, není žádný důvod, abychom hledali, který objekt byl zasažen, a proto ukončíme funkci. Pokud je hráč stále ve hře, přehrajeme zvuk výstřelu. Tato funkce je volána pouze tehdy, když hráč stiskl tlačítko myši. A pokud stiskl tlačítko myši, znamená to, že chtěl vystřelit. Nezáleží, jestli zasáhl nebo ne, zvuk výstřelu je slyšet vždy. Přehrajeme ho v asynchroním módu (`SND_ASYNC`), aby běžel na pozadí a program nemusel čekat až skončí.

```

if (game) // Konec hry?
{
    return; // Není důvod testovat na zásah
}

PlaySound("data/shot.wav", NULL, SND_ASYNC); // Přehraje zvuk výstřelu

```

Nastavíme pole viewport tak, aby obsahovalo pozici `x`, `y` se šířkou a výškou aktuálního viewportu (OpenGL okna). Voláním funkce `glSelectBuffer()` nařídíme OpenGL, aby použilo naše pole `buffer` pro svůj `selection buffer`.

```

GLint viewport[4]; // Velikost viewportu. [0] = x, [1] = y, [2] = výška, [3] = šířka

glGetIntegerv(GL_VIEWPORT, viewport); // Nastaví pole podle velikosti a lokace scény
relativně k oknu
glSelectBuffer(512, buffer); // Přikáže OpenGL, aby pro selekci objektů použilo pole
buffer

```

Všechn kód níže je velmi důležitý. Nejdříve převedeme OpenGL do selection módu. Nic, co se vykresluje, se nezobrazí, ale namísto toho se informace o renderovaných objektech uloží do selection bufferu. Potom voláním `glInitNames()` a `glPushName(0)` inicializujeme name stack (stack jmen). Kdyby OpenGL nebylo v selection módu, `glPushName()` by bylo ignorováno.

```

(void) glRenderMode(GL_SELECT); // Převedení OpenGL do selection módu

glInitNames(); // Inicializace name stacku
glPushName(0); // Vloží 0 (nejméně jedna položka) na stack

```

Po přípravě name stacku musíme omezit kreslení na oblast pod kurzorem. Zvolíme projekční matici, pushneme ji na stack a resetujeme ji voláním `glLoadIdentity()`.

```

glMatrixMode(GL_PROJECTION); // Zvolí projekční matici
glPushMatrix(); // Uložení projekční matice
glLoadIdentity(); // Reset matice

```

Oblast kreslení omezíme příkazem `gluPickMatrix()`. První parametr určuje pozice myši na ose x, druhý je na ose y. Jedničky představují šířku a výšku picking regionu. Posledním parametrem je pole viewport, které určuje aktuální okraje viewportu. `mouse_x` a `mouse_y` budou středem picking regionu.

```

// Vytvoření matice, která zvětší malou část obrazovky okolo kurzoru myši
gluPickMatrix((GLdouble) mouse_x, (GLdouble) (viewport[3] - mouse_y), 1.0f, 1.0f,
viewport);

```

Voláním `gluPerspective` vynásobíme perspektivní matici pick maticí, která omezuje vykreslování na oblast vyžádanou od `gluPickMatrix()`. Potom přepneme na matici `modelview` a vykreslíme sestřelované objekty. Kreslíme je funkcí `DrawTargets()` a ne `Draw()`, protože chceme určit zásahy do objektů a ne do oblohy, země nebo kurzoru. Po vykreslení objektů přepneme zpět na projekční matici a popneme ji ze stacku. Nakonec se znovu vrátíme k matici `modelview`. Posledním příkazem přepneme OpenGL zpět do renderovacího módu, takže se opět budou vykreslované objekty zobrazovat na scénu. Proměnná `hits` bude po přiřazení obsahovat počet objektů, které byly vykresleny na oblast specifikovanou `gluPickMatrix()`. Tedy tam, kde se nacházel kurzor myši při výstřelu.

```

// Aplikování perspektivní matice
gluPerspective(45.0f, (GLfloat) (viewport[2] - viewport[0]) / (GLfloat) (viewport[3]
- viewport[1]), 0.1f, 100.0f);
glMatrixMode(GL_MODELVIEW); // Modelview matice

DrawTargets(); // Renderuje objekty do selection bufferu

glMatrixMode(GL_PROJECTION); // Projekční matice
glPopMatrix(); // Obnovení projekční matice
glMatrixMode(GL_MODELVIEW); // Modelview matice

hits = glRenderMode(GL_RENDER); // Přepnutí do renderovacího módu, uložení počtu
objektů pod kurzorem

```

Zjistíme, jestli bylo zaznamenáno více než nula zásahů. Pokud ano, přiřadíme proměnné `choose` jméno prvního objektu, který byl vykreslen do picking oblasti. `Depth` ukládá, jak hluboko ve scéně se tento objekt nachází. Každý zásah zabírá v bufferu čtyři položky. První je počtem jmen v name stacku, když se zásah udál. Druhá položka představuje minimální z hodnotu (hloubku) ze všech vertexů, které protínaly zobrazenou oblast v čase zásahu. Třetí naopak obsahuje maximální z hodnotu a poslední položka je obsahem name stacku v čase zásahu, nebo-li jméno objektu. V tomto programu nás zajímá minimální z hodnota a jméno objektu.

```

if (hits > 0) // Bylo více než nula zásahů?
{
    int choose = buffer[3]; // Uloží jméno prvního objektu
    int depth = buffer[1]; // Uloží jeho hloubku
}

```

Založíme cyklus skrz všechny zásahy, abychom se ujistili, že žádný z objektů není blíže než ten první. Jinými slovy potřebujeme najít nejbližší objekt ke střelci. Kdybychom ho nehledali a střelec zasáhl dva překrývající se objekty najednou, mohl by ten v pořadí pole první být vzdálenější od pozorovatele. kliknutí myši by sestřelilo špatný objekt. Je jasné, že pokud je několik terčů za sebou, tak se při výstřelu zasáhne vždy ten nejbližší.

Každý objekt má v poli `buffer` čtyři položky, takže násobíme aktuální průběh čtyřmi. Abychom získali hloubku objektu (druhá položka), přičítáme jedničku. pokud je právě testovaná hloubka menší než aktuálně nejvyšší, přičítáme

informace o jménu objektu a jeho hloubce. Po všech průchodech cyklem, bude choose obsahovat jméno ke střelci nejbližšího zasaženého objektu a depth jeho hloubku.

```
for (int loop = 1; loop < hits; loop++)// Prochází všechny detekované zásahy
{
    if (buffer[loop*4 + 1] < GLuint(depth))// Je tento objekt blíže než některý
    z předchozích?
    {
        choose = buffer[loop*4 + 3];// Uloží jméno bližšího objektu
        depth = buffer[loop*4 + 1];// Uloží jeho hloubku
    }
}
```

Našli jsme zasažený objekt. Přiřazením TRUE do hit ho označíme, aby nemohl být zasažen po druhé nebo zničen automaticky po opuštění scény. Přičteme k hráčovu score jedničku a také inkrementujeme počet zásahů v daném levelu.

```
if (!object[choose].hit)// Nebyl ještě objekt zasažen?
{
    object[choose].hit = TRUE;// Označí ho jako zasažený

    score += 1;// Zvýší celkové skóre
    kills += 1;// Zvýší počet zásahů v levelu
}
```

Chceme, aby v každém následujícím levelu musel hráč sestřelit větší počet objektů. Tím se znesnadňuje postup mezi levely. Zkontrolujeme, jestli je kills větší než aktuální level násobený pěti. V levelu jedna stačí pro postup sestřelit pouze pět objektů (1*5). V druhém levelu už je to deset (2*5), atd. Hra začíná být těžší a těžší.

Nastal-li čas pro přesun do následujícího levelu, nastavíme počet nezasažených objektů na nulu, aby jím měl hráč větší šanci úspěšně projít. Ale aby vše nebylo zase tak jednoduché, vynulujeme i počet zasažených objektů. Nakonec nesmíme zapomenout inkrementovat level a otestovat, jestli už nebyl poslední. Důvod proč máme zrovna třicet levelů je velice jednoduchý. Třicátý level je už šíleně obtížný, myslím, že nikdo nemá šanci ho dosáhnout. Druhým důvodem je maximální počet objektů - je jich právě třicet. Chcete-li jich více upravujte program.

Na scéně můžete mít ale MAXIMÁLNĚ 64 objektů (0 až 63). Pokud jich zkusíte renderovat 65 a více, PICKING PŘESTANE PRACOVAT SPRÁVNĚ a začnou se dít podivné věci. Všechno od náhodně vybuchujících objektů až k celému vašemu počítači se kompletně zhroutí. 64 objektů je fyzikální limit OpenGL, stejně jako například 8 světel ve scéně.

Pokud jste nějakou šťastnou náhodou bohem :-)) a dostanete se až k třicátému levelu, výše už bohužel nepostoupíte. Nicméně celkové skóre se bude stále zvyšovat a počet zasažených i nezasažených objektů se vždy na tomto místě resetuje.

```
if (kills > level*5)// Čas pro další level?
{
    miss = 0;// Nulování nezasažených objektů
    kills = 0;// Nulování zasažených objektů v tomto levelu
    level += 1;// Posun na další level

    if (level > 30)// Poslední level?
    {
        level = 30;// Nastavení levelu na poslední
    }
}
}
```

Ve funkci Update() testujeme stisk kláves a aktualizujeme umístění objektů ve scéně. Jednou z příjemných věcí je předávaný parametr milliseconds, který definuje uplynulý čas od předchozího volání. Na jeho bázi posuneme objekt o danou vzdálenost. A výsledek? Hra půjde stejně rychle na libovolném procesoru. ALE je zde jeden nedostatek. Řekněme, že máme objekt pohybující se pět jednotek za deset sekund. Rychlý počítač posune objektem o půl jednotky za sekundu. Na pomalém systému může trvat 2 sekundy, než se funkce znovu zavolá. Tím vznikají různá zpoždění a trhání, zkrátka animace už není plynulá. Lepší řešení však neexistuje. Pomalý počítač nezrychlíte, leda koupit nový...

Ale zpátky ke kódu. První podmínka zjišťuje stisk klávesy ESC, který ukončuje aplikaci.

```
void Update(DWORD milliseconds)// Aktualizace pohybů ve scéně a stisk kláves
{
    if (g_keys->keyDown[VK_ESCAPE])// Klávesa ESC?
    {
        TerminateApplication(g_window);// Ukončení programu
    }
}
```



```
}
```

Klávesa F1 přepíná mód okna mezi systémem a fullscreenem.

```
if (g_keys->keyDown[VK_F1])// Klávesa F1?
{
    ToggleFullscreen(g_window);// Přepnutí fullscreen/okno
}
```

Stisk mezerníku po skončení hry založí novou. Inicializujeme všech třicet objektů, nastavíme konec hry na false, skóre na nulu, první level a zasažené i nezasažené objekty v tomto levelu také na nulu. Nic nepochopitelného.

```
if (g_keys->keyDown[' '] && game)// Mezerník na konci hry?
{
    for (int loop = 0; loop < 30; loop++)// Prochází všechny objekty
    {
        InitObject(loop);// Jejich inicializace
    }

    game = FALSE;// Ještě není konec hry

    score = 0;// Nulové skóre
    level = 1;// První level
    kills = 0;// Nula zasažených objektů
    miss = 0;// Nula nezasažených objektů
}
```

K vytvoření iluze plujících mraků a pohybující se země, odečteme od roll číslo 0.00005f násobené počtem milisekund od minulého renderingu. Princip časování jsme si vysvětlili výše.

```
roll -= milliseconds * 0.00005f;// Mraky plují a země se pohybuje
```

Založíme cyklus, který prochází všechny objekty ve scéně a aktualizuje je. Jejich počet je roven aktuálnímu levelu.

```
for (int loop = 0; loop < level; loop++)// Aktualizace všech viditelných objektů
{
```

Potřebujeme zjistit, kterým směrem se který objekt otáčí. Podle směru rotace upravíme aktuální úhel natočení o 0.2 stupňů vynásobených řídicí proměnnou cyklu sečtenou s milisekundami. Přičítáním loop získáme rozdílnou rotaci pro každý objekt. Druhý objekt se nyní otáčí rychleji než první a třetí objekt ještě rychleji než druhý.

```
if (object[loop].rot == 1)// Rotace po směru hodinových ručiček?
    object[loop].spin -= 0.2f * (float(loop + milliseconds));

if (object[loop].rot == 2)// Rotace proti směru hodinových ručiček?
    object[loop].spin += 0.2f * (float(loop + milliseconds));
```

Přesuneme se ke kódu zajišťujícímu pohyby. Pokud se objekt pohybuje doprava (dir == 1), přičteme k x pozici 0.0012f. Podobným způsobem ošetříme posun doleva (dir == 0). Při směru nahoru (dir == 2) zvětšíme y hodnotu, protože kladná část osy y leží nahoře. Směr dolů (dir == 3) je úplně stejný jako předchozí. Odečítáme však menší číslo, aby byl pád pomalejší.

```
if (object[loop].dir == 1)// Pohyb doprava?
    object[loop].x += 0.012f * float(milliseconds);

if (object[loop].dir == 0)// Pohyb doleva?
    object[loop].x -= 0.012f * float(milliseconds);

if (object[loop].dir == 2)// Pohyb nahoru?
    object[loop].y += 0.012f * float(milliseconds);

if (object[loop].dir == 3)// Pohyb dolů?
    object[loop].y -= 0.0025f * float(milliseconds);
```

Posunuli jsme objektem a nyní potřebujeme otestovat, jestli je na scéně ještě vidět. Můžeme to zjistit podle hloubky ve scéně mínus 15.0f (malá tolerance navíc) a dělením dvěma. Pro ty z vás, kteří od inicializace objektů už zapoměli... Pokud jste dvacet jednotek ve scéně, máte z každé strany zhruba deset jednotek viditelné scény (záleží na nastavení perspektivy). Takže -20.0f (hloubka) -15.0f (extra okraj) = -35.0f. Vydělíme 2.0f a získáme -17.5f, což je přibližně 7.5 jednotek vlevo od viditelné scény. Objekt tedy už určitě není vidět.

Musí také platit podmínka, že se objekt pohybuje doleva (dir == 0). Pokud ne, nestaráme se o něj. Poslední část logického výrazu představuje test zásahu. Shrňme to: pokud objekt vyletěl vlevo ze scény, pohybuje se doleva a nebyl zasažen, uživatel ho už nemá šanci zasáhnout. Zvýšíme počet nezasažených objektů a označíme objekt jako zasažený, aby se o něj program už příště nestaral. Touto cestou (hit = true) také zajistíme autodestrukci, což nám po nějaké době

umožní jeho automatickou reinicializaci - nová textura, směr pohybu, rotace ap.

```
// Objekt vyletěl vlevo ze scény, pohybuje se vlevo a ještě nebyl zasažen
if ((object[loop].x < (object[loop].distance - 15.0f) / 2.0f) && (object
[loop].dir == 0) && !object[loop].hit)
{
    miss += 1; // Zvýšení počtu nezasažených objektů
    object[loop].hit = TRUE; // Odstranění objektu (zajišťuje animaci exploze a
    reinicializaci)
}
```

Analogicky ošetříme opuštění scény vpravo a náraz do země.

```
// Objekt vyletěl vpravo ze scény, pohybuje se vpravo a ještě nebyl zasažen
if ((object[loop].x > -(object[loop].distance - 15.0f) / 2.0f) && (object
[loop].dir == 1) && !object[loop].hit)
{
    miss += 1; // Zvýšení počtu nezasažených objektů
    object[loop].hit = TRUE; // Odstranění objektu (zajišťuje animaci exploze a
    reinicializaci)
}

// Objekt narazil do země, pohybuje se dolů a ještě nebyl zasažen
if ((object[loop].y < -2.0f) && (object[loop].dir == 3) && !object[loop].hit)
{
    miss += 1; // Zvýšení počtu nezasažených objektů
    object[loop].hit = TRUE; // Odstranění objektu (zajišťuje animaci exploze a
    reinicializaci)
}
```

Narozdíl od předchozích testů při letu vzhůru uděláme menší změnu. Pokud se objekt dostane na ose y výše než 4.5f jednotek (těsně pod mraky), nezničíme ho, ale pouze změním jeho směr, aby se pohyboval dolů. Destrukci zajistí předchozí kód pro naražení do země.

```
if ((object[loop].y > 4.5f) && (object[loop].dir == 2)) // Objekt je pod mraky a
směřuje vzhůru
    object[loop].dir = 3; // Změna směru na pád
}
```

Do mapy zpráv ve funkci WindowProc() přidáme dvě větve, které obsluhují události myši. Při stisknutí levého tlačítka uložíme pozici kliknutí v okně a ve funkci Selection() zjistíme, jestli se hráč střelil do některého z objektů nebo ne. Protože vykreslujeme vlastní OpenGL kurzor, potřebujeme při renderingu znát jeho pozici. O to se stará WM_MOUSEMOVE.

```
// Funkce WindowProc
case WM_LBUTTONDOWN: // Stisknutí levého tlačítka myši
    mouse_x = LOWORD(lParam);
    mouse_y = HIWORD(lParam);
    Selection();
    break;

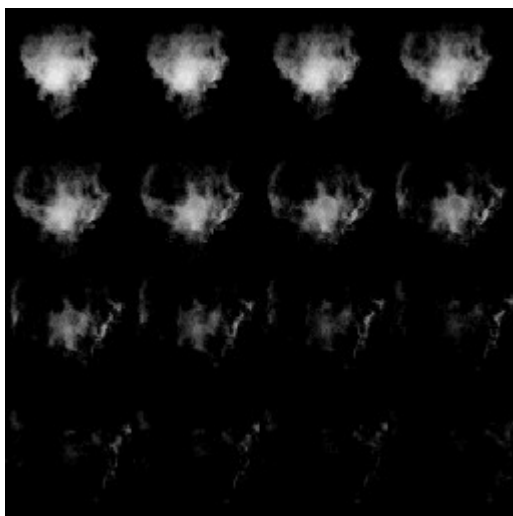
case WM_MOUSEMOVE: // Pohyb myši
    mouse_x = LOWORD(lParam);
    mouse_y = HIWORD(lParam);
    break;
```

Přistoupíme k vykreslení objektu. Funkci se předávají celkem tři parametry, které ho dostatečně popisují - šířka, výška a textura. Obdélník renderujeme zadáváním bodů proti směru hodinových ručiček, abychom mohli použít culling.

```
void Object(float width, float height, GLuint texid) // Vykreslí objekt
{
    glBindTexture(GL_TEXTURE_2D, textures[texid].texID); // Zvolí správnou texturu

    glBegin(GL_QUADS); // Kreslení obdélníků
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-width, -height, 0.0f); // Levý dolní
        glTexCoord2f(1.0f, 0.0f); glVertex3f( width, -height, 0.0f); // Pravý dolní
        glTexCoord2f(1.0f, 1.0f); glVertex3f( width, height, 0.0f); // Pravý horní
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-width, height, 0.0f); // Levý horní
    glEnd(); // Konec kreslení
}
```

Kód pro renderování exploze dostává pouze jeden parametr - identifikátor objektu. Potřebujeme nagrahovat souřadnice oblasti na textuře exploze. Uděláme to podobnou cestou, jako když jsme získávali jednotlivé znaky z textury fontu. Ex a ey představují sloupec a řádek závislý na pořadí snímku animace (framu).



Pozici na ose x získáme dělením aktuálního snímku čtyřmi. Protože máme 64 snímků a pouze 16 obrázků, potřebujeme animaci zpomalit. Zbytek po dělení upraví číslo na hodnoty 0 až 3 a aby texturové koordináty byly v rozmezí 0.0f a 1.0f, dělíme čtyřmi. Získali jsme sloupec, nyní ještě řádek. První dělení opět zmenšuje číslo, druhé dělení eliminuje celý řádek a posledním dělením získáme vertikální souřadnici na textuře.

Pokud je aktuální snímek 16, $ey = 16/4/4/4 = 4/4/4 = 0,25$. Jeden řádek dolů. je-li snímek 60, $ey = 60/4/4/4 = 15/4/4 = 3/4 = 0,75$. Matematici nevěří vlastním očím... Důvod proč se $15/4$ nerovná $3,75$ je to, že do posledního dělení pracujeme s celými čísly. Počítáme-li se zaokrouhlováním dojdeme k závěru, že výsledkem jsou vždy čísla 0.0f, 0.25f, 0.50f nebo 0.75f. Doufám, že to dává smysl. Je to jednoduché, ale matematika zastrašuje.

```
void Explosion(int num)// Animace exploze objektu
{
    float ex = (float)((object[num].frame/4)%4)/4.0f;// Výpočet x snímku exploze (0.0f - 0.75f)
    float ey = (float)((object[num].frame/4)/4)/4.0f;// Výpočet y snímku exploze (0.0f - 0.75f)
```

Získali jsme texturovací koordináty, zbývá vykreslit obdélník. Vertexy jsou fixovány na -1.0f a 1.0f. U textur odečítáme ey od 1.0f. Pokud bychom to neudělali animace by probíhala v opačném pořadí. Počátek texturovacích souřadnic je vlevo dole.

```
glBindTexture(GL_TEXTURE_2D, textures[5].texID);// Textura exploze

glBegin(GL_QUADS);// Kreslení obdélníků
glTexCoord2f(ex, 1.0f - (ey));
glVertex3f(-1.0f, -1.0f, 0.0f);// Levý dolní

glTexCoord2f(ex + 0.25f, 1.0f - (ey));
glVertex3f( 1.0f, -1.0f, 0.0f);// Pravý dolní

glTexCoord2f(ex + 0.25f, 1.0f - (ey + 0.25f));
glVertex3f( 1.0f, 1.0f, 0.0f);// Pravý horní

glTexCoord2f(ex, 1.0f - (ey + 0.25f));
glVertex3f(-1.0f, 1.0f, 0.0f);// Levý horní
glEnd();// Konec kreslení
```

Jak je vysvětleno výše, snímek nesmí být vyšší než 63, jinak by animace začala nanovo. Při přesáhnutí tohoto čísla reinitializujeme objekt.

```
object[num].frame += 1;// Zvýší snímek exploze

if (object[num].frame > 63)// Poslední snímek?
{
    InitObject(num);// Reinitializace objektu
}
}
```

Následující sekce kódu vykresluje objekty. Začneme resetováním matice a přesunem o deset jednotek do hloubky.

```
void DrawTargets(void) // Vykreslí objekty
{
    glLoadIdentity(); // Reset matice
    glTranslatef(0.0f, 0.0f, -10.0f); // Posun do hloubky
```

Založíme cyklus procházející všechny aktivní objekty. Funkcí `glLoadName()` skrytě označíme individuální objekty - každému se určí jméno (číslo), které odpovídá indexu v poli. Prvnímu se přiřadí nula, druhému jednička atd. Podle tohoto jména můžeme zjistit, který objekt byl zasažen. Pokud program není v selection módu `glLoadName()` je ignorováno. Po přiřazení jména uložíme matici.

```
for (int loop = 0; loop < level; loop++) // Prochází aktivní objekty
{
    glLoadName(loop); // Přiřadí objektu jméno (pro detekci zásahů)
    glPushMatrix(); // Uložení matice
```

Přesuneme se na pozici objektu, kde má být vykreslen.

```
glTranslatef(object[loop].x, object[loop].y, object[loop].distance); // Umístění objektu
```

Před renderingem testujeme, jestli byl zasažen nebo ne. Pokud podmínka platí, vykreslíme místo objektu snímek animace exploze, jinak otočíme objektem na ose z o jeho úhel spin a až potom ho vykreslíme. Pro určení rozměrů použijeme pole `size`, které jsme vytvořili na začátku programu. `Texid` reprezentuje typ objektu (texturu).

```
if (object[loop].hit) // Byl objekt zasažen?
{
    Explosion(loop); // Vykreslí snímek exploze
}
else // Objekt nebyl zasažen
{
    glRotatef(object[loop].spin, 0.0f, 0.0f, 1.0f); // Natočení na ose z
    Object(size[object[loop].texid].w, size[object[loop].texid].h, object[loop].texid); // Vykreslení
}
```

Po renderingu popneme matici, abychom zrušili posun a natočení.

```
glPopMatrix(); // Obnoví matici
}
```

`Draw()` je hlavní vykreslovací funkcí. Jako obvykle smažeme buffery a resetujeme matici, kterou následně pushneme.

```
void Draw(void) // Vykreslení scény
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže buffery
    glLoadIdentity(); // Reset matice

    glPushMatrix(); // Uloží matici
```

Zvolíme texturu (v pořadí sedmá) a pokusíme se vykreslit oblohu. Je složena ze čtyř otexturovaných obdélníků. První představuje oblohu od země přímo vzhůru. Textura na něm roluje docela pomalu. Druhý obdélník je vykreslen na stejném místě, ale jeho textura roluje rychleji. Obě textury se blendingem spojí dohromady a vytvoří tak hezký vícevrstvý efekt.

```
glBegin(GL_QUADS); // Kreslení obdélníků
glTexCoord2f(1.0f, roll/1.5f+1.0f); glVertex3f( 28.0f, +7.0f, -50.0f); // Pravý horní
glTexCoord2f(0.0f, roll/1.5f+1.0f); glVertex3f(-28.0f, +7.0f, -50.0f); // Levý horní
glTexCoord2f(0.0f, roll/1.5f+0.0f); glVertex3f(-28.0f, -3.0f, -50.0f); // Levý dolní
glTexCoord2f(1.0f, roll/1.5f+0.0f); glVertex3f( 28.0f, -3.0f, -50.0f); // Pravý dolní

glTexCoord2f(1.5f, roll+1.0f); glVertex3f( 28.0f, +7.0f, -50.0f); // Pravý horní
glTexCoord2f(0.5f, roll+1.0f); glVertex3f(-28.0f, +7.0f, -50.0f); // Levý horní
glTexCoord2f(0.5f, roll+0.0f); glVertex3f(-28.0f, -3.0f, -50.0f); // Levý dolní
glTexCoord2f(1.5f, roll+0.0f); glVertex3f( 28.0f, -3.0f, -50.0f); // Pravý dolní
```

Abychom přidali iluzi, že mraky plují směrem k pozorovateli, třetí obdélník směřuje z hloubky dopředu. Další obdélník je opět na stejném místě, ale textura roluje rychleji. Výsledkem čtyř obyčejných obdélníků je obloha, která se jeví, jako by

stoupala od země vzhůru a přibližovala se k pozorovateli. Mohl jsem použít otexturovanou polokouli, ale byl jsem příliš líný. Efekt s obdélníky vypadá celkem slušně.

```
glTexCoord2f(1.0f,roll/1.5f+1.0f); glVertex3f( 28.0f,+7.0f,0.0f);// Pravý horní
glTexCoord2f(0.0f,roll/1.5f+1.0f); glVertex3f(-28.0f,+7.0f,0.0f);// Levý horní
glTexCoord2f(0.0f,roll/1.5f+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f);// Levý dolní
glTexCoord2f(1.0f,roll/1.5f+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f);// Bottom
Right

glTexCoord2f(1.5f,roll+1.0f); glVertex3f( 28.0f,+7.0f,0.0f);// Pravý horní
glTexCoord2f(0.5f,roll+1.0f); glVertex3f(-28.0f,+7.0f,0.0f);// Levý horní
glTexCoord2f(0.5f,roll+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f);// Levý dolní
glTexCoord2f(1.5f,roll+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f);// Pravý dolní
glEnd();// Konec kreslení
```

Nyní vykreslíme zemi. Začíná tam, kde se nachází nejnižší bod oblohy a směřuje směrem k pozorovateli. Roluje stejně rychle jako mraky. Abychom přidali trochu více detailů a zamezili tak nepříjemnému kostičkování při velkém zvětšení, namapujeme texturu sedmkrát na ose x a čtyřikrát na ose y.

```
glBindTexture(GL_TEXTURE_2D, textures[6].texID);// Textura země

glBegin(GL_QUADS);// Kreslení obdélníků
glTexCoord2f(7.0f,4.0f-roll); glVertex3f( 27.0f,-3.0f,-50.0f);// Pravý horní
glTexCoord2f(0.0f,4.0f-roll); glVertex3f(-27.0f,-3.0f,-50.0f);// Levý horní
glTexCoord2f(0.0f,0.0f-roll); glVertex3f(-27.0f,-3.0f,0.0f);// Levý dolní
glTexCoord2f(7.0f,0.0f-roll); glVertex3f( 27.0f,-3.0f,0.0f);// Pravý dolní
glEnd();// Konec kreslení
```

Pozadí je vykresleno, přistoupíme k sestřelovaným objektům. Napsali jsme pro ně speciální funkci. Potom obnovíme matici.

```
DrawTargets();// Sestřelované objekty
glPopMatrix();// Obnovení matice
```

Vykreslíme kurzor myši. Nagrabované rozměry okna uložíme do struktury obdélníku window. Zvolíme projekční matici a pushneme ji, resetujeme ji a převedeme scénu z perspektivního módu do pravoúhlé projekce. Souřadnice 0, 0 se nacházejí vlevo dole.

Ve funkci glOrtho() prohodíme třetí a čtvrtý parametr, aby byl kurzor renderován proti směru hodinových ručiček a culling pracoval tak, jak chceme. Kdyby byl počátek souřadnic nahoře, zadávání bodů by probíhalo v opačném směru a kurzor s textem by se nezobrazil.

```
RECT window;// Proměnná obdélníku
GetClientRect (g_window->hWnd,&window);// Grabování rozměrů okna

glMatrixMode(GL_PROJECTION);// Projekční matice
glPushMatrix();// Uloží projekční matici
glLoadIdentity();// Reset projekční matice

glOrtho(0, window.right, 0, window.bottom, -1, 1);// Nastavení pravoúhlé scény
```

Po nastavení kolmé projekce zvolíme modelview matici a umístíme kurzor. Problém je v tom, že počátek scény (0, 0) je vlevo dole, ale okno (systém) ho má vlevo nahoře. Kdybychom pozici kurzoru neinvertovali, tak by se při posunutí dolů, pohyboval nahoru. Od spodního okraje okna odečteme mouse_y. Namísto předávání velikosti v OpenGL jednotkách, specifikujeme šířku a výšku v pixelech.

Rozhodl jsem se použít vlastní a ne systémový kurzor ze dvou důvodů. První a více důležitý je, že vypadá lépe a může být modifikován v jakémkoli grafickém editoru, který podporuje alfa kanál. Druhým důvodem je, že některé grafické karty kurzor ve fullscreenu nezobrazují. Hrát hru podobného typu bez kurzoru není vůbec snadné :-).

```
glMatrixMode(GL_MODELVIEW);// Zvolí matici modelview
glTranslated(mouse_x, window.bottom-mouse_y, 0.0f);// Posun na pozici kurzoru

Object(16, 16, 8);// Vykreslí kurzor myši
```

Vypíšeme logo NeHe Productions zarovnané na střed horní části okna, dále zobrazíme aktuální level a skóre.

```
glPrint(240, 450, "NeHe Productions");// Logo
glPrint(10, 10, "Level: %i", level);// Level
glPrint(250, 10, "Score: %i", score);// Skóre
```

Otestujeme, jestli hráč nestrefil více než devět objektů. Pokud ano, nastavíme game na true, čímž indikujeme konec hry.

```
if (miss > 9)// Nestrefil hráč více než devět objektů?
```

```

{
    miss = 9; // Limit je devět
    game = TRUE; // Konec hry
}

```

Po skončení hry vypisujeme text GAME OVER. Je-li hráč ještě ve hře, vypíšeme kolik objektů mu může ještě uniknout. Text je ve formátu např. '6/10' - může ještě nezasáhnout šest objektů z deseti.

```

if (game) // Konec hry?
{
    glPrint(490, 10, "GAME OVER"); // Vypíše konec hry
}
else
{
    glPrint(490, 10, "Morale: %i/10", 10-miss); // Vypíše počet objektů, které nemusí
    sestřelit
}

```

Zbývá obnovit původní nastavení. Zvolíme projekční matici, obnovíme ji, zvolíme modelview matici a vyprázdníme buffer, abychom se ujistili, že všechny objekty byly v pořádku zobrazeny.

```

glMatrixMode(GL_PROJECTION); // Projekční matice
glPopMatrix(); // Obnovení projekční matice
glMatrixMode(GL_MODELVIEW); // Modelview matice

glFlush(); // Vyprázdní OpenGL renderovací pipeline
}

```

Tento tutoriál je výsledkem mnoha probdělých nocí při kódování a psaní HTML. Nyní byste měli rozumět pickingu, alfa testingu a řazení podle hloubky při alfa blendingu. Picking umožňuje vytvořit interaktivní software, který se ovládá myší. Všechno od her až po nádherné GUI. Největší výhodou pickingu je, že si nemusíme vést složitý záznam, kde se objekty nacházejí, o translacích a rotacích ani nemluvě. Objektu stačí přiřadit jméno a počkat na výsledek. S alfa blendingem a testingem můžete vykreslit objekt kompletně neprůhledný a/nebo plný otvorů. Výsledek je úžasný, nemusíte se starat o prosvítání textur.

Mohl jsem strávit spoustu času přidáváním pohybů podle fyzikálních zákonů, grafiky, zvuků a podobně. Nicméně jsem vysvětlil OpenGL techniky bez dalších zbytečností. Doufám, že se po čase objeví nějaké skvělé modifikace kódu, které už ale nechám na vás.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 33 - Nahrávání komprimovaných i nekomprimovaných obrázků TGA

V lekci 24 jsem vám ukázal cestu, jak nahrávat nekomprimované 24/32 bitové TGA obrázky. Jsou velmi užitečné, když potřebujete alfa kanál, ale nesmíte se starat o jejich velikost, protože byste je ihned přestali používat. K diskovému místu nejsou zrovna šetrné. Problém velikosti vyřeší nahrávání obrázků komprimovaných metodou RLE. Kód pro loading a hlavičkové soubory jsou odděleny od hlavního projektu, aby mohly být snadno použity i jinde.

Začneme dvěma hlavičkovými soubory. Texture.h, první z nich, popisuje strukturu textury. Každý hlavičkový soubor by měl obsahovat ochranu proti vícenásobnému vložení. Zajišťují ji příkazy preprocesoru jazyka C. Pokud není definovaná symbolická konstanta `__TEXTURE_H__`, nadefinujeme ji a do stejného bloku podmínky vepíšeme zdrojový kód. Při následujícím pokusu o inkudování hlavičkového souboru existence konstanty oznámí preprocesoru, že už byl soubor jednou vložen, a tudíž ho nemá vkládat podruhé.

```
#ifndef __TEXTURE_H__
#define __TEXTURE_H__
```

Budeme potřebovat strukturu informací o obrázku, ze kterého se vytváří textura. Ukazatel `imageData` obsahuje data obrázku, `bpp` barevnou hloubku, `width` a `height` rozměry. `TexID` je identifikátorem OpenGL textury, který se předává funkci `glBindTexture()`. Type určuje typ textury - `GL_RGB` nebo `GL_RGBA`.

```
typedef struct// Struktura textury
{
    GLubyte* imageData;// Data
    GLuint bpp;// Barevná hloubka v bitech
    GLuint width;// Šířka
    GLuint height;// Výška
    GLuint type;// Typ (GL_RGB, GL_RGBA)
    GLuint texID;// ID textury
} Texture;

#endif
```

Druhý hlavičkový soubor, `tga.h`, je speciálně určen pro loading TGA. Opět začneme ošetřením vícenásobného inkudování, poté vložíme hlavičkový soubor textury.

```
#ifndef __TGA_H__
#define __TGA_H__

#include "texture.h"// Hlavičkový soubor textury
```

Strukturu `TGAHeader` představuje pole dvanácti bytů, které ukládají hlavičku obrázku. Druhá struktura obsahuje pomocné proměnné pro nahrávání - např. velikost dat, barevnou hloubku a podobně.

```
typedef struct// Hlavička TGA souboru
{
    GLubyte Header[12];// Dvanáct bytů
} TGAHeader;

typedef struct// Struktura obrázku
{
    GLubyte header[6];// Šest užitečných bytů z hlavičky
    GLuint bytesPerPixel;// Barevná hloubka v bytech
    GLuint imageSize;// Velikost paměti pro obrázek
    // GLuint temp;// Překl.: nikde není použita
    GLuint type;// Typ
    GLuint Height;// Výška
    GLuint Width;// Šířka
    GLuint Bpp;// Barevná hloubka v bitech
} TGA;
```

Deklarujeme instance právě vytvořených struktur, abychom je mohli použít v programu.

```
TGAHeader tgaheader;// TGA hlavička
```

```
TGA tga;// TGA obrázek
```

Následující dvě pole pomohou určit validitu nahrávaného souboru. Pokud se hlavička obrázku neshoduje s některou z nich, neumíme ho nahrát.

```
GLubyte uTGACompare[12] = { 0,0, 2,0,0,0,0,0,0,0,0,0 };// TGA hlavička nekomprimovaného obrázku
```

```
GLubyte cTGACompare[12] = { 0,0,10,0,0,0,0,0,0,0,0,0 };// TGA hlavička komprimovaného obrázku
```

Obě funkce nahrávají TGA - jedna nekomprimovaný druhá komprimovaný.

```
bool LoadUncompressedTGA(Texture*, char*, FILE*);// Nekomprimovaný TGA
```

```
bool LoadCompressedTGA(Texture*, char*, FILE*);// Komprimovaný TGA
```

```
#endif
```

Přesuneme se k souboru TGAloader.cpp, který implementuje nahrávací funkce. Prvním řádkem kódu vložíme hlavičkový soubor. Inkludujeme pouze tga.h, protože texture.h jsme už vložili v něm.

```
#include "tga.h"// Hlavičkový soubor TGA
```

Funkce LoadTGA() je ta, kterou v programu voláme, abychom nahráli obrázek. V parametrech se jí předává ukazatel na texturu a řetězec diskové cesty. Nic dalšího nepotřebuje, protože si všechny ostatní parametry detekuje sama (ze souboru). Deklarujeme handle souboru a otevřeme ho pro čtení v binárním módu. Pokud něco selže, např. soubor neexistuje, vypíšeme chybovou zprávu a vrátíme false jako indikaci chyby.

```
bool LoadTGA(Texture* texture, char* filename)// Nahraje TGA soubor
```

```
{
    FILE* fTGA;// Handle souboru
    fTGA = fopen(filename, "rb");// Otevře soubor

    if(fTGA == NULL)// Nepodařilo se ho otevřít?
    {
        MessageBox(NULL, "Could not open texture file", "ERROR", MB_OK);
        return false;
    }
}
```

Zkusíme načíst hlavičku obrázku (prvních 12 bytů souboru), která určuje jeho typ. Výsledek se uloží do proměnné tgaheader.

```
if(fread(&tgaheader, sizeof(TGAHeader), 1, fTGA) == 0)// Načte hlavičku souboru
{
    MessageBox(NULL, "Could not read file header", "ERROR", MB_OK);

    if(fTGA != NULL)
    {
        fclose(fTGA);
    }

    return false;
}
```

Právě načtenou hlavičku porovnáme s hlavičkou nekomprimovaného obrázku. Jsou-li shodné nahrajeme obrázek funkcí LoadUncompressedTGA(). Pokud shodné nejsou zkusíme, jestli se nejedná o komprimovaný obrázek. V tomto případě použijeme pro nahrávání funkci LoadCompressedTGA(). S jinými typy souborů pracovat neumíme, takže jediné, co můžeme udělat, je oznámení neúspěchu a ukončení funkce.

Překl.: Měla by se ještě testovat návratová hodnota, protože, jak uvidíte dále, funkce v mnoha případech vrací false. Program by si bez kontroly ničeho nevšiml a pokračoval dále.

```
if(memcmp(uTGACompare, &tgaheader, sizeof(tgaheader)) == 0)// Nekomprimovaný
{
    LoadUncompressedTGA(texture, filename, fTGA);

    // Překl.: Testovat návratovou hodnotu !!!
    // if(!LoadUncompressedTGA(texture, filename, fTGA))// Test návratové hodnoty
    // {
    //     // return false;
    // }
}
else if(memcmp(cTGACompare, &tgaheader, sizeof(tgaheader)) == 0)// Komprimovaný
{
```



```

LoadCompressedTGA(texture, filename, fTGA);

// Překl.: Testovat návratovou hodnotu !!!
// if(!LoadCompressedTGA(texture, filename, fTGA))// Test návratové hodnoty
// {
//     // return false;
// }
}
else// Ani jeden z nich
{
    MessageBox(NULL, "TGA file be type 2 or type 10 ", "Invalid Image", MB_OK);
    fclose(fTGA);
    return false;
}
}

```

Pokud dosud nenastala žádná chyba, můžeme oznámit volajícímu kódu, že obrázek byl v pořádku nahrán a že může z jeho dat vytvořit texturu.

```

return true;// Vše v pořádku
}

```

Přistoupíme k opravdovému nahrávání obrázků, začneme nekomprimovanými. Tato funkce je z velké části založena na té z lekce 24, moc novinek v ní nenajdete. Zkusíme načíst dalších šest bytů ze souboru a uložíme je do tga.header.

```

bool LoadUncompressedTGA(Texture* texture, char* filename, FILE* fTGA)// Nahraje
nekomprimovaný TGA
{
    if(fread(tga.header, sizeof(tga.header), 1, fTGA) == 0)// Šest užitečných bytů
    {
        MessageBox(NULL, "Could not read info header", "ERROR", MB_OK);

        if(fTGA != NULL)
        {
            fclose(fTGA);
        }

        return false;
    }
}

```

Máme dost informací pro určení výšky, šířky a barevné hloubky obrázku. Uložíme je do obou struktur - textury i obrázku.

```

texture->width = tga.header[1] * 256 + tga.header[0];// Šířka
texture->height = tga.header[3] * 256 + tga.header[2];// Výška
texture->bpp = tga.header[4];// Barevná hloubka v bitech

// Kopírování dat do struktury obrázku
tga.Width = texture->width;
tga.Height = texture->height;
tga.Bpp = texture->bpp;

```

Otestujeme, jestli má obrázek alespoň jeden pixel a jestli je barevná hloubka 24 nebo 32 bitů.

```

// Platné hodnoty?
if((texture->width <= 0) || (texture->height <= 0) || ((texture->bpp != 24) &&
(texture->bpp != 32)))
{
    MessageBox(NULL, "Invalid texture information", "ERROR", MB_OK);
    if(fTGA != NULL)
    {
        fclose(fTGA);
    }

    return false;
}
}

```

Nyní nastavíme typ obrázku. V případě 24 bitů je jím GL_RGB, u 32 bitů má obrázek i alfa kanál, takže použijeme GL_RGBA.

```

if(texture->bpp == 24)// 24 bitový obrázek?
{
    texture->type = GL_RGB;
}
}

```

```

else// 32 bitový obrázek
{
    texture->type = GL_RGBA;
}

```

Spočítáme barevnou hloubku v BYTECH a celkovou velikost paměti potřebnou pro data. Vzápětí se ji pokusíme alokovat.

```

tga.bytesPerPixel = (tga.Bpp / 8);// BYTY na pixel
tga.imageSize = (tga.bytesPerPixel * tga.Width * tga.Height);// Velikost paměti
texture->imageData = (GLubyte *)malloc(tga.imageSize);// Alokace paměti pro data
if(texture->imageData == NULL)// Alokace neúspěšná
{
    MessageBox(NULL, "Could not allocate memory for image", "ERROR", MB_OK);
    fclose(fTGA);
    return false;
}

```

Pokud se podařila alokace paměti, nahrajeme do ní data obrázku.

```

// Pokusí se nahrát data obrázku
if(fread(texture->imageData, 1, tga.imageSize, fTGA) != tga.imageSize)
{
    MessageBox(NULL, "Could not read image data", "ERROR", MB_OK);

    if(texture->imageData != NULL)
    {
        free(texture->imageData);// Uvolnění paměti
    }

    fclose(fTGA);
    return false;
}

```

Formát TGA se od formátu OpenGL liší tím, že má v pixelech přehozené R a B složky barvy (BGR místo RGB). Musíme tedy zaměnit první a třetí byte v každém pixelu. Abychom tuto operaci urychlili, provedeme tři binární operace XOR. Výsledek je stejný jako při použití pomocné proměnné.

```

// Převod BGR na RGB
for(GLuint cswap = 0; cswap < (int)tga.imageSize; cswap += tga.bytesPerPixel)
{
    texture->imageData[cswap] ^= texture->imageData[cswap+2] ^=
    texture->imageData[cswap] ^= texture->imageData[cswap+2];
}

```

Obrázek jsme úspěšně nahráli, takže zavřeme soubor a vrácením true oznámíme úspěch.

```

fclose(fTGA);// Zavření souboru
return true;// Úspěch

// Paměť dat obrázku se uvolňuje až po vytvoření textury
}

```

Nyní přistoupíme k nahrávání obrázku komprimovaného metodou RLE (RunLength Encoded). Začátek je stejný jako u nekomprimovaného obrázku - načteme výšku, šířku a barevnou hloubku, ošetříme neplatné hodnoty a spočítáme velikost potřebné paměti, kterou opět alokujeme. Všimněte si, že velikost požadované paměti je taková, aby do ní mohla být uložena data PO DEKOMPRIMOVÁNÍ, ne před dekomprimováním.

```

bool LoadCompressedTGA(Texture* texture, char* filename, FILE* fTGA)// Nahraje
komprimovaný obrázek
{
    if(fread(tga.header, sizeof(tga.header), 1, fTGA) == 0)// Šest užitečných bytů
    {
        MessageBox(NULL, "Could not read info header", "ERROR", MB_OK);

        if(fTGA != NULL)
        {
            fclose(fTGA);
        }

        return false;
    }
}

```

```

}

texture->width = tga.header[1] * 256 + tga.header[0]; // Šířka
texture->height = tga.header[3] * 256 + tga.header[2]; // Výška
texture->bpp = tga.header[4]; // Barevná hloubka v bitech

// Kopírování dat do struktury obrázku
tga.Width = texture->width;
tga.Height = texture->height;
tga.Bpp = texture->bpp;

// Platné hodnoty?
if((texture->width <= 0) || (texture->height <= 0) || ((texture->bpp != 24) &&
(texture->bpp != 32)))
{
    MessageBox(NULL, "Invalid texture information", "ERROR", MB_OK);
    if(ftGA != NULL)
    {
        fclose(ftGA);
    }

    return false;
}

if(texture->bpp == 24) // 24 bitový obrázek?
{
    texture->type = GL_RGB;
}
else // 32 bitový obrázek
{
    texture->type = GL_RGBA;
}

tga.bytesPerPixel = (tga.Bpp / 8); // BYTY na pixel
tga.imageSize = (tga.bytesPerPixel * tga.Width * tga.Height); // Velikost paměti

texture->imageData = (GLubyte *)malloc(tga.imageSize); // Alokace paměti pro data (po
dekomprimování)

if(texture->imageData == NULL) // Alokace neúspěšná
{
    MessageBox(NULL, "Could not allocate memory for image", "ERROR", MB_OK);
    fclose(ftGA);
    return false;
}

```

Dále potřebujeme zjistit přesný počet pixelů, ze kterých je obrázek složen. Jednoduše vynásobíme výšku obrázku se šířkou. Také musíme znát, na kterém pixelu se právě nacházíme a kam do paměti zapisujeme.

```

GLuint pixelcount = tga.Height * tga.Width; // Počet pixelů
GLuint currentpixel = 0; // Aktuální načítaný pixel
GLuint currentbyte = 0; // Aktuální načítaný byte

```

Alokujeme pomocné pole tří nebo čtyř bytů (podle barevné hloubky) k uložení jednoho pixelu. Překl.: Měla by se testovat správnost alokace paměti!

```

GLubyte* colorbuffer = (GLubyte *)malloc(tga.bytesPerPixel); // Paměť pro jeden pixel

// Překl.: Test úspěšnosti alokace paměti !!!
// if(colorbuffer == NULL) // Alokace neúspěšná
// {
//     // MessageBox(NULL, "Could not allocate memory for color buffer", "ERROR",
//     // MB_OK);
//     // fclose(ftGA);
//     // return false;
// }

```

V hlavním cyklu deklaruujeme proměnnou k uložení bytu hlavičky, který definuje, jestli je následující sekce obrázku ve formátu RAW nebo RLE a jak dlouhá je. Pokud je byte hlavičky menší nebo roven 127, jedná se o RAW hlavičku. Hodnota, v ní uložená, určuje počet pixelů mínus jedna, které vzápětí načteme a zkopírujeme do paměti. Po těchto pixelech se v souboru vyskytuje další byte hlavičky. Pokud je byte hlavičky větší než 127, představuje toto číslo (zmenšené o 127), kolikrát se má následující pixel v dekomprimovaném obrázku opakovat. Hned po něm se bude

vyskytovat další hlavičkový byte. Načteme hodnoty tohoto pixelu a zkopírujeme ho do imageData tolikrát, kolikrát potřebujeme.

Podstatu komprese RLE tedy už znáte, podívejme se na kód. Jak jsem již zmínil, založíme cyklus přes celý soubor a pokusíme se načíst byte první hlavičky.

```
do// Prochází celý soubor
{
    GLubyte chunkheader = 0;// Byte hlavičky

    if(fread(&chunkheader, sizeof(GLubyte), 1, fTGA) == 0)// Načte byte hlavičky
    {
        MessageBox(NULL, "Could not read RLE header", "ERROR", MB_OK);

        if(fTGA != NULL)
        {
            fclose(fTGA);
        }

        if(texture->imageData != NULL)
        {
            free(texture->imageData);
        }

        // Překl.: Uvolnění dynamické paměti !!!
        // if(colorbuffer != NULL)
        // {
        //     // free(colorbuffer);
        // }

        return false;
    }
}
```

Pokud se jedná o RAW hlavičku, přičteme k bytu jedničku, abychom získali počet pixelů následujících po hlavičce. Potom založíme další cyklus, který načítá všechny požadované pixely do pomocného pole colorbuffer a vzápětí je ve správném formátu ukládá do imageData.

```
if(chunkheader < 128)// RAW část obrázku
{
    chunkheader++;// Počet pixelů v sekci před výskytem dalšího bytu hlavičky

    for(short counter = 0; counter < chunkheader; counter++)// Jednotlivé pixely
    {
        // Načítání po jednom pixelu
        if(fread(colorbuffer, 1, tga.bytesPerPixel, fTGA) != tga.bytesPerPixel)
        {
            MessageBox(NULL, "Could not read image data", "ERROR", MB_OK);

            if(fTGA != NULL)
            {
                fclose(fTGA);
            }

            if(colorbuffer != NULL)
            {
                free(colorbuffer);
            }

            if(texture->imageData != NULL)
            {
                free(texture->imageData);
            }

            return false;
        }
    }
}
```

Při kopírování do imageData prohodíme pořadí bytů z formátu BGR na RGB. Pokud je v obrázku i alfa kanál, zkopírujeme i čtvrtý byte. Abychom se přesunuli na další pixel popř. byte hlavičky, zvětšíme aktuální byte o barevnou hloubku (+3 nebo +4). Inkrementujeme také počet načtených pixelů.

```
// Zápis do paměti, prohodí R a B složku barvy
texture->imageData[currentbyte] = colorbuffer[2];
texture->imageData[currentbyte + 1] = colorbuffer[1];
```

```

texture->imageData[currentbyte + 2] = colorbuffer[0];
if(tga.bytesPerPixel == 4)// 32 bitový obrázek?
{
    texture->imageData[currentbyte + 3] = colorbuffer[3];// Kopírování
    alfy
}

currentbyte += tga.bytesPerPixel;// Aktualizuje byte
currentpixel++;// Přesun na další pixel

```

Zjistíme, jestli je pořadová číslo aktuálního pixelu větší než celkový počet pixelů. Pokud ano, je soubor obrázku poškozen nebo je v něm někde chyba. Jak jsme na to přišli? Máme načítat další pixel, ale defakto je už máme všechny načtené, protože aktuální hodnota je větší než maximální. Nestáčila by alokovaná paměť pro dekomprimovanou verzi obrázku. Tuto skutečnost musíme každopádně ošetřit.

```

if(currentpixel > pixelcount)// Jsme za hranicí obrázku?
{
    MessageBox(NULL, "Too many pixels read", "ERROR", NULL);

    if(ftGA != NULL)
    {
        fclose(ftGA);
    }

    if(colorbuffer != NULL)
    {
        free(colorbuffer);
    }

    if(texture->imageData != NULL)
    {
        free(texture->imageData);
    }

    return false;
}
}
}

```

Vyřešili jsme část RAW, nyní implementujeme sekci RLE. Ze všeho nejdříve od bytu hlavičky odečteme číslo 127, abychom získali kolikrát se má následující pixel opakovat.

```

else// RLE část obrázku
{
    chunkheader -= 127;// Počet pixelů v sekci

```

Načteme jeden pixel po hlavičce a potom ho požadovaně-krát vložíme do imageData. Opět zaměňujeme formát BGR za RGB. Stejně jako minule inkrementujeme aktuální byte i pixel a ošetřujeme přetečení.

```

if(fread(colorbuffer, 1, tga.bytesPerPixel, ftGA) != tga.bytesPerPixel)//
Načte jeden pixel
{
    MessageBox(NULL, "Could not read from file", "ERROR", MB_OK);

    if(ftGA != NULL)
    {
        fclose(ftGA);
    }

    if(colorbuffer != NULL)
    {
        free(colorbuffer);
    }

    if(texture->imageData != NULL)
    {
        free(texture->imageData);
    }

    return false;
}

```

```

for(short counter = 0; counter < chunkheader; counter++)// Kopírování pixelu
{
    // Zápis do paměti, prohodí R a B složku barvy
    texture->imageData[currentbyte] = colorbuffer[2];
    texture->imageData[currentbyte + 1] = colorbuffer[1];
    texture->imageData[currentbyte + 2] = colorbuffer[0];

    if(tga.bytesPerPixel == 4)// 32 bitový obrázek?
    {
        texture->imageData[currentbyte + 3] = colorbuffer[3];// Kopírování
        alfy
    }

    currentbyte += tga.bytesPerPixel;// Aktualizuje byte
    currentpixel++;// Přesun na další pixel

    if(currentpixel > pixelcount)// Jsme za hranicí obrázku?
    {
        MessageBox(NULL, "Too many pixels read", "ERROR", NULL);

        if(ftGA != NULL)
        {
            fclose(ftGA);
        }

        if(colorbuffer != NULL)
        {
            free(colorbuffer);
        }

        if(texture->imageData != NULL)
        {
            free(texture->imageData);
        }

        return false;
    }
}
}
}

```

Hlavní cyklus opakujeme tak dlouho, dokud v souboru zbývají nenačtené pixely. Po konci loadingu soubor zavřeme a vrácením true indikujeme úspěch.

```

} while(currentpixel < pixelcount);// Pokračuj dokud zbývají pixely

// Překl.: Uvolnění dynamické paměti !!!
// if(colorbuffer != NULL)
// {
//     // free(colorbuffer);
// }

fclose(ftGA);// Zavření souboru
return true;// Úspěch

// Paměť dat obrázku se uvolňuje až po vytvoření textury
}

```

Nyní jsou data obrázku připravena pro vytvoření textury a to už jistě zvládnete sami. V tomto tutoriálu nám šlo především o nahrávání TGA obrázků. Ukázkové demo bylo vytvořeno jen proto, abyste viděli, že kód opravdu funguje.

A jak je to s úspěšností komprimace metody RLE? Je jasné, že nejmenší paměť bude zabírat obrázek s rozsáhlými plochami stejných pixelů (na řádcích). Pokud chcete čísla, tak si vezmeme na pomoc obrázku použité v tomto demu: oba jsou 128x128 pixelů veliké, nekomprimovaný zabírá na disku 48,0 kB a komprimovaný pouze 5,29 kB. Na obou je sice něco jiného, ale devítinásobné zmenšení velikosti mluví za vše.

**napsal: Evan Piphon - Terminate <terminate (zavináč) gdnmail.net>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>**

Lekce 34 - Generování terénů a krajin za použití výškového mapování textur

Chtěli byste vytvořit věrnou simulaci krajiny, ale nevíte, jak na to? Bude nám stačit obyčejný 2D obrázek ve stupních šedi, pomocí kterého deformujeme rovinu do třetího rozměru. Na první pohled těžko řešitelné problémy bývají častokrát velice jednoduché.

Nyní byste už měli být opravdovými experty na OpenGL, ale možná nevíte, co to je výškové mapování (height mapping). Představte si rovinu, vytlačenou podle nějaké formy do 3D prostoru. Této formě se říká výšková mapa, kterou může být defakto jakýkoli typ dat. Obrázky, textové soubory nebo třeba datový proud zvuku - záleží jen na vás. My budeme používat .RAW obrázek ve stupních šedi.

Definujeme tři opravdu důležité symbolické konstanty. MAP_SIZE představuje rozměr mapy, v našem případě se jedná o šířku/výšku obrázku (1024x1024). Konstanta STEP_SIZE určuje velikost kroků při grabování hodnot z obrázku. V současné chvíli bereme v úvahu každý šestnáctý pixel. Zmenšením čísla přidáváme do výsledného povrchu polygony, takže vypadá méně hranatě, ale zároveň zvyšujeme náročnost na rendering. HEIGHT_RATIO slouží jako měřítko výšky na ose y. Malé číslo zredukuje vysoké hory s údolími na plochou rovinu.

```
#define MAP_SIZE 1024// Velikost .RAW obrázku výškové mapy
#define STEP_SIZE 16// Hustota grabování pixelů
#define HEIGHT_RATIO 1.5f// Zoom výšky terénu na ose y
```

Proměnná bRender představuje přepínač mezi pevnými polygony a drátěným modelem, scaleValue určuje zoom scény na všech třech osách.

```
bool bRender = TRUE;// Polygony - true, drátěný model - false
float scaleValue = 0.15f;// Měřítko velikosti terénu (všechny osy)
```

Deklarujeme jednorozměrné pole pro uložení všech dat výškové mapy. Používaný .RAW obrázek neobsahuje RGB složky barvy, ale každý pixel je tvořen jedním bytem, který specifikuje jeho odstín. Nicméně o barvu se starat nebudeme, jde nám především o hodnoty. Číslo 255 bude představovat nejvyšší možný bod povrchu a nula nejnižší.

```
BYTE g_HeightMap[MAP_SIZE * MAP_SIZE];// Ukládá data výškové mapy
```

Funkce LoadRawFile() nahrává RAW soubor s obrázkem. Nic komplexního! V parametrech se jí předává řetězec diskové cesty, velikost dat obrázku a ukazatel na paměť, do které se ukládá. Otevřeme soubor pro čtení v binárním módu a ošetříme situaci, kdy neexistuje.

```
void LoadRawFile(LPSTR strName, int nSize, BYTE* pHeightMap)// Nahraje .RAW soubor
{
    FILE *pFile = NULL;// Handle souboru

    pFile = fopen(strName, "rb");// Otevření souboru pro čtení v binárním módu

    if (pFile == NULL)// Otevření v pořádku?
    {
        MessageBox(NULL, "Can't Find The Height Map!", "Error", MB_OK);
        return;
    }
}
```

Pomocí fread() načteme po jednom bytu ze souboru pFile data o velikosti nSize a uložíme je do paměti na lokaci pHeightMap. Vyskytne-li se chyba, vypíšeme varovnou zprávu.

```
fread(pHeightMap, 1, nSize, pFile);// Načte soubor do paměti

int result = ferror(pFile);// Výsledek načítání dat

if (result)// Nastala chyba?
{
    MessageBox(NULL, "Failed To Get Data!", "Error", MB_OK);
}
```

Na konci zbývá už jenom zavřít soubor.

```
fclose(pFile);// Zavření souboru
}
```

Kód pro inicializaci OpenGL byste měli bez problémů pochopit sami.

```
int InitGL(GLvoid) // Inicializace OpenGL
{
    glShadeModel(GL_SMOOTH); // Jemné stínování
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glEnable(GL_DEPTH_TEST); // Zapne testování hloubky
    glDepthFunc(GL_LEQUAL); // Typ testování hloubky
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Perspektivní korekce
}
```

Před vrácením true ještě do `g_HeightMap` nahrajeme `.RAW` obrázek.

```
LoadRawFile("Data/Terrain.raw", MAP_SIZE * MAP_SIZE, g_HeightMap); // Načtení dat
výškové mapy

return TRUE; // Vše v pořádku
}
```

Máme zde jeden problém - uložili jsme dvourozměrný obrázek do jednorozměrného pole. Co s tím? Funkce `Height()` provede výpočet pro transformaci `x, y` souřadnic na index do tohoto pole a vrátí hodnotu, která je na něm uložena. Při práci s poli bychom se vždy měli start o možnost přetečení paměti. Jednoduchým trikem zmenšíme vysoké hodnoty tak, aby byly vždy platné. Pokud některá z hodnot přesáhne daný index, zbytek po dělení ji zmenší do rozmezí, které můžeme bez obav použít. Dále otestujeme, jestli se v poli opravdu nacházejí data.

```
int Height(BYTE *pHeightMap, int X, int Y) // Přepočítá 2D souřadnice na 1D a vrátí
uloženou hodnotu
{
    int x = X % MAP_SIZE; // Protí přetečení paměti
    int y = Y % MAP_SIZE;

    if(!pHeightMap) // Obsahuje paměť data?
    {
        return 0;
    }
}
```

Aby se jednorozměrné pole chovalo jako dvojrozměrné, musíme zapojit trochu matematiky. Index do 1D pole na 2D souřadnicích získáme tak, že vynásobíme řádek (`y`) jeho šířkou (`MAP_SIZE`) a přičteme konkrétní pozici na řádku (`x`).

```
return pHeightMap[(y * MAP_SIZE) + x]; // Vrátí hodnotu z pole
}
```

Na tomto místě nastavujeme barvu vertexu podle aktuální výšky nad `height mapou`. Získáme hodnotu na indexu pole a dělením `256.0f` ji zmenšíme do rozmezí `0.0f` až `1.0f`. Abychom ji ještě trochu ztmavili, odečteme `-0.15f`. Výsledek předáme funkci `glColor3f()` jako modrou složku barvy.

```
void SetVertexColor(BYTE *pHeightMap, int x, int y) // Získá barvu v závislosti na výšce
{
    if(!pHeightMap) // Obsahuje paměť data?
    {
        return;
    }

    // Získání hodnoty, přepočet do rozmezí 0.0f až 1.0f, ztmavení
    float fColor = (Height(pHeightMap, x, y) / 256.0f) - 0.15f;

    glColor3f(0, 0, fColor); // Odstíny modré barvy
}
```

Dostáváme se k nejpodstatnější části celého tutoriálu - renderování terénu. Proměnné `X, Y` slouží k procházení výškové mapy a `x, y, z` jsou 3D souřadnicemi vertexu.

```
void RenderHeightMap(BYTE pHeightMap[]) // Renderuje terén
{
    int X = 0, Y = 0; // Pro procházení polem
    int x, y, z; // Souřadnice vertexů

    if(!pHeightMap) // Obsahuje paměť data?
    {
        return;
    }
}
```


Podle logické hodnoty bRender přepínáme mezi vykreslováním obdélníků a linek.

```
if(bRender)// Co chce uživatel renderovat?
{
    glBegin(GL_QUADS);// Polygony
}
else
{
    glBegin(GL_LINES);// Drátěný model
}
```

Založíme dva vnořené cykly, které procházejí jednotlivé pixely výškové mapy. Vnější se stará o osu x a vnitřní o osu y, z čehož plyne, že vykreslujeme po sloupcích a ne po řádcích. Všimněte si, že po každém průchodu nezvětšujeme řídicí proměnnou o jeden pixel, ale hned o několik najednou. Sice výsledný terén nebude tak hladký a přesný, ale díky menšímu počtu polygonů se rendering urychlí. Pokud by se STEP_SIZE rovnalo jedné, každému pixelu by se přiřadil jeden polygon. Myslím, že číslo šestnáct bude vyhovující, ale pokud zapnete světla, které zvýrazňují hrnatost povrchu, měli byste ho snížit.

Překl.: Úplně nejlepší by bylo, kdyby se velikost kroku určovala před vstupem do cyklů podle aktuálního FPS. Následující ukázkový kód zavádí zpětnovazební regulační smyčku.

```
// Překl.: Regulace počtu polygonů
// if(FPS < 30)// Nižší hodnoty => viditelné trhání pohybů animace
// {
//     // if(STEP_SIZE > 1)// Dolní mez (1 pixel)
//     // {
//         // STEP_SIZE--;// Musí být proměnnou a ne symbolickou konstantou
//     // }
// }
// else
// {
//     // if(STEP_SIZE < MAP_SIZE-1)// Horní mez (velikost výškové mapy)
//     // {
//         // STEP_SIZE++;// Musí být proměnnou a ne symbolickou konstantou
//     // }
// }

for (X = 0; X < MAP_SIZE; X += STEP_SIZE)// Řádky výškové mapy
{
    for (Y = 0; Y < MAP_SIZE; Y += STEP_SIZE)// Sloupce výškové mapy
    {
```

Přepokládám, že to, jak určit pozici vertexu, jste už dávno vytušili. Hodnota na ose x odpovídá x-ové souřadnici výškové mapy a na ose z y-ové. Získali jsme umístění bodu na rovině, potřebujeme ho ještě vyzdvihnout do výšky, které v OpenGL odpovídá osa y. Tato výška je definována hodnotou uloženou na daném prvku pole (světlostí obrázku). Opravdu nic složitého...

```
// Souřadnice levého dolního vertexu
x = X;
y = Height(pHeightMap, X, Y );
z = Y;
```

Určíme barvu bodu podle výšky nad rovinou. Čím výše se nachází, tím bude světlejší. Potom pomocí funkce glVertex3i () předáme OpenGL souřadnice vertexu.

```
SetVertexColor(pHeightMap, x, z);// Barva vertexu
glVertex3i(x, y, z);// Definování vertexu
```

Druhý vertex určíme přičtením STEP_SIZE k ose z. Na tomto místě se budeme nacházet při příštím průchodu cyklem, takže se mezi jednotlivými polygony nebudou vyskytovat mezery. Analogicky získáme i další dva body obdélníku. Nyní mi už věříte, když jsem na začátku tutoriálu psal, že složitě vypadající věci bývají často velice jednoduché?

```
// Souřadnice levého horního vertexu
x = X;
y = Height(pHeightMap, X, Y + STEP_SIZE );
z = Y + STEP_SIZE ;

SetVertexColor(pHeightMap, x, z);// Barva vertexu
glVertex3i(x, y, z);// Definování vertexu

// Souřadnice pravého horního vertexu
```

```

x = X + STEP_SIZE;
y = Height(pHeightMap, X + STEP_SIZE, Y + STEP_SIZE );
z = Y + STEP_SIZE ;

SetVertexColor(pHeightMap, x, z); // Barva vertexu
glVertex3i(x, y, z); // Definování vertexu

// Souřadnice pravého dolního vertexu
x = X + STEP_SIZE;
y = Height(pHeightMap, X + STEP_SIZE, Y );
z = Y;

SetVertexColor(pHeightMap, x, z); // Barva vertexu
glVertex3i(x, y, z); // Definování vertexu
}
}

glEnd(); // Konec kreslení

```

Po vykreslení terénu reinitializujeme barvu na bílou, abychom neměli starosti s barvou ostatních objektů ve scéně (netýká se tohoto dema).

```

glColor4f(1.0f, 1.0f, 1.0f, 1.0f); // Reset barvy
}

```

Na začátku DrawGLScene() začneme klasicky smazáním bufferů a resetem matice.

```

int DrawGLScene(GLvoid) // Vykreslení OpenGL scény
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Vymaže buffery
    glLoadIdentity(); // Reset matice

```

Pomocí funkce gluLookAt() umístíme a natočíme kameru tak, aby byl renderovaný terén v záběru. První tři parametry určují její pozici vzhledem k počátku souřadnicového systému, další tři body reprezentují místo, kam je natočená a poslední tři představují vektor vzhůru. V našem případě se nacházíme nad sledovaným terénem a díváme se na něj trochu dolů (55 je menší než 60) spíše doleva (186 je menší než 212). Hodnota 171 představuje vzdálenost od kamery na ose z. Protože se hory zvedají od zdola nahoru, nastavíme u vektoru vzhůru jedničku na ose y. Ostatní dvě hodnoty zůstanou na nule.

Při prvním použití může být gluLookAt() trochu odstrašující, asi jste zmateni. Nejlepší radou je pohrát si se všemi hodnotami, abyste viděli, jak se pohled na scénu postupně mění. Pokud byste například přepsal pozici z 60 na 120, viděli byste terén spíše shora než z boku, protože se stále díváte na souřadnice 55.

Praktický příklad: Řekněme, že jste vysoký kolem 1,8 m. Oči, které reprezentují kameru, jsou trochu níže - 1,7 m. Stojíte před stěnou, která je vysoká pouze 1 m, takže bez problémů vidíte její horní stranu. Pokud ale zedníci dostaví stěnu do výšky tří metrů, budete se muset dívat VZHŮRU, ale její vrch už NEUVIDÍTE. Výhled se změnil podle toho, jestli se díváte dolů nebo vzhůru (respektive jestli jste nad nebo pod objektem).

```

// Umístění a natočení kamery
gluLookAt(212, 60, 194, 186, 55, 171, 0, 1, 0); // Pozice, směr, vektor vzhůru

```

Aby byl výsledný terén poněkud menší, změním měřítko souřadnicových os. Protože navíc násobíme y-ovou hodnotu, budou se hory jevit vyšší. Mohli bychom také použít translace a rotace, ale to už nechám na vás.

```

glScalef(scaleValue, scaleValue * HEIGHT_RATIO, scaleValue); // Zoom terénu

```

Pomocí dříve napsané funkce vyrenderujeme terén.

```

RenderHeightMap(g_HeightMap); // Renderování terénu
return TRUE; // Vše v pořádku
}

```

Kliknutím levého tlačítka myši může uživatel přepnout mezi renderováním polygonů a linek (drátěný model).

```

// Funkce WndProc()
case WM_LBUTTONDOWN: // Levé tlačítko myši
{
    bRender = !bRender; // Přepne mezi polygony a drátěným modelem
    return 0; // Konec funkce
}

```

Šipkami nahoru a dolů zvětšujeme/zmenšujeme měřítko scény a tím i velikost terénu.

```
// Funkce WinMain()
    if (keys[VK_UP])// Šipka nahoru
    {
        scaleValue += 0.001f;// Vyvýší hory
    }

    if (keys[VK_DOWN])// Šipka dolů
    {
        scaleValue -= 0.001f;// Sníží hory
    }
```

Tak to je všechno, výškovým mapováním textur jsme naprogramovali nádherou krajinu, která je ale zabarvená do modra. Zkuste si nakreslit texturu (letecký pohled), která reprezentuje zasněžené vrcholy hor, louky, jezera a podobně a namapujte ji na terén. Texturovací koordináty získáte vydělením pozice na rovině rozměrem obrázku (zmenšení hodnot do rozsahu 0.0f až 1.0f). Plazmovými efekty a rolováním se může krajina dynamicky měnit. Déšť a sníh zajistí částicové systémy, které už také znáte. Vložíte-li krajinu do skyboxu, nikdo nepozná, že se jedná o počítačový model a ne o video animaci.

Nebo můžete vytvořit mořskou hladinu s vlnami, na kterých se pohupuje uplavaný míč (výšku nad mořským dnem přece znáte - hodnota na indexu v poli). Nechte uživatele, ať ho může ovládat. Možnosti jsou bez hranic...

napsal: Ben Humphrey - DigiBen
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 35 - Přehrávání videa ve formátu AVI

Přehrávání AVI videa v OpenGL? Na pozadí, povrchu krychle, koule, či válce, ve fullscreenu nebo v obyčejném okně. Co víc si přát...

Na začátku bych chtěl poděkovat Fredsterovi za AVI animaci, Maxwellu Saylesovi za rady při programování, Jonathanu Nixovi a Johnu F. MCGowanovi, Ph. D. za skvělé články/dokumenty o AVI formátu. Moc jste mi pomohli.

Musím říci, že jsem na tento tutoriál opravdu pyšný. když mě Jonathan F. Blok přivedl na nápad AVI přehrávače v OpenGL, neměl jsem nejmenší potuchu, jak AVI otevřít, natož jak by mohl video přehrávač fungovat. Začal jsem listováním ve svých knihách o programování - vůbec nic. Poté jsem zkusil MSDN. Našel jsem spoustu užitečných informací, ale bylo jich potřeba mnohem, mnohem více. Po hodinách prolézání internetu jsem měl poznamenány pouze dva weby. Nemohu říct, že moje vyhledávací postupy jsou úplně nejlepší, ale v cca. 99,9% případů jsem nikdy neměl nejmenší problémy. Byl jsem absolutně šokován, když jsem zjistil, jak málo příkladů na přehrávání videa tam bylo. Většina z nich navíc nešla zkompileovat, některé byly komplexní (alespoň pro mě) a plnily svůj účel, nicméně byly programovány ve VB, Delphi nebo podobně (ne VC++).

První z užitečných stránek, které jsem našel, byl článek od Janathana Nixe nadepsaný **AVI soubory**. Jonathan má u mě obrovský respekt za tak extrémně brilantní dokument. Ačkoli jsem se rozhodl jít jinou cestou než on, vnesl mě do problematiky. Druhý web, tentokrát od Johna F. MCGowana, Ph. D., má titulky The AVI Overview. Mohl bych teď začít popisovat, jak úžasné jsou Johnovi stránky, ale snadnější bude, když se **sami podíváte**. Soustředil na nich snad vše, co je o AVI známo.

Poslední věcí, na kterou chci upozornit, je, že žádná část z celého kódu NEBYLA vypůjčena a nic nebylo okopírováno. Kódování mi zabralo plně tři dny, používal jsem pouze informace z výše uvedených zdrojů. Zároveň cítím, že by bylo vhodné poznamenat, že můj kód nemusí být nejlepším způsobem pro přehrávání AVI souborů. Dokonce nemusí být ani vhodnou cestou, ale funguje a snadno se používá. Nicméně pokud se vám můj styl a kód nelíbí, nebo cítíte-li, že uvolněním tohoto tutoriálu dokonce zraňuji programátorskou komunitu, máte několik možností: 1) zkuste si na internetu najít jiné zdroje, 2) napište si svůj vlastní AVI přehrávač nebo 3) napište lepší tutoriál. Každý, kdo navštíví tento web, by měl vědět, že kóduji pro zábavu. Hlavním účelem těchto stránek je ulehčit život ne-elitním programátorům, kteří začínají s OpenGL. Tutoriály ukazují, jak jsem ljál dokázal vytvořit specifický efekt... nic více, nic méně.

Pojďme ale ke kódu. Jako první věc vložíme a přilinkujeme knihovnu Video For Windows. Obrovské díky MicrosoftŽu (Nikdy bych nevěřil, že to řeknu). Pomocí této knihovny bude otevírání a přehrávání AVI pouhou banalitou.

```
#include <vfw.h> // Hlavičkový soubor knihovny Video pro Windows
#pragma comment(lib, "vfw32.lib") // Přilinkování VFW32.lib
```

Deklarujeme proměnné. Angle je úhel natočení zobrazovaného objektu. Next představuje celé číslo, které použijeme pro spočítání množství uplynulého času (v milisekundách), abychom mohli udržet framerate na správné hodnotě. Více o tomto dále. Frame bude samozřejmě obsahovat číslo aktuálně zobrazovaného snímku animace. Effect představuje druh objektu na obrazovce (krychle, koule, válec, žádný). Bude-li env rovno true, budou se automaticky generovat texturové souřadnice. Bg představuje flag, který definuje, jestli se má pozadí zobrazovat nebo ne. Sp, ep a bp slouží pro ošetření delšího stisku kláves.

```
float angle; // Úhel rotace objektu
int next; // Pro animaci
int frame = 0; // Aktuální snímek videa
int effect; // Zobrazený objekt

bool env = TRUE; // Automaticky generovat texturové koordináty?
bool bg = TRUE; // Zobrazovat pozadí?

bool sp; // Stisknut mezerník?
bool ep; // Stisknuto E?
bool bp; // Stisknuto B?
```

Struktura psi bude udržovat informace o AVI souboru. Pavi představuje ukazatel na buffer, do kterého po otevření AVI obdržíme handle nového proudu. Pgf, pointer na objekt GetFrame, použijeme pro získávání jednotlivých snímků, které pomocí bmih zkonvertujeme do formátu, který potřebujeme pro vytvoření textury. Lastframe ukládá číslo posledního snímku animace. Width a height definují rozměry AVI proudu, pdata je ukazatel na data obrázku vrácené po požadavku na snímek. Mpf (Milliseconds Per Frame) použijeme pro výpočet doby zobrazení snímku. Předpokládám, že nemáte nejmenší ponětí, k čemu všechny tyto proměnné vlastně slouží... vše byste měli pochopit dále.

```
AVIStreamInfo psi; // Informace o datovém proudu videa
PAVStream pavi; // Handle proudu
PGETFrame pgf; // Ukazatel na objekt GetFrame
```

```

BITMAPINFOHEADER bmih;// Hlavička pro DrawDibDraw dekódování

long lastframe;// Poslední snímek proudu
int width;// Šířka videa
int height;// Výška videa
char* pdata;// Ukazatel na data textury
int mpf;// Doba zobrazení jednoho snímku (Milliseconds Per Frame)

```

Pomocí knihovny GLU budeme moci vykreslit dva quadratic útvary, koule a válec. Hdd je handle na DIB (Device Independent Bitmap) a hdc je handle na kontext zařízení. HBitmap představuje handle na bitmapu závislou na zařízení (DDB - Device Dependent Bitmap), použijeme ji dále při konverzích. Data je pointer, který bude ukazovat na data obrázku použitelná pro vytvoření textury. Opět - více pochopíte dále.

```

GLUquadricObj *quadratic;// Objekt quadraticu

HDRAWDIB hdd;// Handle DIBu
HBITMAP hBitmap;// Handle bitmapy závislé na zařízení
HDC hdc = CreateCompatibleDC(0);// Kontext zařízení
unsigned char* data = 0;// Ukazatel na bitmapu o změněné velikosti

```

Nyní malý úvod do jazyka Assembler (ASM). Pokud jste ho ještě nikdy dříve nepoužili, nelekejte se. Může vypadat složitě, ale vše je velmi jednoduché. Při programování tohoto tutoriálu jsem se dostal před velký problém. Aplikace běžela v pořádku, ale barvy byly divné. Vše, co mělo být červené bylo modré, a vše co mělo být modré bylo červené - klasické prohození R a B složky pixelů. Byl jsem absolutně šokovaný. Myslel jsem si, že jsem v kódu udělal nějakou šílenou chybu typu "čárka sem, znaménko tam...". Po pečlivém prostudování všeho, co jsem do té doby napsal, jsem nebyl schopen bug najít. Začal jsem znovu pročítat MSDN. Proč byla červená a modrá složka barvy prohozená?! V MSDN bylo přece jasně napsáno, že 24 bitové bitmapy jsou ve formátu RGB!!! Po spoustě dalšího čtení jsem problém objevil. Ve Windows se RGB data ukládají pozpátku a RGB uložené pozpátku je přeci BGR! Takže si jednou pro vždy zapamatujte, že v OpenGL RGB znamená RGB a ve Windows RGB znamená BGR - jak jednoduché.

Po stížnostech od fanoušků Microsoftu (Překl.: Ono něco takového existuje?!): Rozhodl jsem se přidat krátké vysvětlení... Nepomlouvám Microsoft kvůli tomu, že označil BGR formát barvy za RGB. Jestli se mu převrácená zkratka líbí více, ať si ji používá. Nicméně nalezení chyby může být pro cizího programátora velice frustrující (zvláště když žádná neexistuje).

Blue přidal: Má to co dělat s konvencemi little endian a big endian. Intel a Intel kompatibilní systémy používají little endian, u kterého se méně významné byty ukládají dříve než více významné. Specifikaci OpenGL vytvořila firma SGI (Silicon Graphic), jejíž systémy pravděpodobně používají big endian, a tudíž OpenGL standardně vyžadují bitmapy ve formátu big endian.

Skvělý! Takže jsem vytvořil přehrávač, který je absolutně k ničemu (Překl.: v originále absolute crap - zkuste si toto slovo najít ve slovníku, já chci být slušný :-). Prvním řešením, které mě napadlo, bylo prohodit byty manuálně pomocí cyklu for. Pracovalo to v pořádku, ale strašně pomalu. Měl jsem všeho po krk. Zkusil jsem modifikoval generování textury na GL_BGR_EXT místo GL_RGB. Obrovský nárůst rychlosti a barvy vypadají skvěle! Takže jsem problém konečně vyřešil... alespoň jsem si to myslel. Některé OpenGL ovladače mají s GL_BGR_EXT problémy :(Maxwell Sayles mi doporučil prohození bytů pomocí ASM. O minutku později mi ICQ-oval kód uvedený níže, který je rychlý a plní dokonale svou funkci.

Každý snímek animace se ukládá do bufferu, obrázek má vždy čtvercovou velikost 256 pixelů a 3 barevné složky ve formátu BGR (speciálně pro Billa Gatese: RGB). Funkce flipIt() prochází tento buffer po tří bytových krocích a zaměňuje červenou složku za modrou. R má být uloženo na pozici abx+0 a B na abx+2. Cyklus se opakuje tak dlouho, dokud nejsou všechny pixely ve formátu RGB.

Předpokládám, že většina z vás není z ASM moc nadšená. Jak už jsem psal, původně jsem plánoval použít GL_BGR_EXT. Funguje, ale ne na všech kartách. Potom jsem se rozhodl jít cestou minulých tutoriálů a swapovat byty pomocí bitových operací XOR, které pracují na všech počítačích, ale ne extrémně rychle. Dokud jsme nepracovali s real-time videem, stačily, ale tentokrát potřebujeme co možná nejrychlejší metodu. Zvážíme-li všechny možnosti, je ASM podle mého názoru nejlepší volbou. Pokud máte ještě lepší způsob, prosím... POUŽIJTE HO! Neříkám vám, jak co MÁTE dělat, já pouze ukazuji, jak jsem problémy vyřešil já. Vše proto také vysvětluji do detailů, abyste můj kód, pokud znáte lepší, mohli nahradit.

```

void flipIt(void* buffer)// Prohodí červenou a modrou složku pixelů v obrázku
{
    void* b = buffer;// Ukazatel na buffer

    __asm // ASM kód
    {
        mov ecx, 256*256 // Řídící "proměnná" cyklu
        mov ebx, b // Ebx ukazuje na data

        label: // Návěští pro cyklus
        mov al, [ebx+0] // Přesune B složku do al
    }
}

```

```

    mov ah, [ebx+2] // Přesune R složku do ah
    mov [ebx+2], al // Vloží B na správnou pozici
    mov [ebx+0], ah // Vloží R na správnou pozici

    add ebx, 3 // Přesun na další tři byty
    dec ecx // Dekrementuje čítač
    jnz label // Pokud se čítač nerovná nule skok na návěští
}
}

```

Jak už z názvu funkce OpenAVI() vyplývá, otevírá AVI soubor. Parametr szFile je řetězec s diskovou cestou k souboru. Řetězec title použijeme pro zobrazení informací o AVI do titulku okna.

```

void OpenAVI(LPCSTR szFile)// Otevře AVI soubor
{
    TCHAR title[100];// Pro vypsání textu do titulku okna

```

Abychom inicializovali knihovnu AVI file, zavoláme AVIFileInit(). Existuje mnoho způsobů, jak otevřít video soubor. Rozhodl jsem se použít AVIStreamOpenFromFile(), která otevře jeden datový proud. Pavi představuje ukazatel na buffer, kam funkce vrací handle nového proudu, szFile označuje diskovou cestu k souboru. Třetí parametr určuje typ proudu, který si přejeme otevřít. V tomto projektu nás zajímá pouze video. Nula, další parametr, oznamuje, že se má použít první výskyt proudu streamtypeVIDEO - v AVI jich může být více. OF_READ definuje, že nám stačí otevření pouze pro čtení a NULL na konci je ukazatel na třídní identifikátor handleru (Překl.: class identifier of the handler). Abych byl upřímný nemám nejmenší představu, co to znamená, proto pomocí NULL nechávám knihovnu, aby vybrala za mě.

Nastanou-li při otevírání jakékoli problémy, zobrazí se uživateli informační okno, nicméně ukončení programu není implementováno. Přidání nějakého druhu chybových testů by pro vás nemělo být moc těžké, já jsem byl příliš líný.

```

    AVIFileInit();// Připraví knihovnu AVIFile na použití

    if (AVIStreamOpenFromFile(&pavi, szFile, streamtypeVIDEO, 0, OF_READ, NULL) != 0)//
    Otevře AVI proud
    {
        // Chybová zpráva
        MessageBox (HWND_DESKTOP, "Failed To Open The AVI Stream", "Error", MB_OK |
        MB_ICONEXCLAMATION);
    }

```

Pokud jsme se dostali až sem, můžeme předpokládat, že se soubor otevřel v pořádku a video proud byl lokalizován. U deklarace proměnných jsme vytvořili objekt struktury AVIStreamInfo a nazvali ho psi. Voláním funkce AVIStreamInfo () do něj nagrabujeme různé informace o AVI, s jejichž pomocí spočítáme šířku a výšku snímku v pixelech. Potom funkcí AVIStreamLength() získáme číslo posledního snímku videa, které zároveň označuje celkový počet všech snímků.

Výpočet framerate je snadný. Počet snímků za sekundu se rovná psi.dwRate děleno psi.dwScale. Tato hodnota by měla odpovídat číslu, které lze získat kliknutím na AVI soubor a zvolením vlastností. Ptáte se, co to má co společného s mpf (čas zobrazení jednoho snímku)? Když jsem poprvé psal kód pro animaci, zkoušel jsem pro zvolení správného snímku animace použít FPS. Dostal jsem se do problémů... všechna videa se přehrávala příliš rychle. Proto jsem nahlédl do vlastností video souboru face2.avi. Je dlouhé 3,36 sekund, framerate činí 29,974 FPS a má celkem 91 snímků. Pokud vynásobíme 3,36 krát 29,976 dostaneme 100 snímků - velmi nepřesné.

Proto jsem se rozhodl dělat věci trochu jinak. Namísto počtu snímků za sekundu spočítáme, jak dlouho by měl být snímek zobrazen. Funkce AVIStreamSampleToTime() zkonvertuje pozici v animaci na čas v milisekundách, než se video dostane do této pozice. Získáme tedy čas posledního snímku, vydělíme ho jeho pozicí (=počtem všech snímků) a výsledek vložíme do proměnné mpf. Stejně hodnoty byste dosáhli nagrabováním množství času potřebného pro jeden snímek. Příkaz by vypadal takto: AVIStreamSampleToTime(pavi, 1). Oba způsoby jsou možné. Děkuji Albertu Chaulkovi za nápad.

```

    AVIStreamInfo(pavi, &psi, sizeof(psi));// Načte informace o proudu

    width = psi.rcFrame.right - psi.rcFrame.left;// Výpočet šířky
    height = psi.rcFrame.bottom - psi.rcFrame.top;// Výpočet výšky

    lastframe = AVIStreamLength(pavi);// Poslední snímek proudu
    mpf = AVIStreamSampleToTime(pavi, lastframe) / lastframe;// Počet milisekund na
    jeden snímek

```

OpenGL požaduje, aby rozměry textury byly mocninou čísla 2, ale většina videí mívá velikost 160x120, 320x240 nebo jiné nevhodné hodnoty. Pro konverzi na potřebné rozměry použijeme Windows funkce pro práci s DIB obrázky. Jako první věc specifikujeme hlavičku bitmapy a to tak, že vyplníme BITMAPINFOHEADER proměnnou bmih. Nastavíme velikost struktury a biPlanes. Barevnou hloubku určíme na 24 bitů (RGB), obrázek bude mít rozměry 256x256 pixelů a nebude komprimovaný.

```

    bmih.biSize = sizeof(BITMAPINFOHEADER);// Velikost struktury

```

```

bmih.biPlanes = 1; // BiPlanes
bmih.biBitCount = 24; // Počet bitů na pixel
bmih.biWidth = 256; // Šířka bitmapy
bmih.biHeight = 256; // Výška bitmapy
bmih.biCompression = BI_RGB; // RGB mód

```

Funkce `CreateDibSection()` vytvoří obrázek DIB, do kterého budeme moci přímo zapisovat. Pokud vše proběhne v pořádku měl by `hBitmap` obsahovat nově vytvořený obrázek. `Hdc` představuje handle kontextu zařízení, druhý parametr je ukazatel na strukturu, kterou jsme právě inicializovali. Třetí parametr specifikuje RGB typ dat. Do proměnné `data` se uloží ukazatel na data vytvořeného obrázku. Nastavíme-li předposlední parametr na `NULL`, funkce za nás sama alokuje paměť. Poslední parametr budeme jednoduše ignorovat. Příkaz `SelectObject()` zvolí obrázek do kontextu zařízení.

```

hBitmap = CreateDIBSection(hdc, (BITMAPINFO*)&bmih, DIB_RGB_COLORS, (void**)
(&data), NULL, NULL);
SelectObject(hdc, hBitmap); // Zvolí bitmapu do kontextu zařízení

```

Předtím než budeme moci načítat jednotlivé snímky, musíme připravit program na dekomprimaci videa. Zavoláme funkci `AVIStreamGetFrameOpen()` a předáme jí ukazatel na datový proud videa. Za druhý parametr se může předat struktura podobná té výše, pomocí které lze specifikovat vrácený video formát. Bohužel jedinou věcí, kterou lze ovlivnit je šířka a výška obrázku. V MSDN se také uvádí, že se může předat `AVISETFRAMEF_BESTDISPLAYFMT`, který automaticky zvolí nejlepší formát zobrazení. Nicméně můj kompilátor nemá pro tuto symbolickou konstantu žádnou definici. Dopadne-li vše dobře, získáme `GETFRAME` objekt potřebný pro čtení dat jednotlivých snímků. Při problémech se zobrazí chybové okno.

```

pgf = AVIStreamGetFrameOpen(pavi, NULL); // Vytvoří PGETFRAME použitím požadovaného
módu

if (pgf == NULL) // Neúspěch?
{
    MessageBox (HWND_DESKTOP, "Failed To Open The AVI Frame", "Error", MB_OK |
MB_ICONEXCLAMATION);
}

```

Jako třešničku na dortu zobrazíme do titulku okna šířku, výšku a počet snímků videa.

```

// Informace o videu (šířka, výška, počet snímků)
wprintf (title, "NeHe's AVI Player: Width: %d, Height: %d, Frames: %d", width,
height, lastframe);
SetWindowText(g_window->hWnd, title); // Modifikace titulku okna
}

```

Otevírání AVI proběhlo bez problémů, následující funkce nagrahuje jeho jeden snímek, zkonvertuje ho do použitelné formy (velikost, barevná hloubka RGB) a vytvoří z něj texturu. Proměnná `lpbi` bude ukládat informace o hlavičce bitmapy snímku. Příkaz na dalším řádku plní hned několik funkcí. Nagrahuje snímek specifikovaný pomocí `frame` a vyplní `lpbi` informacemi o hlavičce snímku. Přeskočením hlavičky (`lpbi->biSize`) a informací o barvách (`lpbi->biClrUsed * sizeof(RGBQUAD)`) získáme ukazatel na opravdová data obrázku.

```

void GrabAVIFrame(int frame) // Grabuje požadovaný snímek z proudu
{
    LPBITMAPINFOHEADER lpbi; // Hlavička bitmapy

    lpbi = (LPBITMAPINFOHEADER)AVIStreamGetFrame(pgf, frame); // Grabuje data z AVI
proudu
    pdata = (char *)lpbi + lpbi->biSize + lpbi->biClrUsed * sizeof( RGBQUAD); // Ukazatel
na data
}

```

Kvůli textuře musíme zkonvertovat právě získaný obrázek na použitelnou velikost a barevnou hloubku. Pomocí funkce `DrawDibDraw()` můžeme kreslit přímo do našeho DIBu. Její první parametr je `DrawDib DC`, další parametr představuje handle na kontext zařízení. Nuly definují levý horní a 256 pravý dolní roh výsledného obdélníku. `lpbi` je ukazatel na hlavičku snímku, který jsme právě načítali, a `pdata` ukazuje na data obrázku. Následuje levý horní a pravý dolní roh zdrojového obrázku (čili šířka a výška snímku). Poslední parametr necháme na nule. Touto cestou můžeme zkonvertovat obrázek o jakékoli šířce, výšce a barevné hloubce na obrázek 256x256x24.

```

// Konvertování obrázku na požadovaný formát
DrawDibDraw(hdc, hdc, 0, 0, 256, 256, lpbi, pdata, 0, 0, width, height, 0);

```

V současné chvíli už v rukách držíme data, ze kterých lze vygenerovat texturu. Nicméně její R a B složky jsou prohozeny. Proto zavoláme naši ASM funkce, která jednotlivé byty umístí na korektní pozice v obrázku.

```

flipIt(data); // Prohodí R a B složku pixelů

```

Původně jsem texturu aktualizoval jejím smazáním a znovuvytvořením. Několik lidí mi nezávisle na sobě poradilo, abych

zkusil použít `glTexSubImage2D()`. Uvádím citaci z OpenGL Red Book: "Vytvoření textury může být mnohem náročnější než modifikace už existující. V OpenGL Release 1.1 přibýly nové rutiny pro nahrazení všech částí textury za nové informace. Toto může být užitečné pro programy, které např. v real-timu snímají obrázky videa a vytvářejí z nich textury. Aplikace pak za běhu vytvoří pouze jednu texturu a pomocí `glTexSubImage2D()` bude postupně nahrazovat její data za nové snímky videa."

Osobně jsem nezaznamenal větší nárůst rychlosti, ale na pomalejších kartách může být vše jinak. Parametry funkce jsou následující: typ výstupu, úroveň detailů pro mipmapping, x a y offset počátku kopírované oblasti (0, 0 - levý dolní roh), šířka a výška oblasti, RGB formát pixelů, typ dat a ukazatel na data.

Kevin Rogers přidal: Chtěl bych poukázat na další důležitou vlastnost `glTexSubImage2d()`. Nejen, že je rychlejší na mnoha OpenGL implementacích, ale cílová oblast obrázku nemusí být nutně mocninou čísla 2. Toto je především užitečné pro přehrávání videa, jehož rozlišení bývá mocninou dvojky opravdu zřídka (většinou 320x200). Dostáváme tak flexibilní možnost přehrávat video v jeho originální velikosti než jej složitě měnit, někdy i dvakrát (do textury, zpět na obrazovku).

Není možné aktualizovat texturu, pokud jste ji ještě nevytvořili! My ji vytváříme v kódu funkce `Initialize()`. Druhá důležitá věc spočívá v tom, že pokud váš projekt obsahuje více než jednu texturu, musíte před aktualizací zvolit jako aktivní (`glBindTexture()`) tu správnou, protože byste mohli přepsat texturu, kterou nechcete.

```
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 256, 256, GL_RGB, GL_UNSIGNED_BYTE, data); //
    Aktualizace textury
}
```

Následující funkce je volána při ukončování programu. Má za úkol smazat `DrawDib DC` a uvolnit alokované zdroje. Zavírá také `GetFrame` zdroj, odstraňuje souborový proud a ukončuje práci s AVI souborem.

```
void CloseAVI(void) // Zavření AVI souboru
{
    DeleteObject(hBitmap); // Smaže bitmapu
    DrawDibClose(hdd); // Zavře DIB
    AVIStreamGetFrameClose(pgf); // Dealokace GetFrame zdroje
    AVIStreamRelease(pavi); // Uvolnění proudu
    AVIFileExit(); // Uvolnění souboru
}
```

Inicializace je hezky přímočará. nastavíme počáteční úhel na nulu a pomocí knihovny `DrawDib` nagrabujeme `DC`. Pokud se vše zdaří, tak by se mělo `hdd` stát handlem na nově vytvořený kontext zařízení. Dále určíme černé pozadí, zapneme hloubkové testování atd.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Inicializace
{
    g_window = window;
    g_keys = keys;

    angle = 0.0f; // Na počátku nulový úhel
    hdd = DrawDibOpen(); // Kontext zařízení DIBu

    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glDepthFunc(GL_EQUAL); // Typ testů hloubky
    glEnable(GL_DEPTH_TEST); // Zapne testování hloubky
    glShadeModel(GL_SMOOTH); // Jemné stínování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Perspektivní korekce
}
```

V další části kódu zapneme mapování 2D textur, nastavíme filtr `GL_NEAREST` a definujeme kulové mapování, které umožní automatické generování texturových koordinátů. Pokud máte výkonný systém, zkuste použít lineární filtrování, bude vypadat lépe.

```
quadratic = gluNewQuadric(); // Vytvoří objekt quadraticu
gluQuadricNormals(quadratic, GLU_SMOOTH); // Normály
gluQuadricTexture(quadratic, GL_TRUE); // Texturové koordináty

glEnable(GL_TEXTURE_2D); // Zapne texturování
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST); // Filtry textur
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // Automatické generování
koordinátů
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

Po obvyklé inicializaci otevřeme AVI soubor. Jistě jste si všimli, že jsem se snažil udržet rozhraní v co nejjednodušší

formě, takže stačí předat pouze řetězec se jménem souboru. Na konci vytvoříme texturu a ukončíme funkci.

```
OpenAVI("data/face2.avi");// Otevření AVI souboru

// Vytvoření textury
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE, data);

return TRUE;// Vše OK
}
```

Při deinitializaci zavoláme CloseAVI(), čímž kompletně ukončíme práci s videem.

```
void Deinitialize(void)// Deinicializace
{
    CloseAVI();// Zavře AVI
}
```

Ve funkci Update() zjišťujeme případné stisky kláves a v závislosti na uplynulém čase aktualizujeme poměry ve scéně. Jako vždy ESC ukončuje program a F1 přepíná mód fullscreen/okno. Mezerníkem inkrementujeme proměnnou efekt, jejíž hodnota určuje, jestli se ve scéně zobrazuje krychle, koule, válec, popř. nic (pouze pozadí).

```
void Update(DWORD milliseconds)// Aktualizace scény
{
    if (g_keys->keyDown[VK_ESCAPE] == TRUE)// ESC
    {
        TerminateApplication (g_window);// Konec programu
    }

    if (g_keys->keyDown[VK_F1] == TRUE)// F1
    {
        ToggleFullscreen (g_window);// Zamění mód fullscreen/okno
    }

    if ((g_keys->keyDown[' ']) && !sp)// Mezerník
    {
        sp = TRUE;
        effect++;// Následující objekt v řadě

        if (effect > 3)// Přetečení?
        {
            effect = 0;
        }
    }

    if (!g_keys->keyDown[' '])// Uvolnění mezerníku
    {
        sp = FALSE;
    }
}
```

Pomocí klávesy B zapínáme/vypínáme pozadí. Generování texturových koordinátů určuje flag env, který negujeme po stisku klávesy E.

```
if ((g_keys->keyDown['B']) && !bp)// Klávesa B
{
    bp = TRUE;
    bg = !bg;// Nastaví flag pro zobrazování pozadí
}

if (!g_keys->keyDown['B'])// Uvolnění B
{
    bp = FALSE;
}

if ((g_keys->keyDown['E']) && !ep)// Klávesa E
{
    ep = TRUE;
    env = !env;// Nastaví flag pro automatické generování texturových koordinátů
}

if (!g_keys->keyDown['E'])// Uvolnění E
{
    ep = FALSE;
}
```

```
}
```

V závislosti na uplynulém čase zvětšíme úhel natočení objektu.

```
angle += (float)(milliseconds) / 60.0f; // Aktualizace úhlu natočení
```

V originální verzi tutoriálu byla všechna videa přehrávána vždy stejnou rychlostí a to nebylo příliš vhodné. Proto jsem kód přepsal tak, aby jeho rychlost byla vždy korektní. Obsah proměnné `next` zvětšíme o počet uplynulých milisekund od milého volání. Jistě si pamatujete, že `mpf` obsahuje čas, jak dlouho má být každý snímek zobrazen. Vydělíme-li tedy číslo `next` hodnotou `mpf`, získáme správný snímek. Nakonec se ujistíme, že nově vypočtený snímek nepřetekl přes maximální hodnotu. V takovém případě začneme video přehrávat znovu od začátku.

Asi vás nepřekvapí, že pokud je počítač příliš pomalý, některé snímky se automaticky přeskakují. Pokud chcete, aby byl každý snímek zobrazen, přičemž nezávisí na tom, jak pomalu program běží, můžete otestovat, jestli je `next` vyšší než `mpf` a pokud ano, inkrementujte snímek o jedničku a resetujte `next` zpět na nulu. Oba způsoby pracují, ale pro rychlé počítače je vhodnější uvedený kód.

Cítíte-li se plni síly a energie, zkuste implementovat obvyklé funkce video přehrávačů - např. rychlé převíjení, pauzu nebo zpětný chod.

```
next += milliseconds; // Zvětšení next o uplynulý čas
frame = next / mpf; // Výpočet aktuálního snímku

if (frame >= lastframe) // Přetečení snímků?
{
    frame = 0; // Přetočí video na začátek
    next = 0; // Nulování času
}
}
```

Už máme téměř vše, zbývá pouze vykreslování scény. Jako vždy na začátku smažeme obrazovku a hloubkový buffer. Potom nagrabujeme požadovaný snímek animace. Pokud byste chtěli současně používat více videí, museli byste přidat i ID textury - další práce pro vás.

```
void Draw(void) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže buffery

    GrabAVIFrame(frame); // Nagrabuje požadovaný snímek videa
}
```

Chceme-li kreslit pozadí, resetujeme modelview matici a na obyčejný obdélník namapujeme daný snímek videa. Aby se objevil až za všemi objekty, umístíme ho dvacet jednotek do scény a samozřejmě ho roztáhneme na požadovanou velikost.

```
if (bg) // Zobrazuje se pozadí?
{
    glLoadIdentity(); // Reset matice

    glBegin(GL_QUADS); // Vykreslování obdélníků
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 11.0f, 8.3f, -20.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-11.0f, 8.3f, -20.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-11.0f, -8.3f, -20.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 11.0f, -8.3f, -20.0f);
    glEnd();
}
```

Resetujeme matici a přesuneme se deset jednotek do scény. Pokud se `env` rovná `TRUE`, zapneme automatické generování texturových koordinátů.

```
glLoadIdentity(); // Reset matice
glTranslatef(0.0f, 0.0f, -10.0f); // Posun do scény

if (env) // Zapnuto generování souřadnic textur?
{
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);
}
```

Na poslední chvíli jsem přidal i rotaci objektu na osách `x`, `y` a následně přiblížení na ose `z`. Objekt se bude pohybovat po scéně. Bez těchto tří řádků by pouze rotoval na jednom místě uprostřed obrazovky.

```
glRotatef(angle*2.3f, 1.0f, 0.0f, 0.0f); // Rotace
glRotatef(angle*1.8f, 0.0f, 1.0f, 0.0f);
```

```
glTranslatef(0.0f, 0.0f, 2.0f); // Přesun na novou pozici
```

Pomocí větvení do více směrů vykreslíme objekt, který je právě aktivní. Jako první možnost máme krychli.

```
switch (effect) // Větvení podle efektu
{
    case 0: // Krychle
        glRotatef(angle*1.3f, 1.0f, 0.0f, 0.0f); // Rotace
        glRotatef(angle*1.1f, 0.0f, 1.0f, 0.0f);
        glRotatef(angle*1.2f, 0.0f, 0.0f, 1.0f);

        glBegin(GL_QUADS); // Kreslení obdélníků
            // Čelní stěna
            glNormal3f(0.0f, 0.0f, 0.5f);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
            // Zadní stěna
            glNormal3f(0.0f, 0.0f, -0.5f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
            glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
            // Horní stěna
            glNormal3f(0.0f, 0.5f, 0.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
            // Spodní stěna
            glNormal3f(0.0f, -0.5f, 0.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
            glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
            // Pravá stěna
            glNormal3f(0.5f, 0.0f, 0.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
            glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
            // Levá stěna
            glNormal3f(-0.5f, 0.0f, 0.0f);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
            glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
            glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
        glEnd();
        break;
```

Jak vykreslit kouli, už jistě dávno víte, nicméně pro jistotu přidávám krátký komentář. Její poloměr činí 1.3f jednotek, skládá se z dvaceti poledníků a dvaceti rovnoběžek. Používám číslo 20, protože chci, aby nebyla perfektně hladká, ale trochu segmentovaná - bude vidět náznak její rotace.

```
case 1: // Koule
    glRotatef(angle*1.3f, 1.0f, 0.0f, 0.0f); // Rotace
    glRotatef(angle*1.1f, 0.0f, 1.0f, 0.0f);
    glRotatef(angle*1.2f, 0.0f, 0.0f, 1.0f);

    gluSphere(quadratic, 1.3f, 20, 20); // Vykreslení koule
    break;
```

Válec vykreslíme pomocí funkce gluCylinder(). Bude mít průměr 1.0f a jeho výška bude činit tři jednotky.

```
case 2: // Válec
    glRotatef(angle*1.3f, 1.0f, 0.0f, 0.0f); // Rotace
    glRotatef(angle*1.1f, 0.0f, 1.0f, 0.0f);
    glRotatef(angle*1.2f, 0.0f, 0.0f, 1.0f);
    glTranslatef(0.0f, 0.0f, -1.5f); // Vycentrování
```

```
        gluCylinder(quadratic, 1.0f, 1.0f, 3.0f, 32, 32); // Vykreslení válce
        break;
    }
```

Pokud je env v jedničce, vypneme generování texturových koordinátů.

```
    if (env) // Zapnuto generování souřadnic textur?
    {
        glDisable(GL_TEXTURE_GEN_S);
        glDisable(GL_TEXTURE_GEN_T);
    }

    glFlush(); // Vyprázdní OpenGL pipeline
}
```

Doufám, že jste si, stejně jako já, užili tento tutoriál. Za chvíli budou 2 hodiny ráno... už na něm pracuji přes šest hodin. Zní to šíleně, ale psaní textu, aby dával smysl, není lehký úkol. Vše jsem třikrát přečetl a snažil se objasnit věci co nejlépe. Věřte nebo ne, pro mě je důležité, abyste pochopili, jak věci pracují a proč vůbec pracují. Bez čtenářů bych brzy skončil.

Jak už jsem napsal, toto je můj první pokus o přehrávání videa. Normálně nepíši o předmětu, který jsem se právě naučil, ale myslím, že mi to pro jednu odpuštěte. Faktem je, že jsem si od cizích lidí půjčil opravdu absolutní minimum kódu, vše je původní. Doufám, že se mi podařilo otevřít dveře povodni přehrávání AVI ve vašich kvalitních demech. Možná se tak stane, možná ne. Každopádně ukázkový tutoriál už máte.

Obrovské díky patří Fredsterovi, který vytvořil ukázkové video tváře. Byla to jedna z celkem šesti animací, které mi poslal. Žádné dotazy, žádné požadavky. Poslal jsem mu email s prosbou a on mi pomohl. Obrovský respekt.

Největší dík však patří Jonathanu de Blok. Nebýt jeho, tento tutoriál by nevznikl. Právě on ve mně vzbudil zájem o AVI formát. Poslal mi totiž část kódu z jeho přehrávače. Trpělivě odpovídal na všechny otázky ohledně jeho kódu. Nic jsem si však nepůjčil, můj kód pracuje na úplně jiném základu.

**napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>**

Lekce 36 - Radial Blur, renderování do textury

Společnými silami vytvoříme extrémně působivý efekt radial blur, který nevyžaduje žádná OpenGL rozšíření a funguje na jakémkoli hardwaru. Naučíte se také, jak lze na pozadí aplikace vyrenderovat scénu do textury, aby pozorovatel nic neviděl.

Ahoj, jmenuji se Dario Corno, ale jsem také z nám jako rlo ze Spinning Kids. První ze všeho vysvětlím, proč jsem se rozhodl napsat tento tutoriál. Roku 1989 jsem se stal "scénařem". Chtěl bych po vás, abyste si stáhli nějaká dema. Pochopíte, co to demo je a v čem spočívají demo efekty.

Dema vytvářejí opravdoví koderři na ukázkou hardcore a často i brutálních kódovacích technik. Svým způsobem jsou druhem umění, ve kterém se spojuje vše od hudby (hudba na pozadí, zvuky) a malířství (grafika, design, modely) přes matematiku a fyziku (vše funguje na nějakých principech) až po programování a detailní znalost počítače na úrovni hardwaru. Obrovské kolekce dem můžete najít na <http://www.pouet.net/> a <http://ftp.scene.org/>, v Čechách pak <http://www.scene.cz/>. Ale abyste se hned na začátku nevylekali... toto není pravý smrtící tutoriál, i když musím uznat, že výsledek stojí za to.

Překl.: Se svým prvním demem jsem se setkal ve druháku na střední, kdy nám spolubydílci na intru Lukáš Duzsty Hoger ukazoval na 486 notebooku jeden prográmeček, který zabíral kolem 2 kB. Na začátku byla vidět ruka, jak kreslí na plátno dům, strom a postavy, scéna se vybouřila do 3D a musím říct, že na 256 barev a DOSovou grafiku vše vypadalo úchvatně - kam se programátoři využívající pohodlných služeb OpenGL vůbec hrabou :-). Proti tomu koderovi fakt batolata. Asi nejlepší demo, které jsem kdy viděl byla 64 kB animace "reálného" 3D prostředí ve video kvalitě, která trvala něco přes čtvrt hodiny. Jenom texty v kreditu na konci musely zabírat polovinu místa. Zkuste si pro zajímavost zkompilovat prázdnou MFC aplikaci vygenerovanou APP Wizzardem, která navíc tahá většinu potřebných funkcí z DLL knihoven - nedostanete se pod 30 kB.

Tolik tedy k úvodu... Co se ale dozvíte v tomto tutoriálu? Vysvětlím vám, jak vytvořit perfektní efekt (používaný v demech), který vypadá jako radial blur (radiální rozmazání). Někdy je také označován jako volumetrická světla, ale nevěřte, je to pouze obyčejný radial blur.

Radial blur bývá obvykle vytvářen (pouze při softwarovém renderingu) rozmazáváním pixelů originálního obrázku v opačném směru než se nachází střed rozmazávání. S dnešním hardwarem je docela obtížné provádět ruční blurring (rozmazávání) za použití color bufferu (alespoň v případě, že je podporován všemi grafickými kartami), takže potřebujeme využít malého triku, abychom dosáhli alespoň podobného efektu. Jako bonus se také dozvíte, jak je snadné renderovat do textury.

Objekt, který jsem se pro tento tutoriál rozhodl použít, je spirála, protože vypadá hodně dobře. Navíc jsem už celkem unavený z krychliček :-] Musím ještě poznamenat, že vysvětluji hlavně vytváření výsledného efektu, naopak pomocný kód už méně detailněji. Měli byste ho mít už dávno zažitý.

```
// Uživatelské proměnné
float angle;// Úhel rotace spirály
float vertexes[4][3];// Čtyři body o třech souřadnicích
float normal[3];// Data normálového vektoru
GLuint BlurTexture;// Textura
```

Tak tedy začneme... Funkce EmptyTexture() generuje prázdnou texturu a vrací číslo jejího identifikátoru. Na začátku alokujeme paměť obrázku o velikosti 128*128*4. Tato čísla označují šířku, výšku a barevnou hloubku (RGBA) obrázku. Po alokaci paměť vynulujeme. Protože budeme texturu roztahovat, použijeme pro ni lineární filtrování, GL_NEAREST v našem případě nevypadá zrovna nejlépe.

```
GLuint EmptyTexture()// Vytvoří prázdnou texturu
{
    GLuint txtnumber;// ID textury
    unsigned int* data;// Ukazatel na data obrázku

    data = (unsigned int*) new GLuint[((128 * 128) * 4 * sizeof(unsigned int))];//
    Alokace paměti
    ZeroMemory(data, ((128 * 128) * 4 * sizeof(unsigned int)));// Nulování paměti

    glGenTextures(1, &txtnumber);// Jedna textura
    glBindTexture(GL_TEXTURE_2D, txtnumber);// Zvolí texturu
    glTexImage2D(GL_TEXTURE_2D, 0, 4, 128, 128, 0, GL_RGBA, GL_UNSIGNED_BYTE, data);//
    Vytvoření textury

    // Lineární filtrování pro zmenšení i zvětšení
```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

delete [] data;// Uvolnění paměti

return txtnumber;// Vrátí ID textury
}

```

Následující funkce normalizuje vektor, který je předán v parametru jako pole tří floatů. Spočítáme jeho délku a s její pomocí vydělíme všechny tři složky.

```

void ReduceToUnit(float vector[3])// Výpočet normalizovaného vektoru (jednotková délka)
{
    float length;// Délka vektoru

    // Výpočet současné délky vektoru
    length = (float)sqrt((vector[0]*vector[0]) + (vector[1]*vector[1]) + (vector[2]
    *vector[2]));

    if(length == 0.0f)// Prevence dělení nulou
    {
        length = 1.0f;
    }

    vector[0] /= length;// Vydělení jednotlivých složek délkou
    vector[1] /= length;
    vector[2] /= length;

    // Výsledný vektor je předán zpět v parametru funkce
}

```

Pomocí funkce calcNormal() lze vypočítat vektor, který je kolmý ke třem bodům tvořícím rovinu. Dostali jsme dva parametry: v[3][3] představuje tři body (o třech složkách x,y,z) a do out[3] uložíme výsledek. Na začátku deklarujeme dva pomocné vektory a tři konstanty, které vystupují jako indexy do pole.

```

void calcNormal(float v[3][3], float out[3])// Výpočet normálového vektoru polygonu
{
    float v1[3], v2[3];// Vektor 1 a vektor 2 (x,y,z)

    static const int x = 0;// Pomocné indexy do pole
    static const int y = 1;
    static const int z = 2;

```

Ze třech bodů předaných funkci vytvoříme dva vektory a spočítáme třetí vektor, který je k nim kolmý.

```

v1[x] = v[0][x] - v[1][x];// Výpočet vektoru z 1. bodu do 0. bodu
v1[y] = v[0][y] - v[1][y];
v1[z] = v[0][z] - v[1][z];

v2[x] = v[1][x] - v[2][x];// Výpočet vektoru z 2. bodu do 1. bodu
v2[y] = v[1][y] - v[2][y];
v2[z] = v[1][z] - v[2][z];

// Výsledkem vektorového součinu dvou vektorů je třetí vektor, který je k nim kolmý
out[x] = v1[y]*v2[z] - v1[z]*v2[y];
out[y] = v1[z]*v2[x] - v1[x]*v2[z];
out[z] = v1[x]*v2[y] - v1[y]*v2[x];

```

Aby vše bylo dokonalé, tak výsledný vektor normalizujeme na jednotkovou délku.

```

ReduceToUnit(out);// Normalizace výsledného vektoru

// Výsledný vektor je předán zpět v parametru funkce
}

```

Následující rutina vykresluje spirálu. Po deklaraci proměnných nastavíme pomocí gluLookAt() výhled do scény. Díváme se z bodu 0, 5, 50 do bodu 0, 0, 0. UP vektor míří vzhůru ve směru osy y.

```

void ProcessHelix()// Vykreslí spirálu
{
    GLfloat x;// Souřadnice x, y, z
    GLfloat y;
    GLfloat z;

```

```

GLfloat phi;// Úhly
GLfloat theta;
GLfloat u;
GLfloat v;

GLfloat r;// Poloměr závitů
int twists = 5;// Pět závitů

GLfloat glfMaterialColor[] = { 0.4f, 0.2f, 0.8f, 1.0f};// Barva materiálu
GLfloat specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};// Specular světlo

glLoadIdentity();// Reset matice
gluLookAt(0,5,50, 0,0,0, 0,1,0);// Pozice očí (0,5,50), střed scény (0,0,0), UP
vektor na ose y

```

Uložíme matici a přesuneme se o padesát jednotek do scény. V závislosti na úhlu angle (globální proměnná) se spirálou rotujeme. Také nastavíme materiály.

```

glPushMatrix();// Uložení matice

glTranslatef(0, 0, -50);// Padesát jednotek do scény
glRotatef(angle/2.0f, 1, 0, 0);// Rotace na ose x
glRotatef(angle/3.0f, 0, 1, 0);// Rotace na ose y

// Nastavení materiálů
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, glfMaterialColor);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);

```

Pokud ovládáte goniometrické funkce, je výpočet jednotlivých bodů spirály relativně jednoduchý, ale nebudu to zde vysvětlovat (Překl.: díky bohu... :-), protože spirála není hlavní náplní tohoto tutoriálu. Navíc jsem si kód půjčil od kamarádů z Listen Software. Půjdeme jednodušší, ale ne nejrychlejší cestou. S vertex arrays by bylo vše mnohem rychlejší.

```

r = 1.5f;// Poloměr

glBegin(GL_QUADS);// Kreslení obdélníků
for(phi = 0; phi <= 360; phi += 20.0)// 360 stupňů v kroku po 20 stupních
{
    for(theta = 0; theta <= 360*twists; theta += 20.0)// 360 stupňů* počet
závitů po 20 stupních
    {
        v = (phi / 180.0f * 3.142f);// Úhel prvního bodu (0)
        u = (theta / 180.0f * 3.142f);// Úhel prvního bodu (0)

        x = float(cos(u) * (2.0f + cos(v))) * r;// Pozice x, y, z prvního bodu
        y = float(sin(u) * (2.0f + cos(v))) * r;
        z = float((u - (2.0f * 3.142f)) + sin(v)) * r);

        vertexes[0][0] = x;// Kopírování prvního bodu do pole
        vertexes[0][1] = y;
        vertexes[0][2] = z;

        v = (phi / 180.0f * 3.142f);// Úhel druhého bodu (0)
        u = ((theta + 20) / 180.0f * 3.142f);// Úhel druhého bodu (20)

        x = float(cos(u) * (2.0f + cos(v))) * r;// Pozice x, y, z druhého bodu
        y = float(sin(u) * (2.0f + cos(v))) * r;
        z = float((u - (2.0f * 3.142f)) + sin(v)) * r);

        vertexes[1][0] = x;// Kopírování druhého bodu do pole
        vertexes[1][1] = y;
        vertexes[1][2] = z;

        v=((phi + 20) / 180.0f * 3.142f);// Úhel třetího bodu (20)
        u=((theta + 20) / 180.0f * 3.142f);// Úhel třetího bodu (20)

        x = float(cos(u) * (2.0f + cos(v))) * r;// Pozice x, y, z třetího bodu
        y = float(sin(u) * (2.0f + cos(v))) * r;
        z = float((u - (2.0f * 3.142f)) + sin(v)) * r);

        vertexes[2][0] = x;// Kopírování třetího bodu do pole
        vertexes[2][1] = y;
        vertexes[2][2] = z;
    }
}

```

```

v = ((phi + 20) / 180.0f * 3.142f); // Úhel čtvrtého bodu (20)
u = ((theta) / 180.0f * 3.142f); // Úhel čtvrtého bodu (0)

x = float(cos(u) * (2.0f + cos(v))) * r; // Pozice x, y, z čtvrtého bodu
y = float(sin(u) * (2.0f + cos(v))) * r;
z = float(((u - (2.0f * 3.142f)) + sin(v)) * r);

vertexes[3][0] = x; // Kopírování čtvrtého bodu do pole
vertexes[3][1] = y;
vertexes[3][2] = z;

calcNormal(vertexes, normal); // Výpočet normály obdélníku

glNormal3f(normal[0], normal[1], normal[2]); // Poslání normály OpenGL

// Rendering obdélníku
glVertex3f(vertexes[0][0], vertexes[0][1], vertexes[0][2]);
glVertex3f(vertexes[1][0], vertexes[1][1], vertexes[1][2]);
glVertex3f(vertexes[2][0], vertexes[2][1], vertexes[2][2]);
glVertex3f(vertexes[3][0], vertexes[3][1], vertexes[3][2]);
}
}
glEnd(); // Konec kreslení
glPopMatrix(); // Obnovení matice
}

```

Funkce ViewOrtho() slouží k přepnutí z perspektivní projekce do pravoúhlé a ViewPerspective() k návratu zpět. Vše už bylo popsáno například v tutoriálech o fontech, ale i jinde, takže to zde nebudu znovu probírat.

```

void ViewOrtho() // Nastavuje pravoúhrou projekci
{
    glMatrixMode(GL_PROJECTION); // Projekční matice
    glPushMatrix(); // Uložení matice
    glLoadIdentity(); // Reset matice
    glOrtho(0, 640, 480, 0, -1, 1); // Nastavení pravoúhlé projekce
    glMatrixMode(GL_MODELVIEW); // Modelview matice
    glPushMatrix(); // Uložení matice
    glLoadIdentity(); // Reset matice
}

void ViewPerspective() // Obnovení perspektivního módu
{
    glMatrixMode(GL_PROJECTION); // Projekční matice
    glPopMatrix(); // Obnovení matice
    glMatrixMode(GL_MODELVIEW); // Modelview matice
    glPopMatrix(); // Obnovení matice
}

```

Pojďme si vysvětlit, jak pracuje naše imitace efektu radial blur. Potřebujeme vykreslit scénu tak, aby se jevila jakoby rozmazaná od středu do všech směrů. Nemůžeme číst ani zapisovat pixely a pokud chceme zachovat kompatibilitu s různými grafickými kartami, neměli bychom používat ani OpenGL rozšíření ani jiné příkazy specifické pro určitý hardware. Řešení je docela snadné, OpenGL nám dává možnost blurnout (rozmazat) textury. OK... ne opravdový blurring. Pokud za použití lineárního filtrování roztáhneme textury, výsledek bude, s trochou představivosti, vypadat podobně jako gausovo rozmazávání (gaussian blur). Takže, co se stane, pokud přilepíme spoustu roztáhnutých textur vyobrazujících 3D objekt na scénu přesně před něj? Odpověď je celkem snadná - radial blur!

Potřebujeme však vyřešit dva související problémy: jak v reálném čase vytvářet tuto texturu a jak ji zobrazit přesně před objekt. Řešení prvního je mnohem snazší než si asi myslíte. Co takhle renderovat přímo do textury? Pokud aplikace používá double buffering, je přední buffer zobrazen na obrazovce a do zadního se kreslí. Dokud nezavoláme příkaz SwapBuffers(), změny se navenek neprojeví. Renderování do textury spočívá v renderingu do zadního bufferu (tedy klasicky, jak jsme zvyklí) a v zkopírování jeho obsahu do textury pomocí funkce glCopyTexImage2D().

Problém dva: vycentrování textury přesně před 3D objekt. Víme, že pokud změníme viewport bez nastavení správné perspektivy, získáme deformovanou scénu. Například, nastavíme-li ho opravdu široký bude scéna roztáhnuta vertikálně.

Nejdříve nastavíme viewport tak, aby byl čtvercový a měl stejné rozměry jako textura (128x128). Po renderování objektu, nakopírujeme color buffer do textury a smažeme ho. Obnovíme původní rozměry a vykreslíme objekt podruhé, tentokrát při správném rozlišení. Poté, co texturu namapujeme na obdélník o velikosti scény, roztáhne se zpět na původní velikost a bude umístěna přesně před 3D objekt. Doufám, že to dává smysl. Představte si 640x480 screenshot zmenšený na bitmapu o velikosti 128x128 pixelů. Tuto bitmapu můžeme v grafickém editoru roztáhnout na původní rozměry 640x480 pixelů. Kvalita bude o mnoho horší, ale obrázku si budou odpovídat.

Pojďme se podívat na kód. Funkce `RenderToTexture()` je opravdu jednoduchá, ale představuje kvalitní "designový trik". Nastavíme viewport na rozměry textury a zavoláme rutinu pro vykreslení spirály. Potom zvolíme blur texturu jako aktivní a z viewportu do ní nakopírujeme color buffer. První parametr funkce `glCopyTexImage2D()` indikuje, že používáme 2D texturu, nula označuje úroveň mip mapy (mip map level), defaultně se zadává nula. `GL_LUMINANCE` představuje formát dat. Používáme právě tuto část bufferu, protože výsledek vypadá přesvědčivěji, než kdybychom zadali např. `GL_ALPHA`, `GL_RGB`, `GL_INTENSITY` nebo jiné. Další dva parametry říkají, kde začít (0, 0), dvakrát 128 představuje výšku a šířku. Poslední parametr bychom změnili, kdybychom požadovali okraj (rámeček), ale teď ho nechceme. V tuto chvíli máme v textuře uloženu kopii color bufferu. Smažeme ho a nastavíme viewport zpět na správné rozměry.

DŮLEŽITÉ: Tento postup může být použit pouze s double bufferingem. Důvodem je, že všechny potřebné operace se musí provádět na pozadí (v zadním bufferu), aby je uživatel neviděl.

```
void RenderToTexture()// Rendering do textury
{
    glViewport(0, 0, 128, 128);// Nastavení viewportu (odpovídá velikosti textury)

    ProcessHelix();// Rendering spirály

    glBindTexture(GL_TEXTURE_2D, BlurTexture);// Zvolí texturu

    // Zkopíruje viewport do textury (od 0, 0 do 128, 128, bez okraje)
    glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, 0, 0, 128, 128, 0);

    glClearColor(0.0f, 0.0f, 0.5f, 0.5);// Středně modrá barva pozadí
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smaže obrazovku a hloubkový
    buffer

    glViewport(0, 0, 640, 480);// Obnovení viewportu
}
```

Funkce `DrawBlur()` vykresluje před scénou několik průhledných otexturovaných obdélníků. Pohrajeme-li si trochu s alfou dostaneme imitaci efektu radial blur. Nejprve vypneme automatické generování texturových koordinát a potom zapneme 2D texturu. Vypneme depth testy, nastavíme blending, zapneme ho a zvolíme texturu. Abychom mohli snadno kreslit obdélníky přesně přes celou scénu, přepneme do pravouhlé projekce.

```
void DrawBlur(int times, float inc)// Vykreslí rozmazaný obrázek
{
    float spost = 0.0f;// Počáteční offset souřadnic na textuře
    float alphainc = 0.9f / times;// Rychlost blednutí pro alfa blending
    float alpha = 0.2f;// Počáteční hodnota alfy

    glDisable(GL_TEXTURE_GEN_S);// Vypne automatické generování texturových koordinátů
    glDisable(GL_TEXTURE_GEN_T);

    glEnable(GL_TEXTURE_2D);// Zapne mapování textur
    glDisable(GL_DEPTH_TEST);// Vypne testování hloubky
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);// Mód blendingu
    glEnable(GL_BLEND);// Zapne blending
    glBindTexture(GL_TEXTURE_2D, BlurTexture);// Zvolí texturu

    ViewOrtho();// Přepne do pravouhlé projekce
```

V cyklu vykreslíme texturu tolikrát, abychom vytvořili radial blur. Souřadnice vertexů zůstávají pořád stejné, ale zvětšujeme koordináty u textur a také snižujeme alfu. Takto vykreslíme celkem 25 quadů, jejichž textura se roztahuje pokaždé o 0.015f.

```
    alphainc = alpha / times;// Hodnota změny alfy při jednom kroku

    glBegin(GL_QUADS);// Kreslení obdélníků
    for (int num = 0; num < times; num++)// Počet kroků renderování skvrn
    {
        glColor4f(1.0f, 1.0f, 1.0f, alpha);// Nastavení hodnoty alfy

        glTexCoord2f(0 + spost, 1 - spost);// Texturové koordináty (0, 1)
        glVertex2f(0, 0);// První vertex (0, 0)

        glTexCoord2f(0 + spost, 0 + spost);// Texturové koordináty (0, 0)
        glVertex2f(0, 480);// Druhý vertex (0, 480)

        glTexCoord2f(1 - spost, 0 + spost);// Texturové koordináty (1, 0)
        glVertex2f(640, 480);// Třetí vertex (640, 480)

        glTexCoord2f(1 - spost, 1 - spost);// Texturové koordináty (1, 1)
```

```

        glVertex2f(640, 0); // Čtvrtý vertex (640, 0)

        spost += inc; // Postupné zvyšování skvrn (zoomování do středu textury)
        alpha = alpha - alphainc; // Postupné snižování alfy (blednutí obrázku)
    }
    glEnd(); // Konec kreslení

```

Zbývá obnovit původní parametry.

```

ViewPerspective(); // Obnovení perspektivy

glEnable(GL_DEPTH_TEST); // Zapne testování hloubky
glDisable(GL_TEXTURE_2D); // Vypne mapování textur
glDisable(GL_BLEND); // Vypne blending
glBindTexture(GL_TEXTURE_2D, 0); // Zrušení vybrané textury
}

```

Draw() je tentokrát opravdu krátká. Nastavíme černé pozadí, smažeme obrazovku i hloubku a resetujeme matici. Vyrenderujeme spirálu do textury, potom i na obrazovku a nakonec vykreslíme blur efekt.

```

void Draw(void) // Vykreslení scény
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.5); // Černé pozadí
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubku
    glLoadIdentity(); // Reset matice

    RenderToTexture(); // Rendering do textury
    ProcessHelix(); // Rendering spirály
    DrawBlur(25, 0.02f); // Rendering blur efektu

    glFlush(); // Vyprázdnění OpenGL pipeline
}

```

Doufám, že se vám tento tutoriál líbil. Nenačili jste se sice nic víc než rendering do textury, ale výsledný efekt vypadá opravdu skvěle.

Máte svobodu v používání tohoto kódu ve svých programech jakkoli chcete, ale před tím, než tak učiníte, podívejte se na něj a pochopte ho - jediná podmínka! Abych nezapomněl, uveďte mě prosím do kreditů.

Tady vám nechávám seznam úloh, které si můžete zkusit vyřešit:

- Modifikujte funkci DrawBlur() tak, abyste získali horizontální rozmazání, vertikální rozmazání nebo další efekty (Twirl blur)
- Pohrajte si s parametry DrawBlur() (přidat, odstranit), abyste grafiku synchronizovali s hudbou
- Modifikujte parametry textury - např. GL_LUMINANCE (hezké stínování)
- Zkuste super falešné volumetrické stínování použitím tmavých textur namísto luminance textury

Tak to už bylo opravdu všechno. Zkuste navštívit mé webové stránky <http://www.spinningkids.org/rio> , naleznete tam několik dalších tutoriálů...

napsal: Dario Corno - rlo <rio (zavináč) spinningkids.org>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 37 - Cel-Shading

Cel-Shading je druh vykreslování, při kterém výsledné modely vypadají jako ručně kreslené karikatury z komiksů (cartoons). Rozličné efekty mohou být dosaženy miniaturní modifikací zdrojového kódu. Cel-Shading je velmi úspěšným druhem renderingu, který dokáže kompletně změnit duch hry. Ne ale vždy... musí se umět a použít s rozmyslem.

Článek o teorii Cel-Shadingu napsal Sami "MENTAL" Hamlaoui a umístil ho na [GameDev.net](#). Poté, co byl jeho článek publikován, zavalili čtenáři Samiho emaily, ve kterých se dotazovali po zdrojovém kódu. Napsal tedy další článek, tentokrát pro NeHe, který už ale popisuje pouze zdrojový kód. Tato česká verze je složena z obou článků - teoretického i praktického.

Teoretická část

Předtím než půjdete dál, měli byste mít dostatečné znalosti z následujících oblastí:

- Mapování 1D textur
- Texturovací koordináty
- Softwarové osvětlení
- Vektorová matematika

Pokud něčemu z těchto čtyř položek nerozumíte, neznamená to, že byste nutně neporozuměli Cel-Shadingu, ale určitě budete mít obrovské potíže s psaním vlastních programů.

Základní rendering

Začneme opravdu jednoduchými věcmi. Žádná světla, žádné obrysy, pouze ploché cartoon modely. Budeme potřebovat jenom dva druhy dat - pozici a barvu každého vertexu. Před kreslením vždy vypneme osvětlení a blending. Co by se stalo? Při zapnutých světlech by objekty vypadaly normálně. Nedosáhli bychom plochého cartoon efektu. Blending vypínáme, aby se jednotlivé vertexy nesmíchaly s ostatními.

Shrnuto

- Vypnout světla
- Vypnout blending
- Vykreslit obarvené body

Základní osvětlení (směrové)

Každý vertex bude potřebovat i další data. Kromě původní pozice a barvy budeme používat i normálový vektor a intenzitu osvětlení (jedna float hodnota). Tyto nové proměnné použijeme pro renderování se základním osvětlením.

Světelné mapy (lighting maps)

Nechci vás poplést, pod lightmapami si nepředstavujte simulaci světla na objektech typu Quake 1 a Quake 2. Podívejte se na stěny, abyste pochopili, co mám na mysli. Nepředstavujte si oblasti, které jsou osvětleny/ztmaveny specifickými místy map. To, co budeme používat zde, je kompletně novou formou lightmap - 1D textury.

Zkuste si najít nějakou animaci (Cartoon Network je vždy dobrým zdrojem) a podívejte se na osvětlení postav. Všimli jste si, že nejsou hladké jako v reálném životě? Světlo se rozděluje do jednotlivých plošek. Nikdy jsem neslyšel žádný termín nebo pojmenování pro tento efekt, takže mu budeme říkat Sharp lighting. Abychom ho vytvořili potřebujeme definovat 1D texturu, která bude ukládat požadované hodnoty.



Toto je textura 1x16 pixelů (velmi zvětšená). Používáme hodnoty stupňů šedi, protože budou zkombinovány s barvou vertexu. Můžete si všimnout, že v lightmapě jsou pouze 3 barvy, které mají podobnou intenzitu, jaká se používá v animovaných filmech. Díky tomu, že používáme velikost právě 16 pixelů, můžeme snadno modifikovat hodnoty, abychom vytvořili rozličné efekty. Pokud chcete, můžete také použít obyčejnou černobílou texturu, ale nedoporučuje se

to. Nikdy byste neměli použít 100% černou, protože tato barva vytváří vyzdvižení a okraje, které vypadají dost špatně.

Jakmile máte vytvořenu svou texturu, nahrajte ji do API, které používáte (OpenGL, DX, software). Vrátime se k ní za chvíli.

Počítání osvětlení

Teď přijdou vhod znalosti ohledně softwarového osvětlení. Pokusím se vše vysvětlit jednoduchým jazykem. Vždy se ujistěte, že máte normalizovaný směrový vektor světla! Vše, co potřebujeme udělat, je spočítání skalárního součinu vektoru světla s normálou vertexu.

Skalární součin je matematická funkce, která spočítá úhel mezi dvěma vektory a vrátí ho jako kosinus úhlu. Invertujeme-li kosinus získáme úhel. Namísto kosinu však považujte číslo za texturovací koordinátu. Texturovací koordináty jsou čísla od nuly do jedné. Kosinus je sice v rozmezí -1 až 1, ale pokud bude číslo záporné můžeme mu přiřadit nulu. Skalární součin vektoru světla a normály vertexu můžeme tedy považovat za texturovací koordináty!

Rendering objektu

Nyní máme texturovací koordináty každého vertexu, je čas vykreslit objekt. Stejně jako minule vypneme světla i blending, ale zapneme 1D texturování. Vykreslíme objekt stejně jako minule, ale před tím, než umístíme vertex, specifikujeme texturové koordináty (simulace světla).

Shrnutí

- Vytvořit Sharp lighting mapu
- Spočítat a uložit skalární součin mezi normálou vertexu a směrovým vektorem světla
- Vypnout světla a blending
- Zapnout texturování
- Zvolit texturu lightmapy
- Vykreslit polygony určené texturovacími koordinátami, barvou a pozicí vertexů

Umístitelná světla

Tato metoda je pouhou modifikací minulého postupu. Umístitelné světlo nabízí mnohem více flexibility než směrové osvětlení, protože může být libovolně posouváno po scéně. Dynamicky osvětlované polygony jsou více realistické, ale použitá matematika je delší. Ne komplikovanější, pouze delší.

Spočítání Sharp koordinátů světla

U směrového osvětlení jsme potřebovali získat skalární součin směrového vektoru světla s normálou vertexu. Nyní, protože umístitelná světla nemají směrový vektor (emitují světlo do všech směrů), bude mít každý vertex svůj paprsek, který září skrz něj. Nejdříve potřebujeme určit vektor směřující z pozice světelného zdroje k pozici vertexu. Normalizujeme ho, takže bude mít jednotkovou délku. tím jsme získali směr světla k vertexu. Vypočítáme skalární součin mezi vektorem světla a normálou vertexu. Vše opakujeme pro každý vertex ve scéně. Těmito nadbytečnými výpočty se však sníží FPS. Pojďme se podívat na rychlejší metodu, která redukuje celkový počet osvětlených vertexů.

Testování vzdálenosti od světla

Ke snížení počtu osvětlených vertexů přiřadíme každému světlu poloměr kam paprsky dosahují. Před počítáním hodnot osvětlení (viz. výše) zkontrolujeme, jestli je vertex v kouli, určené poloměrem světla. Pokud ano, aplikujeme na něj světla. Je to základní detekce kolizí s koulí, na kterou existují spousty článků a tutoriálů.

Rendering

Objekt vykreslíme stejně jako u směrového osvětlení. Specifikujeme barvu, texturovací koordináty a pozici.

Shrnutí

- Vytvořit Sharp lighting mapu
- Při použití poloměru světla zjistit, jestli je bod uvnitř
- Získat a normalizovat vektor od světla k vertexu
- Vypočítat skalární součin vektoru s normálou vertexu
- Zopakovat 2-4x pro každý vertex (Překl.: ???)
- Renderovat jako minule

Obrysy a zvýraznění

Obrysy a zvýraznění jsou tenké černé linky reprezentující tahy tužkou, které zdůrazňují okraje. Můžou znepokojit, ale

jejich vytvoření je mnohem jednodušší než si myslíte.

Výpočet kde zvýrazňovat

Pravidlo je jednoduché: vykreslit linku na okraji, který má jeden přivrácený a jeden odvrácený polygon. Zní to hloupě, ale zkuste se podívat například na klávesnici. Všimli jste si, že nemůžete vidět zadní části kláves? To proto, že jsou odvrácené. Na rozhraní vykreslíme čáru, abychom zvýraznili, že tam je okraj.

Možná, že si to ani neuvědomujete, ale nikde jsem se nezmínil o našem vlastním cullingu polygonů. To proto, že vše za nás udělá API, ve kterém programujeme.

Rendering zvýraznění

Klasicky vykreslíme objekt a pak nastavíme šířku čáry na dva až tři pixely. Můžeme také zapnout antialiasing. Změníme mód cullingu, aby odstraňoval přivrácené polygony. Přepneme do drátěného modelu, takže se budou vykreslovat pouze okrajové hrany polygonů. Vykreslíme je, ale nepotřebujeme specifikovat barvu a texturovací koordináty. Tím vykreslíme drátěný model objektu z relativně širokých linek. Nicméně... cullingem jsou linky přivrácených polygonů odstraněny a depth bufferem se vyřadí všechny linky, které jsou hlouběji než přivrácené (tedy ty zadní). Zdálo by se, že tedy nevykreslíme nic. Ale díky šířce čáry zasahují linky okrajových polygonů až za okraje objektu. Právě ty se vykreslí. Z toho plyne, že tato metoda nebude pracovat při tloušťce čáry nastavené na jeden pixel.

Shrnuto

- Vykreslit objekt jako normálně
- Přepnout orientaci faců
- Nastavit 100% černou barvu
- Změnit mód polygonů na drátěný model
- Vykreslit objekt znovu, ale specifikovat pouze pozice vertexů
- Obnovit originální nastavení

To je z teorie asi všechno. Nyní se ji pokusíme převést do praxe.

Praktická část

Na začátku bych se chtěl omluvit za volbu použitého modelu, ale v poslední době si hodně hraji s Quake 2...

```
#include <windows.h>// Hlavičkový soubor pro Windows

#include <gl\gl.h>// Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h>// Hlavičkový soubor pro Glu32 knihovnu
#include <gl\glaux.h>// Hlavičkový soubor pro Glaux knihovnu

#include <math.h>// Hlavičkový soubor pro matematickou knihovnu
#include <stdio.h>// Hlavičkový soubor pro standardní vstup/výstup

#include "NeHeGL.h"// Hlavičkový soubor pro NeHeGL
```

Nadefinujeme pár struktur, které nám pomohou při ukládání dat. První z těchto struktur je tagMATRIX. Pokud se na ni podíváte, zjistíte, že ukládá matici jako jednorozměrné pole 16ti floatů místo toho, aby to bylo dvourozměrné pole 4x4. To je proto, že OpenGL pracuje taky s jednorozměrným polem. Pokud bychom použili 4x4, hodnoty by byly ve špatném pořadí.

```
typedef struct tagMATRIX// Ukládá OpenGL matici
{
    float Data[16];// Matice ve formátu OpenGL
}
MATRIX;
```

Další strukturou je vektor, který ukládá jednotlivé x, y, z složky na osách.

```
typedef struct tagVECTOR// Struktura vektoru
{
    float X, Y, Z;// Složky vektoru
}
VECTOR;
```

Třetí je vertexová struktura. Každý vertex se bude skládat z pozice a normály (žádné texturovací koordináty). Složky struktury musí být v uvedeném pořadí, jinak se při loadování stane něco OPRAVDU strašného (sám jsem kvůli tomu

rozsekal celý tento kód, abych našel chybu).

```
typedef struct tagVERTEX// Struktura vertexu
{
    VECTOR Nor;// Normála vertexu
    VECTOR Pos;// Pozice vertexu
}
VERTEX;
```

Nakonec struktura polygonu. Víím, že toto není nejlepší způsob, jak ukládat vertexy, ale pro jednoduchost to stačí. Normálně bych použil pole vertexů a pole polygonů obsahujících indexy vertexů tvořících polygon, ale my to uděláme jinak - vše pro jednoduchost.

```
typedef struct tagPOLYGON// Struktura polygonu
{
    VERTEX Verts[3];// Pole tří vertexů
}
POLYGON;
```

Nádherně jednoduchá část kódu. Přečtete si komentář ke každé proměnné a budete vědět, proč jsme ji deklarovali.

```
bool outlineDraw = true;// Flag pro vykreslování obrysu
bool outlineSmooth = false;// Flag pro vyhlazování čar
float outlineColor[3] = { 0.0f, 0.0f, 0.0f };// Barva čar
float outlineWidth = 3.0f;// Tloušťka čar

VECTOR lightAngle;// Směr světla
bool lightRotate = false;// Flag oznamující zda rotujeme světlem

float modelAngle = 0.0f;// Úhel natočení objektu na ose y
bool modelRotate = false;// Flag na otáčení modelem

POLYGON* polyData = NULL;// Data polygonů
int polyNum = 0;// Počet polygonů

GLuint shaderTexture[1];// Místo pro jednu texturu
```

Model je uložen úplně nejjednodušším způsobem. Prvních pár bajtů obsahuje počet polygonů tvořících objekt a zbytek souboru je pole tagPOLYGON struktur. Proto může následující funkce data přímo načíst bez jakéhokoliv dalšího upravování.

```
BOOL ReadMesh()// Načte obsah souboru model.txt
{
    FILE *In = fopen("Data\\model.txt", "rb");// Otevře soubor

    if (!In)// Kontrola chyby otevření
        return FALSE;

    fread(&polyNum, sizeof(int), 1, In);// Načte hlavičku souboru (počet vertexů)

    polyData = new POLYGON[polyNum];// Alokace paměti
    fread(&polyData[0], sizeof(POLYGON) * polyNum, 1, In);// Načte všechna data

    fclose(In);// Zavře soubor
    return TRUE;// Loading objektu úspěšný
}
```

Funkce DotProduct() spočítá úhel mezi dvěma vektory nebo rovinami. Funkce Magnitude() spočítá délku vektoru a funkce Normalize() upraví vektor na jednotkovou délku.

```
inline float DotProduct(VECTOR &V1, VECTOR &V2)// Spočítá odchylku dvou vektorů
{
    return V1.X * V2.X + V1.Y * V2.Y + V1.Z * V2.Z;// Vrátí úhel
}

inline float Magnitude(VECTOR &V)// Spočítá délku vektoru
{
    return sqrtf(V.X * V.X + V.Y * V.Y + V.Z * V.Z);// Vrátí délku vektoru
}

void Normalize(VECTOR &V)// Vytvoří jednotkový vektor
{
    float M = Magnitude(V);// Spočítá aktuální délku vektoru
```

```

if (M != 0.0f) // Proti dělení nulou
{
    V.X /= M; // Normalizování jednotlivých složek
    V.Y /= M;
    V.Z /= M;
}
}

```

Funkce RotateVector() pootočí vektor podle zadané matice. Všimněte si, že vektor pouze otočí, ale už nic nedělá s jeho pozicí. Funkce se používá pro otáčení normál, aby zajistila, že normály budou při počítání osvětlení ukazovat správným směrem.

```

void RotateVector(MATRIX &M, VECTOR &V, VECTOR &D) // Rotace vektoru podle zadané matice
{
    D.X = (M.Data[0] * V.X) + (M.Data[4] * V.Y) + (M.Data[8] * V.Z); // Otočení na x
    D.Y = (M.Data[1] * V.X) + (M.Data[5] * V.Y) + (M.Data[9] * V.Z); // Otočení na y
    D.Z = (M.Data[2] * V.X) + (M.Data[6] * V.Y) + (M.Data[10] * V.Z); // Otočení na z
}

```

První významnější funkcí tohoto engineu je Initialize(), která provádí to, co je z jejího názvu zjevné - inicializaci.

```

BOOL Initialize (GL_Window* window, Keys* keys) // Uživatelská a OpenGL inicializace
{
    int i; // Řídící proměnná cyklů

```

Následující 3 proměnné jsou použity pro načtení shader souboru. Line obsahuje jeden řádek řetězce a pole shaderData ukládá hodnoty pro shading. Používáme 96 hodnot namísto 32, protože potřebujeme převést stupně šedi na hodnoty RGB, aby s nimi mohlo OpenGL pracovat. Můžeme sice hodnoty uložit jako stupně šedi, ale bude jednodušší když při nahrávání textury použijeme stejné hodnoty pro jednotlivé složky RGB.

```

    char Line[255]; // Pole 255 znaků
    float shaderData[32][3]; // Pole 96 shader hodnot

    FILE *In = NULL; // Ukazatel na soubor

```

Klasické nastavení engineu a OpenGL...

```

    g_window = window;
    g_keys = keys;

    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Perspektivní korekce

    glClearColor(0.7f, 0.7f, 0.7f, 0.0f); // Světle šedé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu

    glEnable(GL_DEPTH_TEST); // Zapne testování hloubky
    glDepthFunc(GL_LESS); // Typ testování hloubky

```

Při vykreslování čar chceme, aby byly pěkně vyhlazené. Implicitně je tato funkce vypnuta, ale stiskem klávesy 2 ji můžeme zapínat a vypínat podle libosti.

```

    glShadeModel(GL_SMOOTH); // Jemné stínování
    glDisable(GL_LINE_SMOOTH); // Vypne vyhlazování čar

```

Zapneme ořezávání vnitřních stěn objektu, které stejně nejsou vidět a vypneme OpenGL světla, protože potřebné výpočty provedeme po svém.

```

    glEnable(GL_CULL_FACE); // Zapne face culling (ořezávání stěn)
    glDisable(GL_LIGHTING); // Vypne světla

```

V další části kódu načteme shader soubor. Obsahuje pouze 32 desetinných čísel uložených, pro jednoduchou modifikaci, v ASCII formátu, každé na samostatném řádku.

```

    In = fopen("Data\\shader.txt", "r"); // Otevření shader souboru

    if (In) // Kontrola, zda je soubor otevřen
    {
        for (i = 0; i < 32; i++) // Projde všech 32 hodnot ve stupních šedi
        {
            if (feof(In)) // Kontrola konce souboru
                break;

            fgets(Line, 255, In); // Získání aktuálního řádku

```

Přeměníme načtené stupně šedi na RGB, jak jsme si popsali výše.

```
        // Zkopíruje danou hodnotu do všech složek barvy
        shaderData[i][0] = shaderData[i][1] = shaderData[i][2] = float(atof(Line));
    }

    fclose(In); // Zavře soubor
}
else
{
    return FALSE; // Neúspěch
}
```

Nahrajeme texturu přesně tak, jak je. Bez použití filtrování, jinak by výsledek vypadal opravdu hrozně, přinejmenším. Použijeme GL_TEXTURE_1D, protože jde o jednorozměrné pole hodnot.

```
glGenTextures(1, &shaderTexture[0]); // Získání ID textury

glBindTexture(GL_TEXTURE_1D, shaderTexture[0]); // Přiřazení textury; od teď je 1D
texturou

// Nikdy nepoužívejte bi-/trilineární filtrování!
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexImage1D(GL_TEXTURE_1D, 0, GL_RGB, 32, 0, GL_RGB, GL_FLOAT, shaderData); //
Upload dat
```

Nastavíme směr dopadání světla na objekt ze směru kladné části osy z. Ve svém důsledku to znamená, že světlo bude zepředu svítit na model.

```
lightAngle.X = 0.0f; // Nastavení směru x
lightAngle.Y = 0.0f; // Nastavení směru y
lightAngle.Z = 1.0f; // Nastavení směru z

Normalize(lightAngle); // Normalizování vektoru světla
```

Načtení tvaru ze souboru (funkce popsána výše).

```
return ReadMesh(); // Vrátí návratovou hodnotu funkce ReadMesh()
}
```

Funkce Deinitialize() je pravým opakem předchozí funkce. Smaže texturu a data polygonů nahraná pomocí funkcí Initialize() a ReadMesh().

```
void Deinitialize(void) // Deinicilizace
{
    glDeleteTextures(1, &shaderTexture[0]); // Smaže shader texturu
    delete [] polyData; // Uvolní data polygonů
}
```

Funkce Update() se periodicky volá v hlavní smyčce tohoto dema. Jedinou její funkcí je zpracování vstupu z klávesnice.

```
void Update(DWORD milliseconds) // Aktualizace scény (objektu)
{
    if (g_keys->keyDown [VK_ESCAPE] == TRUE) // Klávesa ESC
    {
        TerminateApplication (g_window); // Ukončení programu
    }

    if (g_keys->keyDown [VK_F1] == TRUE) // Klávesa F1
    {
        ToggleFullscreen(g_window); // Přepnutí módů fullscreen/okno
    }

    if (g_keys->keyDown [' '] == TRUE) // Mezerník
    {
        modelRotate = !modelRotate; // Zapne/vypne rotaci objektu

        g_keys->keyDown [' '] = FALSE;
    }

    if (g_keys->keyDown ['1'] == TRUE) // Klávesa čísla 1
```



```

{
    outlineDraw = !outlineDraw;// Zapne/vypne vykreslování obrysu
    g_keys->keyDown ['1'] = FALSE;
}
if (g_keys->keyDown ['2'] == TRUE)// Klávesa číslo 2
{
    outlineSmooth = !outlineSmooth;// Zapne/vypne anti-aliasing
    g_keys->keyDown ['2'] = FALSE;
}
if (g_keys->keyDown [VK_UP] == TRUE)// Šipka nahoru
{
    outlineWidth++;// Zvětší tloušťku čáry
    g_keys->keyDown [VK_UP] = FALSE;
}
if (g_keys->keyDown [VK_DOWN] == TRUE)// Šipka dolů
{
    outlineWidth--;// Zmenší tloušťku čáry
    g_keys->keyDown [VK_DOWN] = FALSE;
}
if (modelRotate)// Je rotace zapnutá
    modelAngle += (float)(milliseconds) / 10.0f;// Aktualizace úhlu natočení v
    závislosti na FPS
}

```

Funkce, na kterou už určitě netrpělivě čekáte. Draw() provádí většinu nejdůležitější práce v tomto tutoriálu - počítá hodnoty stínu, renderuje daný tvar a renderuje obrys.

```

void Draw(void)// Vykreslování
{
    int i, j;// Řídící proměnné cyklů

```

Proměnná TmpShade se použije na uložení dočasné hodnoty stínu pro aktuální vertex. Všechna data týkajícího se jednoho vertexu jsou spočítána ve stejném čase, což znamená, že můžeme použít jen jednu proměnnou, kterou postupně použijeme pro všechny vertexy. Struktury TmpMatrix, TmpVector a TmpNormal jsou také použity pro spočítání dat jednoho vertexu. TmpMatrix se nastaví vždy jednou při startu funkce Draw() a nezmění se až do jejího dalšího startu. TmpVector a TmpNormal se liší vertex od vertexu.

```

float TmpShade;// Dočasná hodnota stínu
MATRIX TmpMatrix;// Dočasná MATRIX struktura
VECTOR TmpVector, TmpNormal;// Dočasné VECTOR struktury

```

Po deklaraci proměnných vymažeme buffery a data OpenGL matice.

```

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Vymaže buffery
glLoadIdentity();// Reset matice

```

Nejdříve zkontrolujeme zda chceme obrys vyhlazený. Když ano, zapneme anti-aliasing. Když ne, tak ho vypneme. Jak jednoduché...

```

if (outlineSmooth)// Chce uživatel vyhlazené čáry?
{
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);// Použije nejkvalitnější výpočty
    glEnable(GL_LINE_SMOOTH);// Zapne anti-aliasing
}
else// Nechce
{
    glDisable(GL_LINE_SMOOTH);// Vypne anti-aliasing
}

```

Posunutím kamery o 2 jednotky dozadu nastavíme pohled, potom model pootočíme o daný úhel. Poznámka: protože jsme nejdříve pohnuli s kamerou, model se bude točit na místě. Pokud bychom to udělali opačně, model by rotoval kolem kamery.

```

glTranslatef(0.0f, 0.0f, -2.0f);// Posun do hloubky

```

```
glRotatef(modelAngle, 0.0f, 1.0f, 0.0f); // Rotace objektem na ose y
```

Získáme nově vytvořenou OpenGL matici a uložíme ji do TmpMatrix.

```
glGetFloatv(GL_MODELVIEW_MATRIX, TmpMatrix.Data); // Získání matice
```

Kouzla začínají... Povolíme 1D texturování a použijeme texturu stínu. Potom nastavíme barvu modelu. Vybral jsem bílou, protože na ní jde lépe vidět světlo a stín než na ostatních barvách. Nejméně vhodná je zcela určitě černá.

```
// Kód Cel-Shadingu
glEnable(GL_TEXTURE_1D); // Zapne 1D texturování
glBindTexture(GL_TEXTURE_1D, shaderTexture[0]); // Zvolí texturu

glColor3f(1.0f, 1.0f, 1.0f); // Nastavení barvy modelu (bílá)
```

Začneme s kreslením trojúhelníků. Projdeme všechny polygony v poli a všechny vertexy každého z těchto polygonů. Nejdříve zkopírujeme normálu do dočasné struktury. Díky tomu můžeme hodnotami normály otáčet bez toho, že bychom ztratili původní data (bez průběžné degradace).

```
glBegin(GL_TRIANGLES); // Začátek kreslení trojúhelníků

for (i = 0; i < polyNum; i++) // Prochází jednotlivé polygony
{
    for (j = 0; j < 3; j++) // Prochází jednotlivé vertexy
    {
        // Zkopírování aktuální normály do dočasné struktury
        TmpNormal.X = polyData[i].Verts[j].Nor.X;
        TmpNormal.Y = polyData[i].Verts[j].Nor.Y;
        TmpNormal.Z = polyData[i].Verts[j].Nor.Z;
```

Otočíme vektor o matici, kterou jsme získali od OpenGL a normalizujeme ho.

```
RotateVector(TmpMatrix, TmpNormal, TmpVector); // Otočí vektor podle
matice

Normalize(TmpVector); // Normalizace normály
```

Spočítáme odchylku pootočené normály a směru světla. Potom hodnotu dáme do rozmezí 0-1 (z původního -1 až 1).

```
TmpShade = DotProduct(TmpVector, lightAngle); // Spočítání hodnoty stínu

if (TmpShade < 0.0f) // Pokud je TmpShade menší než nula bude se rovnat
nule
    TmpShade = 0.0f;
```

Předáme tuto hodnotu OpenGL jako texturovací souřadnici. Potom předáme pozici vertexu a opakujeme. A opakujeme. A opakujeme. Myslím, že podstatu už chápete.

```
glTexCoord1f(TmpShade); // Nastavení texturovací souřadnice na hodnotu
stínu
glVertex3fv(&polyData[i].Verts[j].Pos.X); // Pošle pozici vertexu
}
}

glEnd(); // Konec kreslení

glDisable(GL_TEXTURE_1D); // Vypne 1D texturování
```

Přesuneme se k obrysům. Obrys můžeme definovat jako hranu, kde je jeden polygon přivrácen směrem k nám a druhý od nás. Použijeme pro OpenGL běžné testování hloubky - méně nebo stejně (GL_LEQUAL) a také nastavíme vyřazování všech polygonů otočených k nám. Také použijeme blending, aby to trochu vypadalo.

Nastavíme OpenGL tak, aby polygony čelem od nás vyrenderoval jako čáry. Vyřadíme všechny polygony čelem k nám a nastavíme testování hloubky na menší nebo stejné na aktuální ose Z. Potom ještě nastavíme barvu čar, projdeme všechny polygony a vykreslíme jejich rohy. Stačí zadat pozici. Nemusíme zadávat normálu a stíny, protože chceme jenom obrys.

```
// Kód pro vykreslení obrysů
if (outlineDraw) // Chceme vůbec kreslit obrys?
{
    glEnable(GL_BLEND); // Zapne blending
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // Mód blendingu

    glPolygonMode(GL_BACK, GL_LINE); // Odvrácené polygony se stanout pouze
```

```

obrysovými čarami
glLineWidth(outlineWidth); // Nastavení šířky čáry

glCullFace(GL_FRONT); // Nerenderovat převrácené polygony
glDepthFunc(GL_EQUAL); // Mód testování hloubky

glColor3fv(&outlineColor[0]); // Barva obrysu (černá)

glBegin(GL_TRIANGLES); // Začátek kreslení trojúhelníků
    for (i = 0; i < polyNum; i++) // Prochází jednotlivé polygony
    {
        for (j = 0; j < 3; j++) // Prochází jednotlivé vertexy
        {
            glVertex3fv(&polyData[i].Verts[j].Pos.X); // Pošle pozici vertexu
        }
    }

glEnd(); // Konec kreslení

```

Na konci už jenom vrátíme nastavení do původního stavu a ukončíme funkci i tutoriál.

```

glDepthFunc(GL_LESS); // Testování hloubky na původní nastavení
glCullFace(GL_BACK); // Nastavení ořezávání na původní hodnotu
glPolygonMode(GL_BACK, GL_FILL); // Normální vykreslování
glDisable(GL_BLEND); // Vypne blending
}
}

```

**napsal: Sami "MENTAL" Hamlaoui <disk_disaster (zavináč) hotmail.com>
teoretickou část přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>
praktickou část přeložil: Václav Slováček - Wessan <horizont (zavináč) host.sk>**

Lekce 38 - Nahrávání textur z resource souboru & texturování trojúhelníků

Tento tutoriál jsem napsal pro všechny z vás, kteří se mě v emailech dotazovali na to "Jak mám loadovat texturu ze zdrojů programu, abych měl všechny obrázky uložené ve výsledném .exe souboru?" a také pro ty, kteří psali "Vím, jak otexturovat obdélník, ale jak mapovat na trojúhelník?" Tutoriál není, oproti jiným, extrémně pokročilý, ale když nic jiného, tak se naučíte, jak skrýt vaše precizní textury před okem uživatele. A co víc - budete moci trochu ztížit jejich kradení :-)

Tak už víte, jak otexturovat čtverec, jak nahrát bitmapu, tga,... Tak jak kruci otexturovat trojúhelník? A co když chci textury ukrýt do .exe souboru? Když zjistíte, jak je to jednoduché, budete se divit, že vás řešení už dávno nenapadlo.

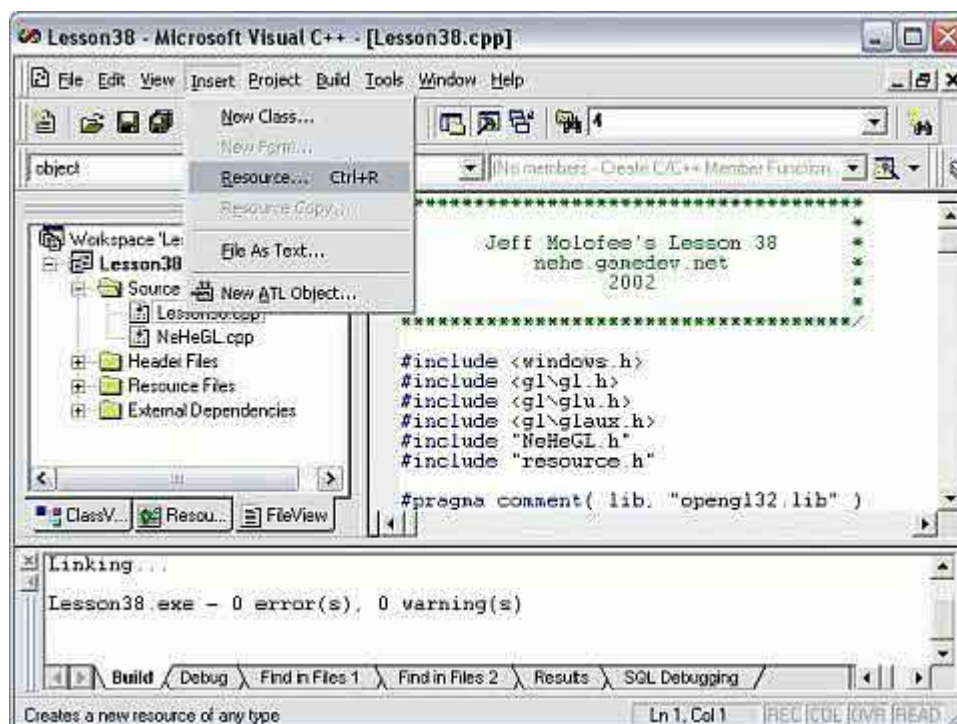
Raději než abych vše do detailů vysvětloval, předvedu pár screenshotů, takže budete přesně vědět, o čem mluvím. Budu používat nejnovější základní kód, který si můžete na <http://nehe.gamedev.net/> pod nadpisem "NeHeGL Basecode" a nebo kliknutím na odkaz na konci tohoto tutoriálu.

První co potřebujeme udělat, je přidat obrázky do zdrojového souboru (resource file). Mnoho z vás už zjistilo, jak to udělat, ale naneštěstí jste často opomínali několik kroků, a proto skončili s nepoužitelným zdrojovým souborem naplněným bitmapami, které nejdou použít.

Tento tutoriál je napsán pro Visual C++ 6.0. Pokud používáte něco jiného, tato část tutoriálu je pro vás zbytečná, obzvláště obrázky prostředí Visual C++.

Momentálně budete schopni nahrát pouze 24-bitové BMP. K nahrání 8-bitového BMP bychom potřebovali mnoho kódu navíc. Rád bych věděl o někom, kdo má malý optimalizovaný BMP loader. Kód, který mám k současnému načítání 8 a 24-bitových BMP je prostě příšerný. Něco, co používá LoadImage, by se hodilo.

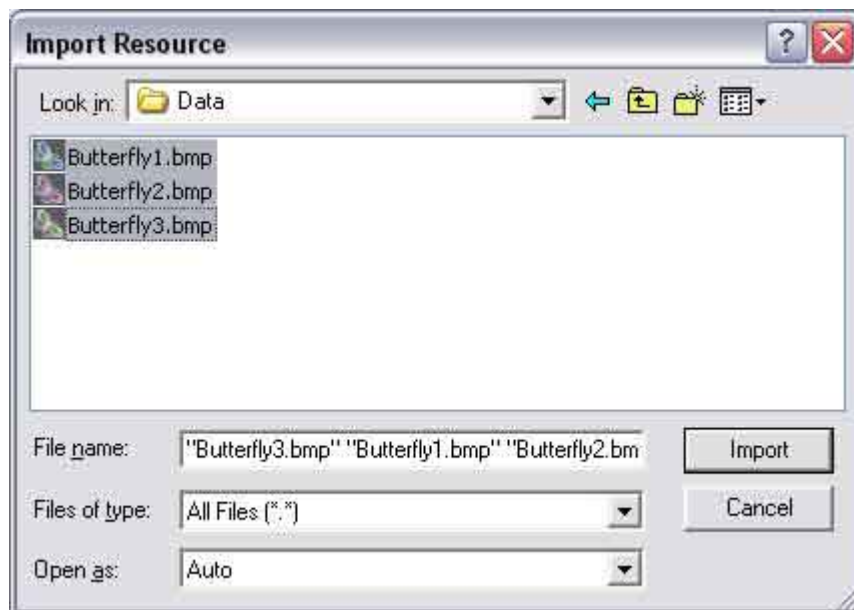
Tak tedy začneme...



Otevřete projekt a vyberte z hlavního menu Insert->Resource.



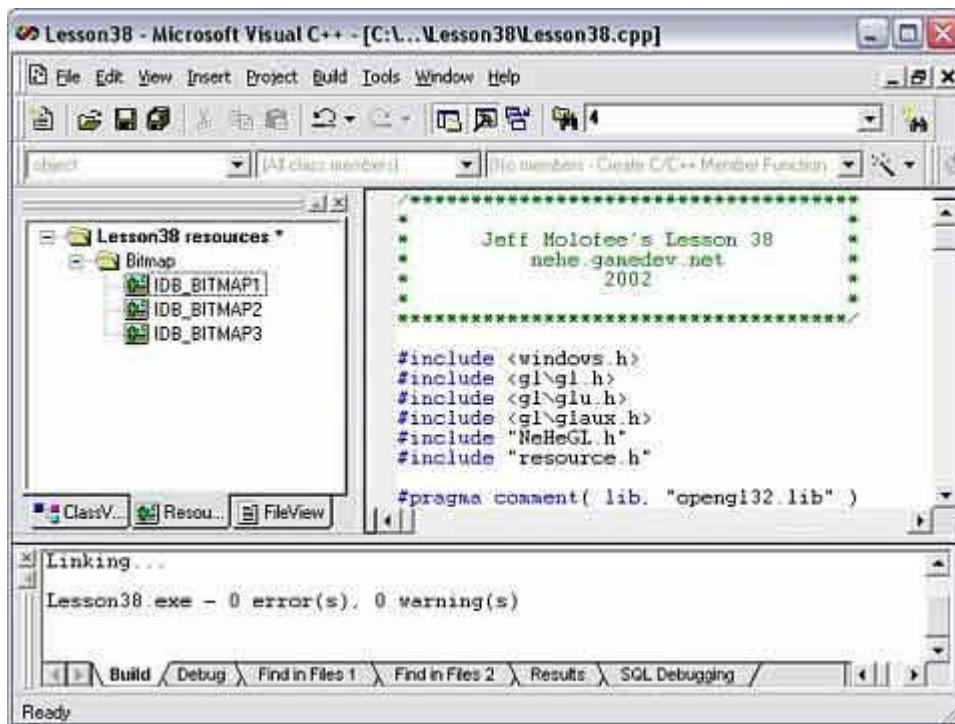
Jste dotázáni na typ zdroje, který si přejete importovat. Vyberte Bitmap a klikněte na tlačítko Import.



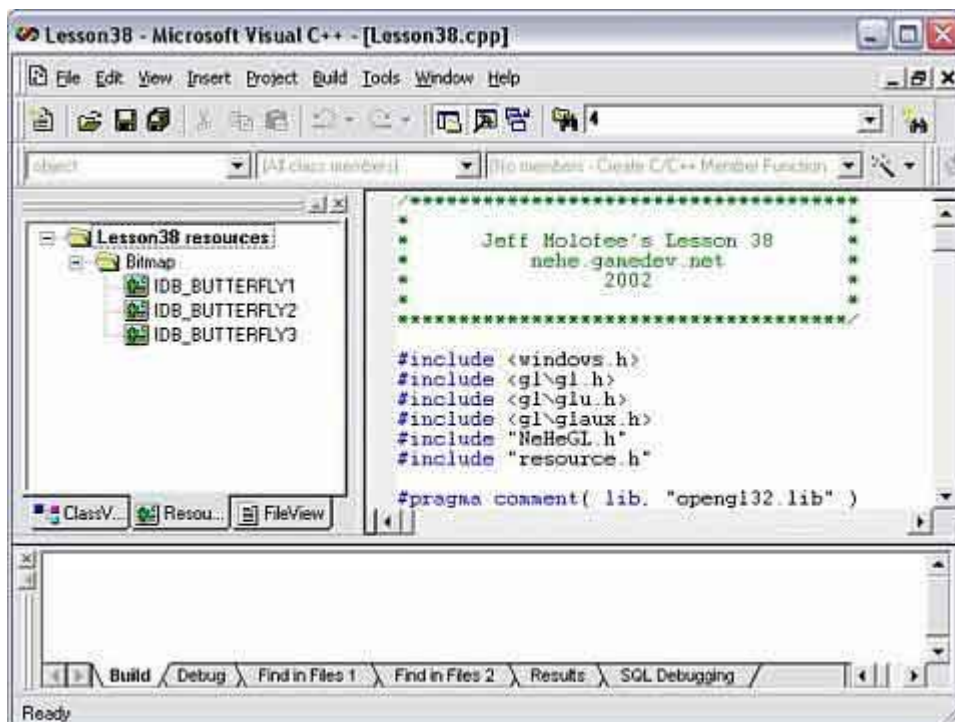
Otevře se prohlížeč souborů. Vstupte do složky Data a označte všechny 3 bitmapy (podržte Ctrl když je budete označovat). Pak klikněte na tlačítko Import. Pokud nevidíte soubory bitmap, ujistěte se, že v poli Files of type je vybráno All Files(*.*).



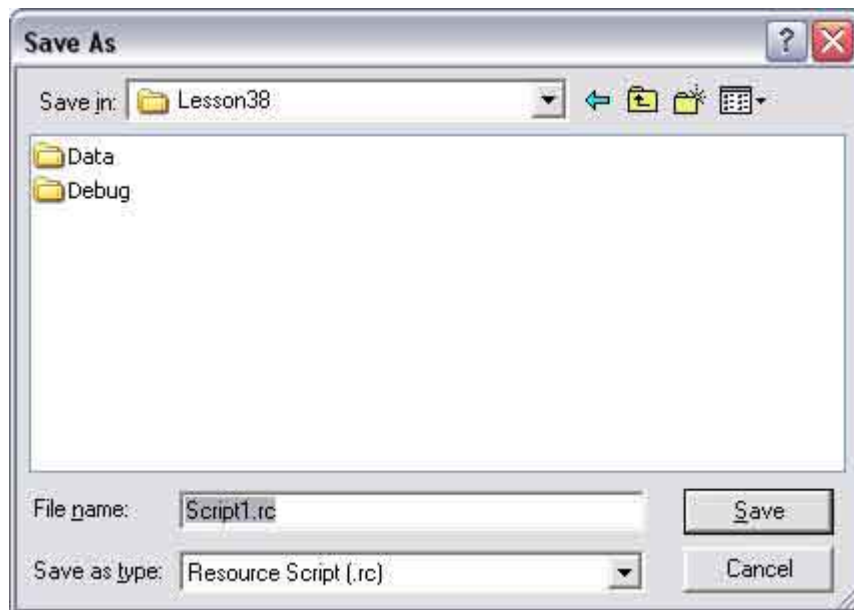
Třikrát se zobrazí varovná zpráva (jednou za každý obrázek). Vše co vám říká je, že obrázky byly v pořádku importovány, ale nemůžete je upravovat, protože mají více než 256 barev. Žádný důvod ke starostem!



Když jsou všechny obrázky importovány, zobrazí se jejich seznam. Každá bitmapa dostane své identifikační jméno (ID), které začíná na IDB_BITMAP a následuje číslo 1 - 3. Pokud jste líní, mohli byste to nechat tak a vrhnout se na kód této lekce. (My ale nejsme líní!

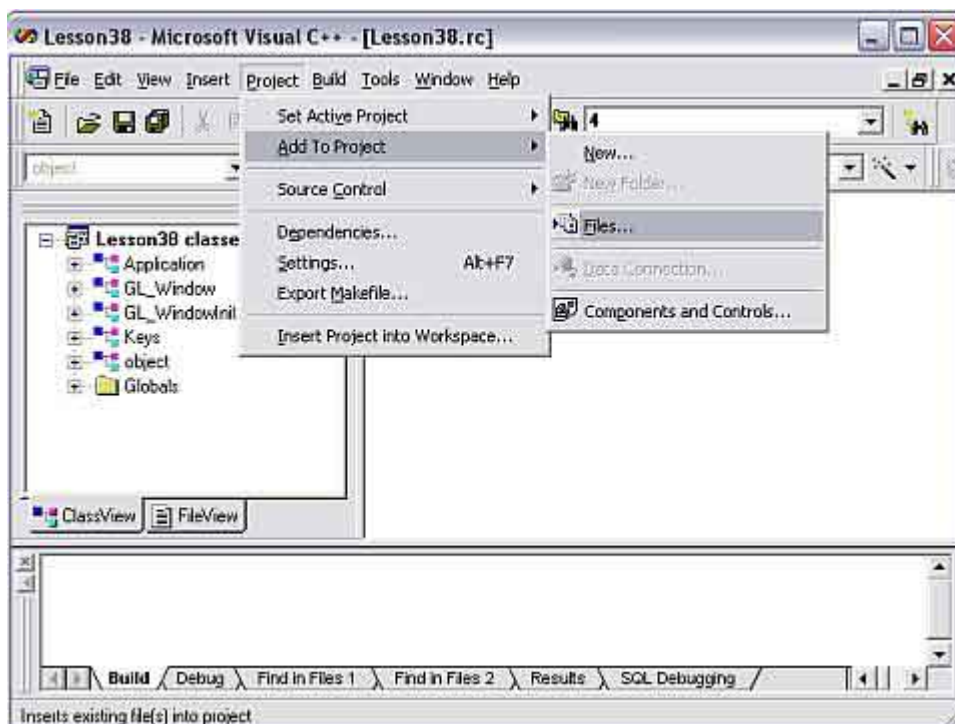


Pravým tlačítkem klikněte na každé ID a vyberte z menu položku Properties. Přejmenujte identifikační jména na původní názvy souborů.

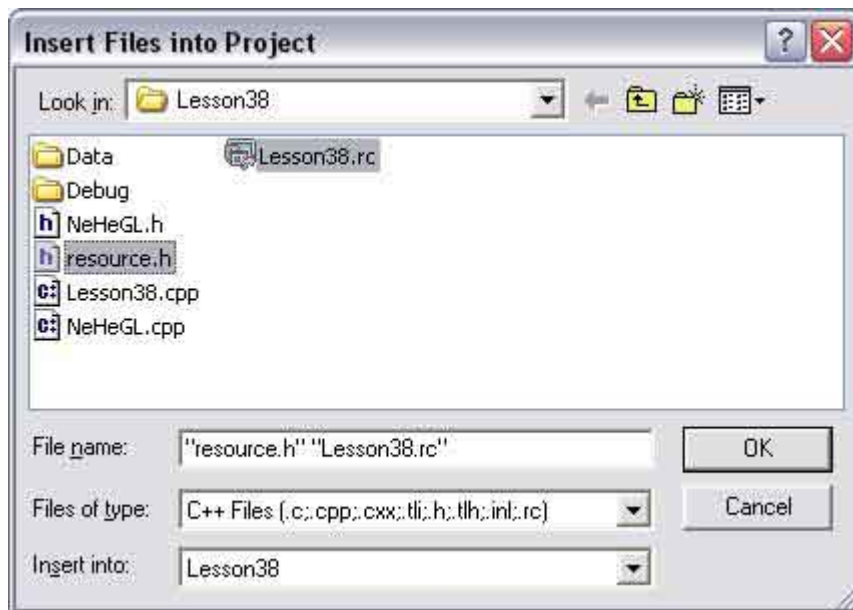


Teď, když jsme hotovi, vyberte z hlavního menu File->Save All. Protože jste právě vytvořili nový zdrojový soubor, budete dotázáni na to, jak chcete soubor pojmenovat. Můžete soubor pojmenovat, jak chcete. Jakmile vyplníte jméno souboru klikněte na tlačítko Save.

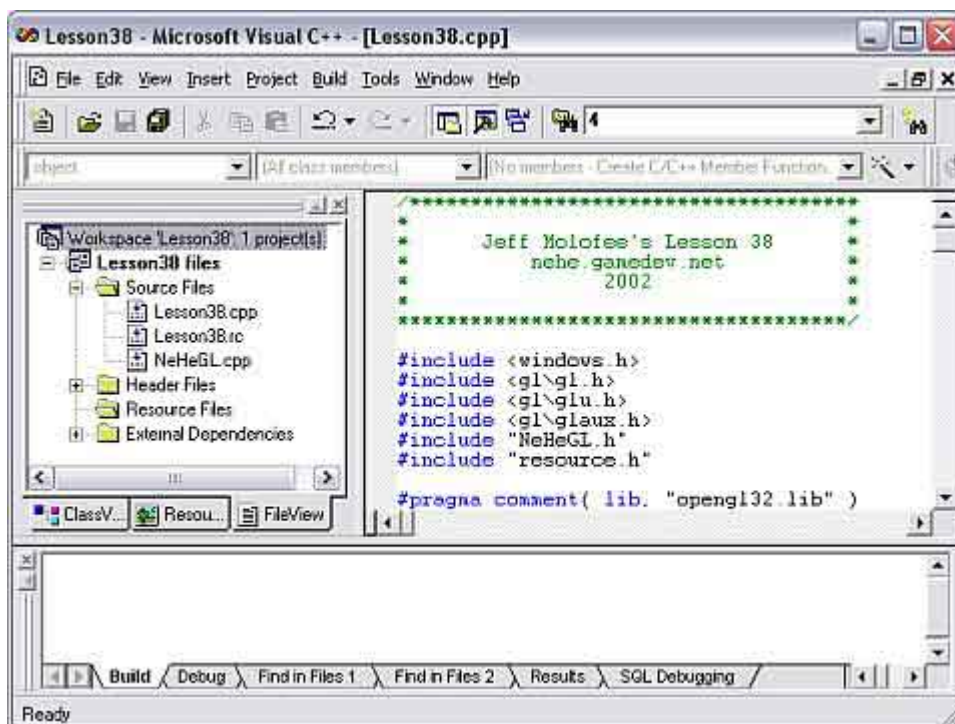
Až sem se hodně z vás propracovalo. Máte zdrojový soubor plný bitmapových obrázků a už jste ho i uložili na disk. Abyste však obrázky mohli použít, musíte udělat ještě pár věcí.



Dále musíte přidat soubor se zdroji do aktuálního projektu. Z hlavního menu vyberte Project->Add To Project->Files.



Vyberte resource.h a váš zdrojový soubor s bitmapami. Podržte Ctrl pro výběr více souborů, nebo je přidejte samostatně.



Poslední věc, kterou je třeba udělat, je kontrola, zda je zdrojový soubor ve složce Resource Files. Jak vidíte na obrázku, byl přidán do složky Source Files. Klikněte na něho a přetáhněte ho do složky Resource Files.

Když je vše hotovo. Vyberte z hlavního menu File->Save All. Máme to těžší za sebou!

Vrhneme na kód! Nejdůležitější řádek v kódu je #include "resource.h". Bez tohoto řádku vám kompilér při kompilování vrátí chybu "undeclared identifier". Resource.h umožňuje přístup k importovaným obrázkům.

```
#include <windows.h> // Hlavičkový soubor pro Windows
#include <gl\gl.h> // Hlavičkový soubor pro OpenGL32 knihovnu
#include <gl\glu.h> // Hlavičkový soubor pro GLu32 knihovnu
#include <gl\glaux.h> // Hlavičkový soubor pro GLaux knihovnu

#include "NeHeGL.h" // Hlavičkový soubor pro NeHeGL
#include "resource.h" // Hlavičkový soubor pro Resource (*DŮLEŽITÉ*)
```



```

#pragma comment( lib, "opengl32.lib" )// Přilinkuje OpenGL32.lib
#pragma comment( lib, "glu32.lib" )// Přilinkuje GLu32.lib
#pragma comment( lib, "glaux.lib" )// Přilinkuje GLaux.lib

#ifndef CDS_FULLSCREEN// Pokud ještě CDS_FULLSCREEN není definován
#define CDS_FULLSCREEN 4// Tak ho nadefinujeme
#endif// Vyhne se tak možným chybám

GL_Window* g_window;
Keys* g_keys;

GLuint texture[3];// Místo pro 3 textury

```

Následující struktura bude obsahovat informace o motýlku, se kterým budeme pohybovat po obrazovce. Tex určuje, jakou texturu na objekt namapujeme. X, y a z udávají pozici objektu v prostoru. Yi bude náhodné číslo udávající, jak rychle motýl padá k zemi. Spinz se při pádu použije na otáčení okolo osy z. Spinzi udává rychlost této rotace. Flap bude použito pro mávání křídly (k tomu se později ještě vrátíme). Fi bude udávat jak rychle objekt mává křídly.

```

struct object// Struktura nazvaná object
{
    int tex;// Kterou texturu namapovat
    float x;// X Pozice
    float y;// Y Pozice
    float z;// Z Pozice
    float yi;// Rychlost pádu
    float spinz;// Úhel otočení kolem osy z
    float spinzi;// Rychlost otáčení kolem osy z
    float flap;// Mávání křídly
    float fi;// Směr mávání
};

```

Vytvoříme padesát těchto objektů pojmenovaných obj[index].

```
object obj[50];// Vytvoří 50 objektů na bázi struktury
```

Následující část kódu nastavuje náhodné hodnoty všem objektům. Loop se bude pohybovat mezi 0 - 49 (celkem 50 objektů). Nejdříve vybereme náhodnou texturu od 0 do 2, aby nebyli všichni stejní. Potom nastavíme náhodnou pozici x od -17.0f do 17.0f. Počáteční pozice y bude 18.0f. Tím zajistíme, že se objekt vytvoří mimo obrazovku, takže ho nevidíme úplně od začátku. Pozice z je rovněž náhodná hodnota od -10.0f do -40.0f. Spinzi opět je náhodná hodnota od -1.0f do 1.0f. Flap nastavíme na 0.0f (křídla budou přesně uprostřed). Fi a yi nastavíme taky na náhodné hodnoty.

```

void SetObject(int loop)// Nastavení základních vlastností objektu
{
    obj[loop].tex = rand() % 3;// Výběr jedné ze tří textur

    obj[loop].x = rand() % 34 - 17.0f;// Náhodné x od -17.0f do 17.0f
    obj[loop].y = 18.0f;// Pozici y nastavíme na 18 (nad obrazovku)
    obj[loop].z = -(rand() % 30000 / 1000.0f) + 10.0f;// Náhodné z od -10.0f do -40.0f

    obj[loop].spinzi = (rand() % 10000) / 5000.0f - 1.0f;// Spinzi je náhodné číslo od -1.0f do 1.0f
    obj[loop].flap = 0.0f;// Flap začne na 0.0f

    obj[loop].fi = 0.05f + (rand() % 100) / 1000.0f;// Fi je náhodné číslo od 0.05f do 0.15f
    obj[loop].yi = 0.001f + (rand() % 1000) / 10000.0f;// Yi je náhodné číslo od 0.001f do 0.101f
}

```

Teď k té zábavnější části. Nahrání bitmapy ze zdrojového souboru a její přeměna na texturu. hBMP je ukazatel na soubor s bitmapami. Řekne našemu programu odkud má brát data. BMP je bitmapová struktura, do které můžeme uložit data z našeho zdrojového souboru.

```

void LoadGLTextures();// Vytvoří textury z bitmap ve zdrojovém souboru
{
    HBITMAP hBMP;// Ukazatel na bitmapu
    BITMAP BMP;// Struktura bitmapy

```

Řekneme jaké identifikační jména chceme použít. Chceme nahrát IDB_BUTTEFLY1, IDB_BUTTEFLY2 a IDB_BUTTERFLY3. Pokud chcete přidat více obrázků, připište jejich ID.

```
byte Texture[] = { IDB_BUTTERFLY1, IDB_BUTTERFLY2, IDB_BUTTERFLY3 };// ID bitmap,
```

které chceme načíst

Na dalším řádku použijeme sizeof(Texture) na zjištění, kolik textur chceme sestavit. V Texture[] máme zadány 3 identifikační čísla, takže výsledkem sizeof(Texture) bude hodnota bude 3.

```
glGenTextures(sizeof(Texture), &texture[0]); // Vygenerování tří textur, sizeof
(Texture) = 3 ID

for (int loop = 0; loop < sizeof(Texture); loop++) // Projde všechny bitmapy ve
zdrojích
{
```

LoadImage() přijímá parametry GetModuleHandle(NULL) - handle instance. MAKEINTRESOURCE(Texture[loop]) přemění hodnotu celého čísla Texture[loop] na hodnotu zdroje (obrázku, který má být načten). Tady je nutné poznamenat, že sice používáme identifikační jméno např. IDB_BUTTERFLY1, ale v souboru Resource.h je napsáno něco ve stylu #define IDB_BUTTERFLY1 115, my se tím ale nemusíme vůbec zabývat. Vývojové prostředí vše automatizuje. IMAGE_BITMAP říká našemu programu, že zdroj, který chceme načíst je bitmapový obrázek.

Další dva parametry (0,0) jsou požadovaná výška a šířka obrázku. Chceme použít implicitní velikost, tak nastavíme obě na 0. Poslední parametr (LR_CREATEDIBSECTION) vrátí DIB část mapy, která obsahuje jen bitmapu bez informací o barvách v hlavičce. Přesně to, co chceme.

hBMP bude ukazatelem na naše bitmapová data nahraná pomocí LoadImage().

```
hBMP = (HBITMAP) LoadImage(GetModuleHandle(NULL), MAKEINTRESOURCE(Texture
[loop]), IMAGE_BITMAP, 0, 0, LR_CREATEDIBSECTION); // Nahraje bitmapu ze zdrojů
```

Dále zkontrolujeme, zda pointer hBMP opravdu ukazuje na data. Pokud byste chtěli přidat ošetření chyb, můžete zkontrolovat hBMP a zobrazit chybové hlášení. Pokud ale data existují, použijeme funkci getObject() na získání všech dat o velikosti sizeof(BMP) a jejich uložení do bitmapové struktury &BMP.

```
if (hBMP) // Pokud existuje bitmapa
{
    GetObject(hBMP, sizeof(BMP), &BMP); // Získání objektu
```

glPixelStorei() oznámí OpenGL, že data jsou uložena ve formátu 4 byty na pixel. Nastavíme filtrování na GL_LINEAR a GL_LINEAR_MIPMAP_LINEAR (kvalitní a vyhlazené) a vygenerujeme texturu.

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 4); // 4 byty na jeden pixel
glBindTexture(GL_TEXTURE_2D, texture[loop]); // Zvolí texturu

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Lineární
filtrování
glTexParameteri
(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR); //
Mipmapované lineární filtrování
```

Všimněte si, že používáme BMP.bmWidth a BMP.bmHeight, abychom získali výšku a šířku bitmapy. Také musíme použitím GL_BGR_EXT prohodit červenou a modrou barvu. Data získáme z BMP.bmBits.

```
// Vygenerování mipmapované textury (3 byty, šířka, výška a BMP data)
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, BMP.bmWidth, BMP.bmHeight, GL_BGR_EXT,
GL_UNSIGNED_BYTE, BMP.bmBits);
```

Posledním krokem je smazání objektu bitmapy, abychom uvolnili všechny systémové prostředky spojené s tímto objektem.

```
DeleteObject(hBMP); // Smaže objekt bitmapy
}
}
}
```

V inicializačním kódu není nic moc zajímavého. Použijeme funkci LoadGLTextures(), abychom zavolali kód, který jsme právě napsali. Nastavíme pozadí na černou barvu. Vyřadíme depth testing (jednoduchý blending). Povolíme texturování, nastavíme a povolíme blending.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Inicializační kód a nastavení
{
    g_window = window;
    g_keys = keys;

    LoadGLTextures(); // Nahraje textury ze zdrojů
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
```

```

glClearDepth(1.0f); // Nastavení hloubkového bufferu
glDepthFunc(GL_LEQUAL); // Typ hloubkového testování
glDisable(GL_DEPTH_TEST); // Vypnutí hloubkového testování

glShadeModel(GL_SMOOTH); // Vyhlazené stínování
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Výpočet perspektivy na nejvyšší
kvalitu

glEnable(GL_TEXTURE_2D); // Povolí texturové mapování

glBlendFunc(GL_ONE, GL_SRC_ALPHA); // Nastavení blendingu (nenáročný / rychlý)
glEnable(GL_BLEND); // Povolení blendingu

```

Hned na začátku potřebujeme inicializovat 50 objektů tak, aby se neobjevily uprostřed obrazovky nebo všechny na stejném místě. I tuto funkci už máme napsanou. Zavoláme ji padesátkrát.

```

for (int loop = 0; loop < 50; loop++) // Inicializace 50 motýlů
{
    SetObject(loop); // Nastavení náhodných hodnot
}

return TRUE; // Inicializace úspěšná
}

```

Deinicializaci tentokrát nevyužijeme.

```

void Deinitialize (void) // Deinicializace
{
}

```

Následující funkce ošetřuje stisk kláves ESC a F1. Periodicky ji voláme v hlavní smyčce programu.

```

void Update (DWORD milliseconds) // Vykonává aktualizace
{
    if (g_keys->keyDown [VK_ESCAPE] == TRUE) // Stisknuta klávesa ESC?
    {
        TerminateApplication(g_window); // Ukončí program
    }

    if (g_keys->keyDown [VK_F1] == TRUE) // Stisknuta klávesa F1?
    {
        ToggleFullscreen(g_window); // Prohodí mód fullscreen/okno
    }
}

```

Teď k vykreslování. Pokusím se vysvětlit nejjednodušší způsob, jak otexturovat jedním obrázkem dva trojúhelníky. Z nějakého důvodu si mnozí myslí, že namapovat texturu na trojúhelník je takřka nemožné. Pravdou je, že s velmi malou námahou můžete otexturovat libovolný tvar. Obrázek může tvaru odpovídat, nebo může být totálně odlišný. Je to úplně jedno.

Tak od začátku... vymažeme obrazovku a deklaruje cyklus na renderování motýlků (objektů).

```

void Draw(void) // Vykreslení scény
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
buffer

    for (int loop = 0; loop < 50; loop++) // Projde 50 motýlků
    {

```

Zavoláme `glLoadIdentity()` pro resetování matice. Pak vybereme texturu, která byla při inicializaci určena pro daný objekt (`obj[loop].tex`). Umístíme motýlka pomocí `glTranslatef()` a otočíme ho o 45 stupňů na ose x. Tím ho natočíme trochu k divákovi, takže nevypadá tak placatě. Nakonec ho ještě otočíme kolem osy z o hodnotu `spinz` - při pádu se bude točit.

```

glLoadIdentity(); // Reset matice

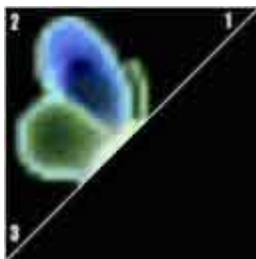
glBindTexture(GL_TEXTURE_2D, texture[obj[loop].tex]); // Zvolí texturu

glTranslatef(obj[loop].x, obj[loop].y, obj[loop].z); // Umístění

glRotatef(45.0f, 1.0f, 0.0f, 0.0f); // Rotace na ose x
glRotatef((obj[loop].spinz), 0.0f, 0.0f, 1.0f); // Rotace na ose y

```

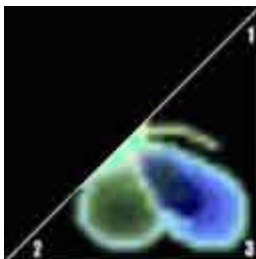
Texturování trojúhelníku se neliší od texturování čtverce. To že máme jen 3 body, neznamená, že nemůžeme čtyřhranným obrázkem otexturovat trojúhelník. Musíme si pouze dávat větší pozor na texturovací souřadnice. V následujícím kódu nakreslíme první trojúhelník. Začneme v pravém horním rohu viditelného čtverce. Pak se přesuneme do levého horního rohu a potom do levého dolního rohu. Kód vyrenderuje následující obrázek:



Všimněte si, že na první trojúhelník se vyrenderuje jen polovina motýla. Druhá část bude pochopitelně na druhém trojúhelníku. Texturovací souřadnice odpovídají tomu, jak jsme texturovali čtverce. Tři souřadnice stačí OpenGL k tomu, aby rozpoznalo jakou část obrázku má na trojúhelník namapovat.

```
glBegin(GL_TRIANGLES); // Kreslení trojúhelníků
// První trojúhelník
glTexCoord2f(1.0f,1.0f); glVertex3f( 1.0f, 1.0f, 0.0f); // Pravý horní bod
glTexCoord2f(0.0f,1.0f); glVertex3f(-1.0f, 1.0f, 0.0f); // Levý horní bod
glTexCoord2f(0.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f); // Levý dolní bod
```

Další část kódu vyrenderuje druhý trojúhelník stejným způsobem jako předtím. Začneme vpravo nahoře, pak půjdeme vlevo dolů a nakonec vpravo dolů.



Druhý bod prvního a třetí bod druhého trojúhelníku se posunují zpět po ose z, aby se vytvořila iluze mávání křídel. To, co se ve skutečnosti děje, je pouze posouvání těchto bodů tam a zpátky od -1.0f do 1.0f, což způsobuje ohýbání v místech, kde má motýl tělo. Pokud se na oba tyto body podíváte, zjistíte, že jsou to rožky křídel. Takto vytvoříme pěkný efekt s minimem námahy.

```
// Druhý trojúhelník
glTexCoord2f(1.0f,1.0f); glVertex3f( 1.0f, 1.0f, 0.0f); // Pravý horní bod
glTexCoord2f(0.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f); // Levý dolní bod
glTexCoord2f(1.0f,0.0f); glVertex3f( 1.0f,-1.0f, 0.0f); // Pravý dolní bod
glEnd(); // Konec kreslení
```

Posuneme motýly směrem dolů odečtením `obj[loop].yi` od `obj[loop].y`. Motýlovo otočení spinz se zvýší o `spinzi` (což může být kladné i záporné číslo) a ohyb křídel se zvýší o `fi`. `fi` může být rovněž kladné, nebo záporné podle směru kam se křídla pohybují.

```
obj[loop].y -= obj[loop].yi; // Pád motýla dolů
obj[loop].spinz += obj[loop].spinzi; // Zvýšení natočení na ose z o spinzi
obj[loop].flap += obj[loop].fi; // Zvětšení máchnutí křídlem o fi
```

Potom co se motýl přesune dolů mimo viditelnou oblast, zavoláme funkci `SetObject(loop)` na tohoto motýla, aby se znovu nastavila náhodná textura, pozice, rychlost,... Jednoduše řečeno: vytvoříme nového motýla v horní části scény, které bude opět padat dolů.

```
if (obj[loop].y < -18.0f) // Je motýl mimo obrazovku?
{
    SetObject(loop); // Nastavíme mu nové parametry
}
```

Aby motýl křídly skutečně mával, musíme zkontrolovat, jestli hodnota mávnutí není větší než 1.0f nebo menší než -1.0f. Pokud ano, změníme směr mávnutí jednoduše nastavením `fi` na opačnou hodnotu (`fi = -fi`). Takže pokud se křídla

pohybují nahoru a dosáhnou 1.0f, fi se změní na záporné číslo a křídla půjdou dolů.

```
    if ((obj[loop].flap > 1.0f) || (obj[loop].flap < -1.0f))// Máme změnit směr
    mávnutí křídly
    {
        obj[loop].fi = -obj[loop].fi;// Změní směr mávnutí
    }
}
```

Sleep(15) bylo přidáno, aby pozastavilo program na 15 milisekund. Na počítačích přátel běžel zběsile rychle a mě se nechtělo nijak upravovat program, takže jsem jednoduše použil tuto funkci. Nicméně osobně její použití ze zásady nedoporučuji, protože se zbytečně plýtvá výpočetním výkonem procesoru.

```
    Sleep(15);// Pozastavení programu na 15 milisekund

    glFlush ();// Vyprázdní renderovací pipeline
}
```

Doufám, že jste si užili tento tutoriál. Snad pro vás udělá nahrávání textur ze zdrojů programu trochu jednodušším na pochopení a texturování trojúhelníků rovněž. Přečetl jsem tento tutoriál snad 5krát a zdá se mi teď už dost jednoduchý.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Václav Slováček - Wessan <horizont (zavináč) host.sk>

Lekce 39 - Úvod do fyzikálních simulací

V gravitačním poli se pokusíme rozpohybovat hmotný bod s konstantní rychlostí, hmotný bod připojený k pružině a hmotný bod, na který působí gravitační síla - vše podle fyzikálních zákonů. Kód je založen na nejnovějším NeHeGL kódu.

Pokud zvládáte fyziku a chcete používat kód pro fyzikální simulaci, tak Vám tento tutoriál může pomoci. Abyste ale mohli něco vytěžit, měli byste vědět něco o počítání s vektory v trojrozměrném prostoru a fyzikálních veličinách, jako je síla nebo rychlost. Tutoriál obsahuje popis velmi jednoduchého fyzikálního simulátoru.

Třída Vector3D

Návrh fyzikálního simulačního enginu není vždy jednoduchý. Ale je zde jednoduchá posloupnost závislostí - aplikace potřebuje simulační část a ta potřebuje matematické knihovny. Tady tuto závislost uplatníme. Naším cílem je získat zásobník na simulaci pohybu objektů v prostoru. Simulační část bude obsahovat třídy Mass a Simulation. Třída Simulation bude naším zásobníkem. Pokud vytvoříme třídu Simulation budeme schopni vyvíjet aplikace, které ji využívají. Ale předtím potřebujeme matematickou knihovnu. Knihovna obsahuje pouze jednu třídu Vector3D, která pro nás bude představovat body, vektory, pozice, rychlost a sílu ve 3D prostoru.

Vector3D tedy bude jediným členem naší matematické knihovny. Obsahuje souřadnice x, y, z v přesnosti float a zavádí operátory pro počítání s vektory ve 3D. Abychom byli konkrétní, přetížíme operátory sčítání, odčítání, násobení a dělení. Protože se tento tutoriál zaměřuje na fyziku a ne matematiku, nebudu podrobně vysvětlovat Vector3D. Podívejte-li se na jeho zdrojový kód, myslím si, že nebudete mít problémy porozumět.

Síla a pohyb

Abychom mohli implementovat fyzikální simulaci, měli bychom vědět, jak bude vypadat náš objekt. Bude mít polohu a rychlost. Pokud je umístěn na Zemi, Měsíci, Marsu nebo na jakémkoliv místě, kde je gravitace musí mít také hmotnost, která se liší podle velikosti působící gravitační síly. Vezměme si třeba knihu. Na Zemi váží 1 kg, ale na Měsíci pouze 0,17 kg, protože Měsíc na ni působí menší gravitační silou. My budeme uvažovat hmotnost na Zemi.

Poté, když jsme pochopili, co pro nás znamená hmotnost, měli bychom se přesunout k síle a pohybu. Objekt s nenulovou rychlostí se pohybuje ve směru rychlosti. Proto je jeden z důvodů změny polohy v prostoru rychlost. Ač se to nezdá, je další působící veličinou čas. Posunutí předmětu tedy závisí na tom, jak rychle se pohybuje, a na tom kolik času uplynulo od počátku pohybu. Pokud vám vztah mezi polohou, rychlostí a časem není jasný, tak asi nemá cenu pokračovat. Doporučuji si vzít učebnici fyziky a najít si kapitolu zabývající se Newtonovy zákony.

Rychlost objektu se mění, pokud na objekt působí nějaká síla. Její vektor je kombinací směru (počáteční a koncový bod) a velikosti. Velikost působení je přímo úměrná působící síle a nepřímo úměrná hmotnosti objektu. Změna rychlosti za jednotku času se nazývá zrychlení. Čím větší síla působí na objekt, tím více zrychluje. Čím má, ale větší hmotnost, tím je menší zrychlení.

`zrychlení = síla / hmotnost`

Odsud jednoduše vyjádříme sílu:

`síla = hmotnost * zrychlení`

Při přípravě prostředí simulace si musíte dávat pozor na to, jaké podmínky v tomto prostředí panují. Prostředí v tomto tutoriálu bude prázdný prostor čekající na zaplnění objekty, které vytvoříme. Nejdříve se rozhodneme, jaké jednotky použijeme pro hmotnost, čas a délku. Rozhodl jsem se použít kilogram pro hmotnost, sekundu pro čas a metr pro délku. Takže jednotky rychlosti budou m/s a jednotky zrychlení budou m/s² (metr za sekundu na druhou).

Abychom toto všechno využili v praxi, musíme napsat třídu, která bude reprezentovat objekt a bude obsahovat jeho hmotnost, polohu, rychlost a sílu, která na něho působí.

```
class Mass
{
public:
    float m; // Hmotnost

    Vector3D pos; // Pozice v prostoru
```

```
Vector3D vel;// Rychlosti a směr pohybu
Vector3D force;// Síla působící na objekt
```

V konstruktoru inicializujeme pouze hmotnost, která se jako jediná nebude měnit. Pozice, rychlost i působící síly se určité měnit budou.

```
Mass(float m)// Konstruktor
{
    this->m = m;
}
```

Aplikujeme silové působení. Objekt může současně ovlivňovat několik zdrojů. Vektor v parametru je součet všech sil působících na objekt. Před jeho aplikací bychom měli stávající sílu vynulovat. K tomu slouží druhá funkce.

```
void applyForce(Vector3D force)
{
    this->force += force;// Vnější síla je přičtena
}

void init()
{
    force.x = 0;
    force.y = 0;
    force.z = 0;
}
```

Zde je stručný seznam toho, co při simulaci musíme provést:

1. Vynulovat sílu - metoda init()
2. Vypočítat znovu působící sílu
3. Přizpůsobit pohyb posunu v čase

Pro práci s časem použijeme Eulerovu metodu, kterou využívá většina her. Existují mnohem sofistikovanější metody, ale tahle postačí. Velmi jednoduše se vypočítá rychlost a poloha pro další časový úsek s ohledem na působící sílu a uplynulý čas. Ke stávající rychlosti přičteme její změnu, která je závislá na zrychlení (síla/m) a uplynulém čase (dt). V dalším kroku přizpůsobíme polohu - opět v závislosti na čase.

```
void simulate(float dt)
{
    vel += (force / m) * dt;// Změna rychlosti je přičtena k aktuální rychlosti
    pos += vel * dt;// Změna polohy je přičtena k aktuální poloze
}

};
```

Jak by měla simulace pracovat

Při fyzikální simulaci se během každého posunu opakuje totéž. Síly jsou vynulovány, potom znovu spočítány. V závislosti na nich se určují rychlosti a polohy předmětů. Tento postup se opakuje tolikrát, kolikrát chceme. Je zajišťován třídou Simulation. Jejím úkolem je vytvářet, ukládat a mazat objekty a starat se o běh simulace.

```
class Simulation
{
public:
    int numOfMasses;// Počet objektů v zásobníku
    Mass** masses;// Objekty jsou uchovávány v jednorozměrném poli ukazatelů na objekty

    Simulation(int numOfMasses, float m)// Konstruktor vytvoří objekty s danou hmotností
    {
        this->numOfMasses = numOfMasses;// Inicializace počtu
        masses = new Mass*[numOfMasses];// Alokace dynamické paměti pro pole ukazatelů

        for (int a = 0; a < numOfMasses; ++a)// Projdeme všechny ukazatele na objekty
            masses[a] = new Mass(m);// Vytvoříme objekt a umístíme ho na místo v poli
    }

    ~Simulation();// Smaže vytvořené objekty
    {
```

```

        release();
    }

    virtual void release()// Uvolní dynamickou paměť
    {
        for (int a = 0; a < numOfMasses; ++a)// Smaže všechny vytvořené objekty
        {
            delete(masses[a]);// Uvolní dynamickou paměť objektů
            masses[a] = NULL;// Nastaví ukazatele na NULL
        }

        delete(masses);// Uvolní dynamickou paměť ukazatelů na objekty
        masses = NULL;// Nastaví ukazatel na NULL
    }

    Mass* getMass(int index)// Získání objektu s určitým indexem
    {
        if (index < 0 || index >= numOfMasses)// Pokud index není v rozsahu pole
            return NULL;// Vrátí NULL

        return masses[index];// Vrátí objekt s daným indexem
    }

```

Proces simulace se skládá ze tří kroků:

1. Init() nastaví síly na nulu
2. Solve() znovu aplikuje síly
3. Simulate(float dt) posune objekty v závislosti na čase

```

    virtual void init()// Tato metoda zavolá init() metodu každého objektu
    {
        for (int a = 0; a < numOfMasses; ++a)// Prochází objekty
            masses[a]->init();// Zavolání init() daného objektu
    }

    virtual void solve()
    {
        // Bez implementace, protože nechceme v základním zásobníku žádné síly
        // Ve vylepšených zásobnicích, bude tato metoda nahrazena, aby na objekty
        // působila nějaká síla
    }

    virtual void simulate(float dt)// Výpočet v závislosti na čase
    {
        for (int a = 0; a < numOfMasses; ++a)// Projdeme všechny objekty
            masses[a]->simulate(dt);// Výpočet nové polohy a rychlosti objektu
    }

```

Všechny tyto metody jsou volány v následující funkci.

```

    virtual void operate(float dt)// Kompletní simulační metoda
    {
        init();// Krok 1: vynulování sil
        solve();// Krok 2: aplikace sil
        simulate(dt);// Krok 3: vypočítání polohy a rychlosti objektů v závislosti na
        // čase
    }
};

```

Nyní máme jednoduchý simulační engine. Je založený na matematické knihovně. Obsahuje třídy Mass a Simulation. Používá běžnou Eulerovu metodu na výpočet simulace. Teď jsme připraveni na vývoj aplikací. Aplikace, kterou budeme vyvíjet využívá:

1. Objekty s konstantní hmotností
2. Objekty v gravitačním poli
3. Objekty spojené pružinou s nějakým bodem

Ovládání simulace aplikací

Předtím než napíšeme nějakou simulaci, měli bychom vědět, jak se třídami zacházet. V tomto tutoriálu jsou simulační a aplikační části odděleny do dvou samostatných souborů. V souboru s aplikační částí je funkce Update(), která se volá opakovaně při každém novém framu.

```
void Update (DWORD milliseconds) // Aktualizace pohybu
{
    // Ošetření vstupu z klávesnice
    if (g_keys->keyDown [VK_ESCAPE] == TRUE)
        TerminateApplication (g_window);

    if (g_keys->keyDown [VK_F1] == TRUE)
        ToggleFullscreen (g_window);

    if (g_keys->keyDown [VK_F2] == TRUE)
        slowMotionRatio = 1.0f;

    if (g_keys->keyDown [VK_F3] == TRUE)
        slowMotionRatio = 10.0f;
}
```

DWORD milliseconds je čas, který uplynul od předchozího volání funkce. Budeme počítat čas při simulacích na milisekundy. Pokud bude simulace sledovat tento čas, půjde stejně rychle jako v reálném čase. K provedení simulace jednoduše zavoláme funkci operate(float dt). Předtím než ji zavoláme musíme znát hodnotu dt. Protože ve třídě Simulation nepoužíváme milisekundy, ale sekundy, převedeme proměnnou milliseconds na sekundy. Potom použijeme proměnnou slowMotionRatio, která udává, jak má být simulace zpomalená vzhledem k reálnému času. Touto proměnnou dělíme dt a dostaneme nové dt. Přidáme dt k proměnné timeElapsed, která udává kolik času simulace už uběhlo (neudává tedy reálný čas).

```
float dt = milliseconds / 1000.0f; // Přepočítá milisekundy na sekundy
dt /= slowMotionRatio; // Dělení dt zpomalovací proměnnou
timeElapsed += dt; // Zvětšení uplynulého času
```

Teď už je dt skoro připraveno na použití v simulaci. Ale! je tu jedna důležitá věc, kterou bychom měli vědět: čím menší je dt, tím reálnější je simulace. Pokud nebude dt dostatečně malé, naše simulace se nebude chovat realisticky, protože pohyb nebude spočítán dostatečně precizně. Analýza stability se užívá při fyzikálních simulacích, aby zajistila maximální přijatelnou hodnotu dt. V tomto tutoriálu se nebudeme pouštět do detailů. Pokud vyvíjíte hru a ne specializovanou aplikaci, tato metoda bohatě stačí na to, abyste se vyhnuli chybám.

Například v automobilovém simulátoru je vhodné, aby se dt pohybovalo mezi 2 až 5 milisekundami pro běžné auto a mezi 1 a 3 milisekundami pro formuli. Při arkádovém simulátoru je možné použít dt v rozsahu od 10 do 200 milisekund. Čím nižší je dt, tím silnější procesor potřebujeme, abychom stíhali simulovat v reálném čase. To je důvod proč se u starších her nepoužívají fyzikální simulace.

V následujícím kódu nastavíme maximální hodnotu dt na 0.1 sekundy (100 milisekund). S touto hodnotou spočítáme kolikrát cyklus simulace při každém projití funkce zopakujeme. To řeší následující vzorec:

```
int numOfIterations = (int)(dt / maxPossible_dt) + 1;
```

NumOfIterations je počet cyklů, které při simulaci provedeme. Dejme tomu, že aplikace běží 20 framů za sekundu. Z toho plyne, že $dt=0.05$. numOfIterations tedy bude 1. Simulace se provede jednou po 0.05 sekundách. Pokud by dt bylo 0.12 sekund, pak numOfIterations bude 2. Pod v kódu uvedeným vzorcem můžete vidět, že dt počítáme ještě jednou. Podělíme ho počtem cyklů a bude $dt = 0.12 / 2 = 0.06$. dt bylo původně vyšší než maximální možná hodnota 0.1. Teď se tedy rovná 0.06. My ale provedeme dva cykly simulace, takže v simulaci uběhne čas 0.12 sekund. Prozkoumejte následující kód a ujistěte se, že všemu rozumíte.

```
// Abychom nepřekročili hranici kdy už se simulace nechová reálně
float maxPossible_dt = 0.1f; // Nastavení maximální hodnoty dt na 0.1 sekund

int numOfIterations = (int)(dt / maxPossible_dt) + 1; // Výpočet počtu opakování
simulace v závislosti na dt a maximální možné hodnotě dt

if (numOfIterations != 0) // Vyhneme se dělení nulou
    dt = dt / numOfIterations; // dt by se měla aktualizovat pomocí numOfIterations

for (int a = 0; a < numOfIterations; ++a) // Simulaci potřebujeme opakovat
numOfIterations-krát
{
    constantVelocity.operate(dt); // Provedení simulace konstantní rychlosti za dt
    sekund
    motionUnderGravitation.operate(dt); // Provedení simulace pohybu v gravitaci za
    dt sekund
    massConnectedWithSpring.operate(dt); // Provedení simulace pružiny za dt sekund
}
```

```
}  
}
```

1. Objekt s konstantní rychlostí

Objekt s konstantní rychlostí nepotřebuje působení externí síly. Pouze vytvoříme objekt a nastavíme jeho rychlost na (1.0f, 0.0f, 0.0f), takže se bude pohybovat po ose x rychlostí 1 m/s. Třidu ConstantVelocity odvodíme od třídy Simulation.

```
class ConstantVelocity : public Simulation  
{  
public:  
    // Konstruktor nejdříve použije konstruktor nadřazené třídy, aby vytvořil objekt o  
    hmotnosti 1 kg  
    ConstantVelocity() : Simulation(1, 1.0f)  
    {  
        masses[0]->pos = Vector3D(0.0f, 0.0f, 0.0f); // Nastavíme polohu objektu na  
        počátek  
        masses[0]->vel = Vector3D(1.0f, 0.0f, 0.0f); // Nastavíme rychlost objektu na  
        (1.0f, 0.0f, 0.0f) m/s  
    }  
};
```

Když je volána metoda operate(float dt) třídy ConstantVelocity, vypočítá se nová polohu objektu. Tato metoda je volána hlavní aplikací před každým překreslením okna. Dejme tomu, že aplikace běží 10 framů za sekundu. To znamená, že při každém novém výpočtu bude dt 0.1 sekundy. Když se potom zavolá funkce simulate(float dt) daného objektu, k jeho pozici se přičte rychlost*dt, které se rovná:

```
Vector3D(1.0f, 0.0f, 0.0f) * 0.1 = Vector3D(0.1f, 0.0f, 0.0f)
```

Při každé frame se objekt pohne o 0.1 metru doprava. Po 10 framech to bude právě 1 metr. Rychlost byla 1.0 m/s. Takže to bude fungovat celkem slušně.

Když spustíte aplikaci, uvidíte objekt pohybující se konstantní rychlostí po ose x. Aplikace nabízí dvě rychlosti plynutí času. Stisknutím F2 poběží stejně rychle jako reálný čas. Stisknutím F3 poběží 10krát pomaleji. Na obrazovce uvidíte přímky znázorňující souřadnicovou plochu. Mezery mezi přímkami jsou 1 metr. Díky těmto přímkám uvidíte, že se objekt pohybuje 1 metr za sekundu v reálném čase a 1 metr za 10 sekund ve zpomaleném čase. Výše popsaná technika je způsob, jak udělat simulaci tak, aby běžela v reálném čase. Abyste ji mohli použít musíte se pevně rozhodnout, v jakých jednotkách simulace poběží.

Aplikace síly

Při simulacích s konstantní rychlostí jsme nepoužili sílu působící na objekt, protože víme, že pokud síla působí na objekt, tak mění jeho rychlost. Pokud chceme pohyb s proměnlivou rychlostí použijeme vnější sílu. Nejdříve musíme všechny působící síly sečíst, abychom dostali výslednou sílu, kterou v simulační fázi aplikujeme na objekt.

Dejme tomu, že chcete použít na objekt sílu 1 N ve směru x. Pak do solve() napíšete:

```
mass->applyForce(Vector3D(1.0f, 0.0f, 0.0f));
```

Pokud chcete navíc přidat sílu 2 N ve směru y, napíšete:

```
mass->applyForce(Vector3D(1.0f, 0.0f, 0.0f));  
mass->applyForce(Vector3D(0.0f, 2.0f, 0.0f));
```

Na objekt můžete použít libovolné množství sil, libovolných směrů, abyste ovlivnili pohyb. V následující části použijeme jednoduchou sílu.

2. Pohyb v gravitaci

MotionUnderGravitation vytvoří objekt a nechá na něj působit sílu. Touto silou bude právě gravitace, která se vypočítá vynásobením hmotnosti objektu a gravitačního zrychlení:

$$F = m * g$$

Gravitační zrychlení na Zemi odpovídá 9.81 m/s^2 . To znamená, že objekt při volném pádu zrychlí každou sekundu o 9.81 m/s dokud na něho nepůsobí žádná jiná síla než gravitace. Může jí být odpor vzduchu, který působí vždycky, ale to sem nepatří.

```
class MotionUnderGravitation : public Simulation
{
public:
    Vector3D gravitation;// Gravitační zrychlení
```

Konstruktor přijímá Vector3D, který udává sílu a orientaci gravitace.

```
// Konstruktor nejdříve použije konstruktor nadřazené třídy, aby vytvořil 1 objekt o
hmotnosti 1kg
MotionUnderGravitation(Vector3D gravitation) : Simulation(1, 1.0f)
{
    this->gravitation = gravitation;// Nastavení gravitace
    masses[0]->pos = Vector3D(-10.0f, 0.0f, 0.0f);// Nastavení polohy objektu
    masses[0]->vel = Vector3D(10.0f, 15.0f, 0.0f);// Nastavení rychlosti objektu
}
virtual void solve();// Aplikace gravitace na všechny objekty, na které má působit
{
    // Použijeme gravitaci na všechny objekty (zatím máme jenom jeden, ale to se
    může do budoucna změnit)
    for (int a = 0; a < numOfMasses; ++a)
        masses[a]->applyForce(gravitation * masses[a]->m);// Síla gravitace se
        spočítá  $F = m * g$ 
}
```

V kódu nahoře si můžete všimnout vzorce $F = m * g$. Pro reálné působení gravitace byste měli předat konstruktoru Vector3D(0.0f, -9.81f, 0.0f). -9.81 znamená, že má gravitace působit proti směru y, což způsobuje, že objekt padá směrem dolů. Můžete zkusit zadat kladné číslo a určitě poznáte rozdíl.

3. Objekt spojený pružinou s bodem

V tomto příkladě chceme spojit objekt se statickým bodem. Pružina by měla objekt přitahovat k bodu upevnění a tak způsobovat oscilaci objektu. V konstruktoru nastavíme bod upevnění a pozici objektu.

```
class MassConnectedWithSpring : public Simulation
{
public:
    float springConstant;// Čím vyšší bude tato konstanta, tím tužší bude pružina
    Vector3D connectionPos;// Bod ke kterému bude objekt připojen

    // Konstruktor nejdříve použije konstruktor nadřazené třídy, aby vytvořil 1 objekt o
    hmotnosti 1kg
    MassConnectedWithSpring(float springConstant) : Simulation(1, 1.0f)
    {
        this->springConstant = springConstant;// Nastavení tuhosti pružiny

        connectionPos = Vector3D(0.0f, -5.0f, 0.0f);// Nastavení pozice upevňovacího
        bodu

        masses[0]->pos = connectionPos + Vector3D(10.0f, 0.0f, 0.0f);// Nastavení pozice
        objektu na 10 metrů napravo od bodu, ke kterému je uchycen
        masses[0]->vel = Vector3D(0.0f, 0.0f, 0.0f);// Nastavení rychlosti objektu na
        nulu
    }
}
```

Rychlost objektu je nula a jeho pozice je 10 metrů napravo od úchyty, takže se bude pohybovat ze začátku směrem doleva. Síla pružiny se dá zapsat jako

$$F = -k * x$$

k je tuhost pružiny a x je vzdálenost od úchyty. Záporná hodnota k značí, že jde o přitažlivou sílu. Kdyby bylo k kladné, tak by pružina objekt odpuzovala, což zcela jistě neodpovídá skutečnému chování.

```
virtual void solve();// Užití síly pružiny
{
    // Použijeme sílu na všechny objekty (zatím máme jenom jeden, ale to se může do
```

```

budoucna změnit)
for (int a = 0; a < numOfMasses; ++a)
{
    Vector3D springVector = masses[a]->pos - connectionPos;// Nalezení vektoru
    od pozice objektu k úchytu
    masses[a]->applyForce(-springVector * springConstant);// Použití síly podle
    uvedeného vzorce
}
};

```

Výpočet síly v kódu nahoře odpovídá vzorci, který jsme si uvedli ($F = -k \cdot x$). Jenom je zde místo x trojrozměrný vektor a místo k je zde `springConstant`. Čím vyšší je `springConstant`, tím rychleji objekt osciluje.

V tomto tutoriálu jsem se snažil předvést základní prvky pro tvorbu fyzikálních simulací. Pokud vás zajímá fyzika, nebude pro vás těžké vytvořit vlastní simulace. Můžete zkoušet složitější interakce a vytvořit tak zajímavá dema a hry. Další v pořadí by měli být simulace pevných objektů, jednoduché mechaniky a pokročilé simulační metody.

napsal: Erkin Tunca <erkintunca (zavináč) icqmail.com>
přeložil: Václav Slováček - Wessan <horizont (zavináč) host.sk>

Lekce 40 - Fyzikální simulace lana

Přichází druhá část dvoudílné série o fyzikálních simulacích. Základy už známe, a proto se pustíme do komplikovanějšího úkolu - klávesnicí ovládat pohyby simulovaného lana. Zatáhneme-li za horní konec, prostřední část se rozhoupe a spodek se vláčí po zemi. Skvělý efekt.

Celé demo založíme na jednoduchém fyzikálním enginu z lekce 39. Nyní už byste měli umět aplikovat libovolné síly na jakýkoli hmotný objekt, přepočítat jeho novou pozici i rychlost a samozřejmě provádět operace s 3D vektory. Pokud něčemu z toho nerozumíte, vraťte se k minulému lekci, popř. zkuste jiné zdroje.

Předpokladem pro fyzikální simulace je implementace fyzikálních podmínek a závislostí, kdy se snažíme o to, aby vše vypadalo jako v reálném prostředí. Nejvíce na očích bývá vždy dynamika - pohybové reakce objektů na uživatelské příkazy. Právě na nich hodnotí, zda se naše práce podařila, či ne. Na úplném začátku se vždy musí najít vhodný kompromis mezi rychlostí a kvalitou. Jsme schopni všimnout si atomů, elektronů nebo protonů? Ne. Určitě bude stačit aproximace pohybu skupiny částic.

Matematika pohybů

Klasická mechanika reprezentuje předměty jako částice v prostoru, které mají určitou hmotnost. Jejich zrychlení závisí na působících silách a pozice na uplynulém čase od minulých výpočtů. Můžeme ji použít k simulaci chování objektů, které jsou viditelné prostým okem (pro mikrosvět platí jiné fyzikální zákony). V lekci 39 jsme s její pomocí implementovali pohyb v gravitaci a objekt zavěšený na pružině. Nyní zkusíme simulovat složitější předmět - lano.

Výkon počítače, který používáme k simulaci

Rychlost počítače je hlavním omezením pro množství detailů simulace. Například při simulaci chodícího člověka eliminujeme na pomalém počítači pohyb prstů na nohou, které jsou sice důležité, ale výsledek vypadá přesvědčivě i bez nich. Musíme je vynechat z jednoduchého důvodu: počítač by nestíhal provádět potřebné výpočty, kterých je i bez nich až příliš.

Jako minimální požadavek pro simulaci určíme počítač s frekvencí procesoru kolem 500 MHz. Z toho plyne omezení počtu detailů. Při implementaci použijeme knihovnu Physics1.h z lekce 39. Tato knihovna obsahuje třídu Mass (hmota), která reprezentuje jeden hmotný bod. Spojením několika za sebe získáme fyzikální model, který reprezentuje lano. Můžeme usoudit, že se bude kývat a různě vlát, ale nebude moci kroužit, protože kroužení nejde pomocí hmotných bodů implementovat (nemohou rotovat okolo os). Určíme si, že body spojíme po vzdálenostech 10 cm. Tato hodnota vychází z toho, že kvůli rychlosti použijeme maximálně 50 až 100 částic na 3 až 4 metrové lano. Z toho plynou 3 až 8 cm velké mezery, což bude ještě větší přesnost, než jsme původně zamýšleli.

Odvození rovnice pohybu

Rovnice pohybu je v matematice vyjádřena diferenciální rovnicí druhého stupně. V modelu lana si můžeme každé dvě sousední částice, ze kterých je složeno, představit jako konce jedné pružiny. Znak o představuje částici a pomlčka pružinu.

$o-----o-----o-----o$

První částice poutá druhou, ta zase třetí, ta čtvrtou atd. Vzniká jakýsi řetězec, složený ze čtyř částic a tří pružin. Pružiny představují zdroje síly mezi každými dvěma částicemi. Zapamatujte si, že síla pružiny se dá vyjádřit takto:

```
síla = -k * x
k: konstanta určující tuhost pružiny
x: vzdálenost mezi body pružiny
```

Kdybychom použili tuto rovnici, lano by se za chvíli smrštilo, protože z rovnice vyplývá, že dokud není vzdálenost částic nulová, působí na ně síla. Všechny by tihly k ostatním a to nechceme. Představte si lano položené na stole. Chceme, aby to naše mělo stejnou pevnost a tudíž musíme explicitně udržovat jeho délku konstantní. Abychom toho dosáhli, musí být při určité kladné vzdálenosti síla pružiny nulová. Nic těžkého:

```
síla = -k * (x - d)
```

k: konstanta určující tuhost pružiny
x: vzdálenost mezi body pružiny
d: konstanta označující kladnou vzdálenost částic, při které pružina zůstane ve stálé poloze

Z rovnice vyplývá, že pokud se bude vzdálenost mezi částicemi rovnat konstantě d, nebudou aplikovány žádné síly. Definovali jsme si lano složené ze sta částic. Zvolíme-li d = 5 cm (0,05 metrů), získáme pevné pětimetrové lano. Pokud bude x větší než d, pružina se začne natahovat a při menším x naopak smršťovat. Každopádně se bude neustále nacházet v blízkosti bodu rovnováhy.

Máme zajištěn celkem slušný pohyb, ale něco mu schází. A to něco jsou ztráty - napětí vláken, jejich tření a podobně. Beze ztrát si fyzikální systém uchovává veškerou energii, kterou mu dodáme - lano se nikdy nepřestane houpat. Než se začneme ztrátám věnovat, pojďme se nejprve podívat na kód.

Třída pružiny

Třída pružiny (anglicky spring) popisuje dvě částice a silové působení pružiny na každou z nich.

```
class Spring// Třída pružiny
{
public:
    Mass* mass1;// Částice na prvním konci pružiny
    Mass* mass2;// Částice na druhém konci pružiny

    float springConstant;// Konstanta tuhosti pružiny
    float springLength;// Délka, při které nepůsobí žádné síly
    float frictionConstant;// Konstanta vnitřního tření
```

V konstruktoru nastavíme vnitřní datové členy na hodnoty, které byly předány v parametrech.

```
Spring(Mass* mass1, Mass* mass2, float springConstant, float springLength, float
frictionConstant)// Konstruktor
{
    // Nastavení členských proměnných
    this->springConstant = springConstant;
    this->springLength = springLength;
    this->frictionConstant = frictionConstant;

    this->mass1 = mass1;
    this->mass2 = mass2;
}
```

Nejdůležitější částí třídy je metoda solve(), ve které se aplikují síly. Za číslo x se má dosadit vzdálenost mezi okrajovými body, v našem případě se jedná o délku 3D vektoru, kterou vypočteme odečtením pozic bodů.

```
void solve();// Aplikování sil na částice
{
    Vector3D springVector = mass1->pos - mass2->pos;// Vektor mezi částicemi

    float r = springVector.length();// Vzdálenost částic
```

Vytvoříme další vektor, který bude označovat výslednou sílu. Konstruktor ji automaticky vynuluje. Protože v další části dělíme, musíme ošetřit, jestli se číslo r nerovná nule. Pokud je vše v pořádku, můžeme přistoupit k výpočtu.

```
Vector3D force;// Pomocný vektor síly

if (r != 0)// Proti dělení nulou
{
```

Chceme-li dosáhnout rovnice uvedené výše, potřebujeme získat jednotkový vektor, který reprezentuje směr působení síly. Defakto ho už máme uložen v objektu springVector, ale je v něm navíc započítána i jeho délka a to nechceme. Dá se však velice jednoduše odstranit dělením (springVector / r). Dále se pokusíme implementovat část (x - d). Máme jak vzdálenost bodů, tak délku pružiny, nic nám proto nebrání, abychom je odečetli (r - springLength). Konečný výsledek ještě vynásobíme tuhostí pružiny. Záporná hodnota označuje, že se lano bude spíše vléci než odrážet.

```
force += (springVector / r) * (r - springLength) * (-springConstant);//
Výpočet síly
}
```

Vyřešili jsme silové působení pružiny, ale ještě nám chybí ztráty energie v materiálu. Pokud se na hmotu aplikuje síla v opačném směru, než se pohybuje, zpomalí. Kam se ztratila pohybová energie? Mohla se například přeměnit ve tření a

následně v tepelnou energii.

```
třecí síla = -k * rychlost
k: konstanta představující velikost ztrát (např. v závislosti na drsnosti povrchu)
rychlost: rychlost hmoty, na kterou působí třecí síla
```

Tato rovnice by se dala napsat i jinak (více složitě), ale nám bude tato verze bohatě stačit. Všimněte si, že lze dosadit pouze rychlost jednoho bodu, ale pružina se skládá ze dvou. Co dělat? Vypočteme rozdíl rychlostí a předáme ho jako relativní rychlost. Ztráty tedy budou představovat vnitřní tření v materiálu.

```
force += -(mass1->vel - mass2->vel) * frictionConstant;// Zmenšení síly o tření
```

Podle Newtonova zákona akce a reakce aplikujeme na jeden bod pružiny kladnou sílu a na druhý zápornou. (Působí-li jedno těleso na druhé silou, působí i druhé na první. Obě síly jsou stejně velké, ale opačně orientované.) Představte si dvě lodky na jezeře. Po odstrčení se nezačne pohybovat jenom jedna, ale obě. Pokud by bylo jedno těleso mnohonásobně těžší než to druhé, mohlo by se silové působení na něj zanedbat, jeho zrychlení by se blížilo k nule. Představte si například raketu v gravitačním poli planety, která je přitahována dolů do jejího středu. Raketa se zároveň snaží přitáhnout planetu nahoru, nicméně nemá nejmenší šanci :-)) a tak se druhé působení jednoduše zanedbává. To ale není náš případ, protože oba naše objekty mají stejnou hmotnost.

```
mass1->applyForce(force);// Aplikování síly na částici 1
mass2->applyForce(-force);// Aplikování opačné síly na částici 2
}
};
```

Nyní už máme vyřešenu rovnici pohybu, kterou si můžeme představit jako silové působení pružin. Abychom simulaci dokončili, přidáme ještě gravitační sílu, tření lana se vzduchem a plochý povrch, po kterém lanem posunujeme. První dvě jsou jednoduché, nejprve odpor vzduchu:

```
odpor vzduchu = -k * rychlost
k: konstanta, která představuje velikost odporu
rychlost: rychlost pohybu
```

A gravitace...

```
gravitační síla = gravitační zrychlení * hmotnost
```

Gravitace i odpor vzduchu působí na každou částici lana zvlášť. Co se zemí? Můžeme si vytvořit pomyslnou rovinu a testovat, kolize s částicemi lana. Pokud se ocitne pod úrovní roviny, vyvýšíme ji a rozšíříme působící sílu o tření s podlahou.

Nastavení počátečních hodnot simulace

V tuto chvíli je prostředí připraveno pro simulaci, ale potřebujeme definovat jednotky používaných fyzikálních veličin. Vzdálenost bude specifikována v metrech, čas v sekundách a hmotnost v kilogramech. Gravitaci nastavíme tak, aby působila ve směru záporné části osy y se zrychlením $9,81 \text{ m}\cdot\text{s}^{-2}$ (odpovídá gravitaci na zemi). Před spuštěním umístíme lano rovnoběžně se zemí ve vzdálenosti čtyř metrů od ní. Aby se jí mohlo dotknout bude mít délku (v klidovém stavu) také čtyři metry, Z toho vyplývá 5 cm vzdálenost mezi jednotlivými částicemi ($4 \text{ m} / 80 \text{ částic} = 0,05 \text{ m} = 5 \text{ cm}$). Normální délku pružiny mezi částicemi (délka bez působení žádných sil) tedy nastavíme na těchto 5 cm, aby lano na začátku simulace nebylo ani napnuté ani prohnuté. Celkovou hmotnost lana určíme na 4 kg a to dává 0,05 kg (= 50 gramů) na každou částici. Pro přehlednost si vše shrneme:

- gravitační zrychlení: $9,81 \text{ m}\cdot\text{s}^{-2}$
- počet částic lana: 80
- normální vzdálenost mezi sousedními částicemi (bez působení sil): 5 cm
- hmotnost jedné částice: 50 gramů
- počáteční orientace lana: horizontálně bez napětí

Nyní zkusíme vypočítat konstantu určující tuhost pružiny. Pokud budeme držet lano za jeden konec, tak se natáhne. Představte si elastické lano se závažím. Je to úplně stejné, akorát my nemáme pouze jednu pružinu ale hned několik. Nejvíce se natáhne horní pružina, protože drží hmotnost všech ostatních částic - prakticky celé lano. Naopak nejméně se prodlouží ta spodní, protože je na ní zavěšena jenom jedna jediná částice. Nechceme, aby se ta horní natáhla více než o 1 cm.

```
f = hmotnost lana * gravitační zrychlení = (4 * 9,81) N ~= 40 N
```

Síla pružiny odpovídá přibližně 40 N. Dá se ale vyjádřit i jinak:

```
síla pružiny = -k * x = -k * 0,1 metrů
```

Suma těchto sil by se měla rovnat nule.

```
40 N + (-k * 0,01 metrů) = 0
```

Vypočteme a získáme

```
k = 4000 N/m
```

Pro snadnější zapamatovatelnost budeme předpokládat 10000 N/m, což nám dává větší tuhost lana, které se v horní části natáhne jen o 4 mm.

Aby se našla konstanta vnitřního tření v laně, museli bychom podniknout mnohem více komplikovanější výpočet než je ten výše. Proto jsem použil metodu pokusů a omylů a získal...

```
konstanta vnitřního tření = 0,2 N/(m/s)
```

... což vypadá celkem realisticky.

Třída simulace lana

Předtím než začneme zkoumat tření se vzduchem a se zemí, pojďme se podívat na třídu simulace lana, která je odvozená od obecné třídy simulace z lekce 39. Tato třída obsahuje čtyři metody potřebné pro běh simulace.

- virtual void init() - reset sil
- virtual void solve() - aplikování sil
- virtual void simulate() - iterování pozic a rychlostí
- virtual void operate(float dt) - kompletní simulační metoda

V potomku základní třídy přepíšeme funkci solve() a funkci simulate(float dt), protože kvůli lanu potřebujeme jejich speciální implementaci. Solve() slouží k aplikování sil a simulate(float dt) k ovládní lana za jeden konec pověšený v prostoru. Jak už bylo řečeno, třída RopeSimulation je potomkem třídy Simulation. Obecnou simulaci rozšiřuje o lano složené z hmotných bodů (částic) pospojovaných pružinami. Tyto pružiny mají vnitřní tření a klidovou délku. Jeden konec lana je udržován v prostoru na souřadnicích ropeConnectionPos a je jím možno pohybovat pomocí metody setRopeConnectionVel(Vector3D ropeConnectionVel). Třída dále zajišťuje tření se vzduchem a s rovinným povrchem (nebo-li se zemí), jehož normála z něj vychází ve směru kladné části osy y.

```
class RopeSimulation : public Simulation// Třída simulace lana
{
public:
    Spring** springs;// Pružiny spojující částice

    Vector3D gravitation;// Gravitační zrychlení
    Vector3D ropeConnectionPos;// Bod v prostoru; pozice první částice pro ovládní
    lanem
    Vector3D ropeConnectionVel;// Rychlost a směr požadovaného pohybu

    float groundRepulsionConstant;// Velikost odražení částic od země
    float groundFrictionConstant;// Velikost tření částic se zemí
    float groundAbsorptionConstant;// Velikost absorpce sil částic zemí (vertikální
    kolize)
    float groundHeight;// Pozice roviny země na ose y
    float airFrictionConstant;// Konstanta odporu vzduchu na částice
```

Konstruktořem inicializujeme všechny členské proměnné třídy, alokujeme paměť pro všechny potřebné pružiny a umístíme je v řadě rovnoběžně se zemí.

```
RopeSimulation{// Konstruktor třídy
    int numOfMasses,// Počet částic
    float m,// Hmotnost každé částice
    float springConstant,// Tuhost pružiny
    float springLength,// Délka pružiny v klidovém stavu
    float springFrictionConstant,// Konstanta vnitřního tření pružiny
    Vector3D gravitation,// Gravitační zrychlení
    float airFrictionConstant,// Odpor vzduchu
    float groundRepulsionConstant,// Odražení částic zemí
    float groundFrictionConstant,// Tření částic se zemí
```



```

float groundAbsorptionConstant, // Absorpce sil zemí
float groundHeight // Pozice země na ose y
) : Simulation(numOfMasses, m) // Inicializace předka třídy
{
    this->gravitation = gravitation;
    this->airFrictionConstant = airFrictionConstant;

    this->groundFrictionConstant = groundFrictionConstant;
    this->groundRepulsionConstant = groundRepulsionConstant;
    this->groundAbsorptionConstant = groundAbsorptionConstant;
    this->groundHeight = groundHeight;

    for (int a = 0; a < numOfMasses; ++a) // Nastavení počáteční pozice částic
    {
        masses[a]->pos.x = a * springLength; // Offsety jednotlivých částic
        masses[a]->pos.y = 0; // Rovnoběžně se zemí
        masses[a]->pos.z = 0; // Rovnoběžně s obrazovkou
    }

    springs = new Spring*[numOfMasses - 1]; // Alokace paměti pro ukazatele na pružiny

    for (a = 0; a < numOfMasses - 1; ++a) // Vytvoření jednotlivých pružin
    {
        springs[a] = new Spring(masses[a], masses[a + 1], springConstant,
            springLength, springFrictionConstant); // Dvě částice na pružinu
    }
}

```

Jak jsme si už řekli výše, funkce solve() slouží k aplikování sil na všechny objekty, ze kterých je lano složeno.

```

void solve() // Aplikování sil
{

```

Nejdříve ošetříme všechny pružiny; na jejich pořadí nezáleží. Třída obsahuje svoji vlastní funkci.

```

    for (int a = 0; a < numOfMasses - 1; ++a) // Prochází pružiny
    {
        springs[a]->solve(); // Aplikování sil na pružinu
    }

```

V cyklu přes všechny částice zajistíme působení gravitace a odpor vzduchu.

```

    for (a = 0; a < numOfMasses; ++a) // Prochází částice
    {
        masses[a]->applyForce(gravitation * masses[a]->m); // Gravitace
        masses[a]->applyForce(-masses[a]->vel * airFrictionConstant); // Odpor
        vzduchu
    }

```

Síly ze země vypadají trochu komplikovaněji, ale jsou právě tak jednoduché, jako všechny ostatní. Země na částici může působit pouze tehdy, pokud se navzájem dotknou, tj. částice se na ose y nachází pod její úrovní.

```

        if (masses[a]->pos.y < groundHeight) // Kolize se zemí,
        {

```

Smýkání lana na zemi je zajištěno třecí silou, která ze své podstaty zanedbává rychlost na ose y. Y je směr, kterým země (její normálový vektor) směřuje vzhůru; smýkání nemůže působit v tomto směru.

```

            Vector3D v; // Pomocný vektor

            v = masses[a]->vel; // Grabování rychlosti
            v.y = 0; // Vynechání rychlosti na ose y

            masses[a]->applyForce(-v * groundFrictionConstant); // Třecí síla země
        }
    }

```

Opakem smýkání je absorpční efekt, kdy se síla aplikuje pouze ve směru, kterým země směřuje vzhůru. Proto obě ostatní složky vynulujeme. Absorpce nemůže na částici působit tehdy, když se vzdaluje od země. Pokud bychom nepřidali podmínku $v.y < 0$, lano by tíhlo k zemi, i když by se jí už nedotýkalo.

```

            v = masses[a]->vel; // Grabování rychlosti

            v.x = 0; // Zanedbání rychlosti na osách x a z
            v.z = 0;
        }
    }

```

```

    if (v.y < 0) // Pouze při kolizi směrem k zemi
    {
        masses[a]->applyForce(-v * groundAbsorptionConstant); // Absorpční síla
    }

```

Síla odrazu je poslední ze sil, kterou vyvolává kolize se zemí. Země odráží částice právě tak, jako kdyby se mezi nimi nacházela pružina. Její síla je přímo úměrná rychlosti částice při nárazu.

```

        Vector3D force = Vector3D(0, groundRepulsionConstant, 0) * (groundHeight
        - masses[a]->pos.y); // Síla odrazu
        masses[a]->applyForce(force); // Aplikování síly odrazu
    }
}

```

Abychom vyvolali dojem média, které za jeden konec drží lano a pohybuje s ním, musíme přepsat metodu simulate(float dt).

```

void simulate(float dt) // Simulace lana
{

```

Nejdříve ze všeho zavoláme metodu předka, potom k aktuální pozici přičteme rychlost pohybu a nakonec ošetříme náraz do země.

```

    Simulation::simulate(dt); // Metoda předka
    ropeConnectionPos += ropeConnectionVel * dt; // Zvětšení pozice o rychlost
    if (ropeConnectionPos.y < groundHeight) // Dostala se částice pod zem?
    {
        ropeConnectionPos.y = groundHeight; // Přesunutí na úroveň země
        ropeConnectionVel.y = 0; // Nulování rychlosti na ose y
    }

```

Pomocí právě získaných parametrů nastavíme vlastnosti částice na indexu nula.

```

    masses[0]->pos = ropeConnectionPos; // Pozice první částice
    masses[0]->vel = ropeConnectionVel; // Rychlost první částice
}

```

Potřebujeme funkci, pomocí které budeme moci nastavit rychlost první částice.

```

void setRopeConnectionVel(Vector3D ropeConnectionVel) // Nastavení rychlosti první
částice
{
    this->ropeConnectionVel = ropeConnectionVel; // Přiřazení rychlostí
}

```

Tím končí vysvětlování vnitřních závislostí třídy RopeSimulation. Její objekt v aplikaci vytváříme dynamicky pomocí operátoru new. Změnou hodnot předávaných konstruktoru můžete docílit téměř libovolného chování lana. Všimněte si, že zemi umísťujeme -1.5f jednotek pod osou y. Lano inicializujeme na nule rovnoběžně se zemí. To nám tedy dává možnost vidět hned na začátku efektní pád a kolizi se zemí.

```

RopeSimulation* ropeSimulation = new RopeSimulation( // Vytvoření objektu simulace lana
    80, // 80 částic
    0.05f, // Každá částice váží 50 gramů
    10000.0f, // Tuhost pružin
    0.05f, // Délka pružin, při nepůsobení žádné síly
    0.2f, // Konstanta vnitřního tření pružiny
    Vector3D(0, -9.81f, 0), // Gravitační zrychlení
    0.02f, // Odpor vzduchu
    100.0f, // Síla odrazu od země
    0.2f, // Třecí síla země
    2.0f, // Absorpční síla země
    -1.5f); // Poloha země na ose y

```

Stejně jako v lekcí 39 existuje maximální možná hodnota dt simulace. S výše uvedenými parametry konstruktoru činí přibližně 0,002 sekund. Pokud vaše změna tuto hodnotu sníží, může simulace vypadat poněkud nestabilně a lano nemusí pracovat správně. Abyste poměry stabilizovali, musíte najít nové maximální možné dt. Velké síly a/nebo malé hmotnosti znamenají větší nestabilitu, protože zrychlení bude vyšší (zrychlení = síla / hmotnost).

```

// Funkce Update (DWORD milliseconds)

float dt = milliseconds / 1000.0f; // Převod milisekund na sekundy
float maxPossible_dt = 0.002f; // Maximální možné dt
int numOfIterations = (int)(dt / maxPossible_dt) + 1; // Výpočet počtu opakování při
této aktualizaci

if (numOfIterations != 0) // Proti dělení nulou
{
    dt = dt / numOfIterations; // Aktualizace dt podle numOfIterations
}

for (int a = 0; a < numOfIterations; ++a) // Opakování simulace
{
    ropeSimulation->operate(dt);
}
}

```

Lanem můžete pohybovat pomocí šipek a kláves HOME a END. Pohrajte si, stojí to za to. Simulační procedura hodně závisí na výkonu procesoru, tudíž jsou také důležité optimalizace kompilátoru. Při standardním Visual C++ Release nastavení běží program více než 10 krát rychleji než v Debug módu, pro který činí minimální frekvence procesoru cca. 500 MHz.

V tomto tutoriálu je představeno kompletní fyzikální nastavení, teoretická funkce, design a implementace. Více pokročilejší simulace uvnitř vypadají úplně stejně jako tato.

napsal: Erkin Tunca <erkintunca (zavináč) icqmail.com>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 41 - Volumetrická mlha a nahrávání obrázků pomocí IPicture

V tomto tutoriálu se naučíte, jak pomocí rozšíření EXT_fog_coord vytvořit volumetrickou mlhu. Také zjistíte, jak pracuje IPicture kód a jak ho můžete využít pro nahrávání obrázků ve svých vlastních projektech. Demo sice není až tak komplexní jako některá jiná, nicméně i přesto vypadá hodně efektně.

Pokud demo nebude na vašem systému fungovat, nejdříve se ujistěte, že máte nainstalované nejnovější ovladače grafické karty. Pokud to nepomohlo, zvažujte o koupi nové (Překl.: :-]). V současné době už ne zrovna nejnovější GeForce 2 pracuje dobře a ani nestojí tak moc. Pokud vaše grafická karta nepodporuje rozšíření mlhy, kdo může vědět, jaká další rozšíření nebude podporovat?

Pro ty z vás, kterým toto demo nejede a cítí se vyloučení... mějte na paměti následující: Snad každý den dostávám nejméně jeden email s dotazem na nový tutoriál. Nejhorší z toho je, že většina z nich už je online. Lidé se neobtěžují číst to, co už je napsáno a přeskakují na témata, která je více zajímavá. Některé tutoriály jsou příliš komplexní, a proto z mé strany vyžadují někdy i týdny programování. Pak jsou tady tutoriály, které bych sice mohl napsat, ale většinou se jim vyhýbám, protože nefungují na všech kartách. Nyní jsou už karty jako GeForce levné natolik, aby si je mohl dovolit téměř každý, takže už nebudu dále ospravedlňovat nepsání takovýchto tutoriálů. Popravdě, pokud vaše karta podporuje pouze základní rozšíření, budete s největší pravděpodobností chybět! Pokud se vrátím k přeskakování témat jako jsou např. rozšíření, tutoriály se brzy oproti ostatním začnou výrazně opožďovat.

Kód začíná velmi podobně jako starý základní kód a povětšinou je identický s novým NeHeGL kódem. Jediný rozdíl spočívá v inklování OLECTL hlavičkového souboru, který, chcete-li používat IPicture pro loading obrázků, musí být přítomen.

Překl.: IPicture je podle mě sice hezký nápad a pracuje perfektně, nicméně je kompletně vystavěn na ABSOLUTNĚ NEPŘENOSITELNÝCH technologiích MS, které jdou tradičně používat výhradně pod nejmenovaným OS, všichni víme, o který jde.

```
#include <windows.h> // Windows
#include <gl\gl.h> // OpenGL
#include <gl\glu.h> // GLU
#include <olectl.h> // Knihovna OLE Controls Library (použita při nahrávání obrázků)
#include <math.h> // Matematika

#include "NeHeGL.h" // NeHeGL

#pragma comment(lib, "opengl32.lib") // Přilinkování OpenGL a GLU
#pragma comment(lib, "glu32.lib")

#ifdef CDS_FULLSCREEN // Některé kompilátory CDS_FULLSCREEN nedefinují
#define CDS_FULLSCREEN 4
#endif

GL_Window* g_window; // Struktura okna
Keys* g_keys; // Klávesnice
```

Deklarujeme čtyř prvkové pole fogColor, které bude ukládat barvu mlhy, v našem případě se jedná o tmavě oranžovou (trocha červené smíchaná se špetkou zelené). Desetinná hodnota camz bude sloužit pro umístění kamery na ose z. Před vykreslením vždy provedeme translaci.

```
GLfloat fogColor[4] = {0.6f, 0.3f, 0.0f, 1.0f}; // Barva mlhy
GLfloat camz; // Pozice kamery na ose z
```

Ze souboru glexth převezmeme symbolické konstanty GL_FOG_COORDINATE_SOURCE_EXT a GL_FOG_COORDINATE_EXT. Pokud chcete kód zkompileovat, musí být nastaveny.

```
// Převzato z glexth
#define GL_FOG_COORDINATE_SOURCE_EXT 0x8450 // Symbolické konstanty potřebné pro
rozšíření FogCoordfEXT
#define GL_FOG_COORDINATE_EXT 0x8451
```

Abychom mohli používat funkci glFogCoordfExt(), která bude vstupním bodem pro rozšíření, potřebujeme deklarovat její prototyp. Nejdříve pomocí typedef vytvoříme nový datový typ, ve kterém bude specifikován počet a typ parametrů (jedno desetinné číslo). Vytvoříme globální proměnnou tohoto typu - ukazatel na funkci a prozatím ho nastavíme na NULL.

Jakmile mu přiřadíme pomocí `wglGetProcAddress()` adresu OpenGL ovladače rozšíření, budeme moci zavolat `glFogCoordfEXT()`, jako kdyby to byla normální funkce.

Takže co už máme... Víme, že `PFNGLFOGCOORDFEXTPROC` přebírá jednu desetinnou hodnotu (`GLfloat coord`). Protože je proměnná `glFogCoordfEXT` stejného typu můžeme říct, že také potřebuje jednu desetinnou hodnotu... tedy `glFogCoordfEXT(GLfloat coord)`. Funkci máme definovanou, ale zatím nic nedělá, protože `glFogCoordfEXT` se v tuto chvíli rovná `NULL`. Dále v kódu jí přiřadíme adresu OpenGL ovladače pro rozšíření.

Doufám, že to všechno dává smysl. Pokud jednou víte, jak tento kód pracuje, je velmi jednoduchý, ale jeho popsání je, alespoň pro mě, extrémně složité.

```
typedef void (APIENTRY * PFNGLFOGCOORDFEXTPROC) (GLfloat coord); // Funkční prototyp
PFNGLFOGCOORDFEXTPROC glFogCoordfEXT = NULL; // Ukazatel na funkci glFogCoordfEXT()
```

```
GLuint texture[1]; // Jedna textura
```

Pojďme se podívat na převod obrázků do textury pomocí magické `IPicture`. Funkci se předává řetězec se jménem obrázku a ID textury. Za jméno se může dosadit buď disková cesta nebo webové URL.

Pro pomocnou bitmapu budeme potřebovat kontext zařízení (`hdcTemp`) a místo, kam by se dala uložit (`hbmpTemp`). Ukazatel `pPicture` představuje rozhraní k `IPicture`. `WszPath` a `szPath` slouží k uložení absolutní cesty k souboru nebo URL. Dále potřebujeme dvě proměnné pro šířku a dvě proměnné pro výšku. `LWidth` a `LHeight` ukládají aktuální rozměry obrázku, `IWidthpixels` a `IHeightpixels` obsahují šířku a výšku v pixelech upravenou podle maximální velikosti textury, která může být uložena do grafické karty. Hodnotu maximální velikosti uložíme do `glMaxTexdim`.

```
int BuildTexture(char *szPathName, GLuint &texid) // Nahraje obrázek a konvertuje ho na
texturu
{
    HDC hdcTemp; // Pomocný kontext zařízení
    HBITMAP hbmpTemp; // Pomocná bitmapa
    IPicture *pPicture; // Rozhraní pro IPicture
    OLECHAR wszPath[MAX_PATH+1]; // Absolutní cesta k obrázku (unicode)
    char szPath[MAX_PATH+1]; // Absolutní cesta k obrázku (ascii)
    long lWidth; // Šířka v logických jednotkách
    long lHeight; // Výška v logických jednotkách
    long lWidthPixels; // Šířka v pixelech
    long lHeightPixels; // Výška v pixelech
    GLint glMaxTexDim; // Maximální rozměr textury
}
```

V další části kódu zjistíme, zda je jméno obrázku diskovou cestou nebo URL. Jedná-li se o URL, zkopírujeme jméno do proměnné `szPath`. V opačném případě získáme pracovní adresář a spojíme ho se jménem. Děláme to, protože potřebujeme plnou cestu k souboru. Pokud máme např. demo uložené v adresáři `C:\WOWLESSON41` a pokoušíme se nahrát obrázek `DATA\WALL.BMP`. Uvedená konstrukce přidá doprostřed ještě zpětné lomítko a tak vznikne `C:\WOWLESSON41\DATA\WALL.BMP`.

```
if (strstr(szPathName, "http://")) // Obsahuje cesta řetězec "http://"
{
    strcpy(szPath, szPathName); // Zkopírování do szPath
}
else // Nahrávání ze souboru
{
    GetCurrentDirectory(MAX_PATH, szPath); // Pracovní adresář
    strcat(szPath, "\\"); // Přidá zpětné lomítko
    strcat(szPath, szPathName); // Přidá cestu k souboru
}
```

Aby funkce `OleLoadPicturePath()` rozuměla cestě k souboru, musíme ji převést z ASCII do kódování UNICODE (dvoubytové znaky). Pomůže nám s tím `MultiByteToWideChar()`. První parametr, `CP_ACP`, znamená Ansi Codepage, druhý specifikuje zacházení s nenamapovanými znaky (ignorujeme ho). `SzPath` je samozřejmě převáděný řetězec a čtvrtý parametr představuje šířku řetězce s Unicode znaky. Pokud za něj předáme `-1`, předpokládá se, že bude ukončen pomocí `NULL`. Do `wszPath` se uloží výsledek, `MAX_PATH` je maximální velikostí cesty k souboru (256 znaků).

Po konverzi cesty do kódování Unicode se pokusíme pomocí `OleLoadPicturePath` nahrát obrázek. Při úspěchu bude `pPicture` obsahovat ukazatel na data obrázku, návratový kód se uloží do `hr`.

```
MultiByteToWideChar(CP_ACP, 0, szPath, -1, wszPath, MAX_PATH); // Konverze ascii
kódování na Unicode
HRESULT hr = OleLoadPicturePath(wszPath, 0, 0, 0, IID_IPicture, (void**)
&pPicture); // Loading obrázku

if (FAILED(hr)) // Neúspěch
{
```

```

    return FALSE;// Konec
}

```

Pokusíme se vytvořit kompatibilní kontext zařízení. Pokud se to nepovede uvolníme data obrázku a ukončíme program.

```

hdcTemp = CreateCompatibleDC(GetDC(0));// Pomocný kontext zařízení

if(!hdcTemp)// Neúspěch
{
    pPicture->Release();// Uvolní IPicture
    return FALSE;// Konec
}

```

Přišel čas na položení dotazu grafické kartě, jakou podporuje maximální velikost textury. Tato část kódu je důležitá, protože díky ní bude obrázek vypadat dobře na všech grafických kartách. Nejen, že umožní upravit velikost na mocninou dvou, ale také ho přizpůsobí podle velikosti paměti grafické karty. Zkrátka: budeme moci nahrávat obrázky s libovolnou šířkou a výškou. Jediná nevýhoda pro majitele málo výkonných grafických karet spočívá v tom, že se při zobrazení obrázků s vysokým rozlišením ztratí spousta detailů.

Funkce glGetIntegerv() vrátí maximální rozměry textur (256, 512, 1024, atd.), potom zjistíme aktuální velikost našeho obrázku a převedeme ji na pixely. Matematiku zde nebudu vysvětlovat.

```

glGetIntegerv(GL_MAX_TEXTURE_SIZE, &glMaxTexDim);// Maximální podporovaná velikost
textury

pPicture->get_Width(&lWidth);// Šířka obrázku a konvertování na pixely
lWidthPixels = MulDiv(lWidth, GetDeviceCaps(hdcTemp, LOGPIXELSX), 2540);

pPicture->get_Height(&lHeight);// Výška obrázku a konvertování na pixely
lHeightPixels = MulDiv(lHeight, GetDeviceCaps(hdcTemp, LOGPIXELSY), 2540);

```

Pokud je velikost obrázku menší než maximální podporovaná, změníme velikost na mocninu dvou, která ale bude založená na aktuální velikosti. Přičteme 0.5f, takže se bude vždy zvětšovat na následující velikost. Například rovná-li se šířka 400 pixelům a karta podporuje maximálně 512, bude lepší zvolit 512 než 256, protože by se zbytečně zahodily detaily. Naopak při větší velikosti než maximální musíme zmenšovat na podporovanou velikost. Totéž platí i pro výšku.

Překl.: Opravte mě, jestli se mýlím. Co se stane když např. vezmu obrázek, který má šířku 80 a výšku 300 pixelů? Té matematice sice moc nerozumím :-), ale z toho, co je zde uvedeno, logicky vychází, že vznikne obdélníkový (ne čtvercový!) obrázek o rozměrech 128x512 pixelů. Možná by bylo vhodné ještě přidat něco ve stylu: pokud je jeden rozměr menší než druhý, uprav hodnoty na čtverec.

```

if (lWidthPixels <= glMaxTexDim)// Je šířka menší nebo stejná než maximálně
podporovaná
{
    // Změna velikosti na nejbližší mocninu dvou
    lWidthPixels = 1 << (int)floor((log((double)lWidthPixels)/log(2.0f)) + 0.5f);
}
else// Bude se zmenšovat na maximální velikost
{
    lWidthPixels = glMaxTexDim;
}

if (lHeightPixels <= glMaxTexDim)// Je výška menší nebo stejná než maximálně
podporovaná
{
    // Změna velikosti na nejbližší mocninu dvou
    lHeightPixels = 1 << (int)floor((log((double)lHeightPixels)/log(2.0f)) + 0.5f);
}
else// Bude se zmenšovat na maximální velikost
{
    lHeightPixels = glMaxTexDim;
}

```

V tuto chvíli máme data nahraná a také známe požadovanou velikost obrázku, abychom ho mohli dále upravovat, musíme vytvořit pomocnou bitmapu. Bi bude obsahovat informace o hlavičce a pBits bude ukazovat na data obrázku. Požadujeme barevnou hloubku 32 bitů na pixel, správnou šířku i výšku v kódování RGB s jednou bitplane.

```

// Pomocná bitmapa
BITMAPINFO bi = {0};// Typ bitmapy
DWORD *pBits = 0;// Ukazatel na data bitmapy

bi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);// Velikost struktury

```

```

bi.bmiHeader.biBitCount = 32;// 32 bitů
bi.bmiHeader.biWidth = lWidthPixels;// Šířka
bi.bmiHeader.biHeight = lHeightPixels;// Výška
bi.bmiHeader.biCompression = BI_RGB;// RGB formát
bi.bmiHeader.biPlanes = 1;// 1 Bitplane

```

Převzato z MSDN: Funkce CreateDIBSection() vytváří DIB, do kterého může aplikace přímo zapisovat. Vrací ukazatel na umístění bitů bitmapy, můžeme také nechat systém alokovat paměť.

HdcTemp ukládá pomocný kontext zařízení, bi je hlavička bitmapy. DIB_RGB_COLORS říká programu, že chceme uložit RGB data, která nebudou indexována do logické palety (každý pixel bude mít červenou, zelenou a modrou složku). Ukazatel pBits bude obsahovat adresu výsledných dat a poslední dva parametry budeme ignorovat. Pokud nenastane žádná chyba, pomocí SelectObject() připojíme bitmapu k pomocnému kontextu zařízení.

```

// Touto cestou je možné specifikovat barevnou hloubku a získat přístup k datům
hbmpTemp = CreateDIBSection(hdcTemp, &bi, DIB_RGB_COLORS, (void*)&pBits, 0, 0);

if(!hbmpTemp)// Neúspěch
{
    DeleteDC(hdcTemp);// Uvolnění kontextu zařízení
    pPicture->Release();// Uvolní IPicture
    return FALSE;// Konec
}

SelectObject(hdcTemp, hbmpTemp);// Zvolí bitmapu do kontextu zařízení

```

Nastal čas pro vyplnění pomocné bitmapy daty obrázku. Funkce pPicture->Render() to udělá za nás a navíc upraví obrázek na libovolnou velikost, kterou potřebujeme. HdcTemp představuje pomocný kontext zařízení a další dva následující parametry specifikují vertikální a horizontální offset (počet prázdných pixelů zleva a shora). My chceme, aby byla celá bitmapa kompletně vyplněna, takže zadáme dvě nuly. Další dva parametry určují požadovanou velikost výsledného obrázku (na kolik pixelů se má roztáhnout popř. zmenšit). Nula na dalším místě je horizontální offset ve zdrojových datech, od kterého chceme začít číst, z čehož plyne, že půjdeme zleva doprava. lHeight určuje vertikální offset, data chceme číst od zdola nahoru. Zadáním lHeight se přesuneme na samé dno zdrojového obrázku. lWidth je množstvím pixelů, které se budou kopírovat ze zdrojového obrázku, v našem případě se jedná o všechna horizontální data. Předposlední parametr, trochu odlišný, má zápornou hodnotu, záporné lHeight, abychom byli přesní. Ve výsledku to znamená, že chceme zkopírovat všechna vertikální data, ale od zdola nahoru. Touto cestou bude při kopírování do cílové bitmapy převrácen. Poslední parametr nepoužijeme.

```

// Vykreslení IPicture do bitmapy
pPicture->Render(hdcTemp, 0, 0, lWidthPixels, lHeightPixels, 0, lHeight, lWidth, -
lHeight, 0);

```

Nyní máme k dispozici novou bitmapu se správnými rozměry, ale bohužel je uložena ve formátu BGR. (Překl.: Proč tomu tak je, bylo vysvětlováno v 35. tutoriálu na přehrávání AVI videa.) Pomocí jednoduchého cyklu tyto dvě složky prohodíme a zároveň nastavíme alfu na 255. Dá se říci, že jakákoli jiná hodnota stejně nebude mít nejmenší efekt, protože alfu ignorujeme.

```

// Konverze BGR na RGB
for(long i = 0; i < lWidthPixels * lHeightPixels; i++)// Cyklus přes všechny pixely
{
    BYTE* pPixel = (BYTE*)&pBits[i];// Aktuální pixel
    BYTE temp = pPixel[0];// Modrá složka do pomocné proměnné
    pPixel[0] = pPixel[2];// Uložení červené složky na správnou pozici
    pPixel[2] = temp;// Vložení modré složky na správnou pozici
    pPixel[3] = 255;// Konstantní alfa hodnota
}

```

Po všech nutných operacích můžeme z obrázku vygenerovat texturu. Zvolíme ji jako aktivní a nastavíme lineární filtrování. Myslím, že glTexImage2D() už nemusím vysvětlovat.

```

glGenTextures(1, &texid);// Generování jedné textury

glBindTexture(GL_TEXTURE_2D, texid);// Zvolí texturu
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);// Lineární
filtrování
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Vytvoření textury
glTexImage2D(GL_TEXTURE_2D, 0, 3, lWidthPixels, lHeightPixels, 0, GL_RGBA,
GL_UNSIGNED_BYTE, pBits);

```

Poté, co je textura vytvořena, můžeme uvolnit zabrané systémové zdroje. Už nebudeme potřebovat pomocnou ani

bitmapu ani kontext zařízení ani pPicture.

```
DeleteObject(hbmpTemp); // Smaže bitmapu
DeleteDC(hdcTemp); // Smaže kontext zařízení
pPicture->Release(); // Uvolní IPicture

return TRUE; // OK
}
```

Následující funkce zjišťuje, jestli grafická karta podporuje rozšíření EXT_fog_coord. Tento kód může být použit pouze, pokud už má program k dispozici renderovací kontext. Jestliže ho zkusíme zavolat před inicializací okna, dostaneme chyby.

Vytvoříme pole obsahující jméno našeho rozšíření. Alokujeme dynamickou paměť, do které následně zkopírujeme seznam všech podporovaných rozšíření. Pokud strstr() mezi nimi najde EXT_fog_coord, vrátíme false. (Překl.: Uvolnit dynamickou paměť!!!)

```
int Extension_Init() // Je rozšíření EXT_fog_coord podporováno?
{
    char Extension_Name[] = "EXT_fog_coord";

    // Alokace paměti pro řetězec
    char* glextstring = (char *)malloc(strlen((char *)glGetString(GL_EXTENSIONS)) + 1);
    strcpy(glextstring, (char *)glGetString(GL_EXTENSIONS)); // Grabování seznamu
    podporovaných rozšíření

    if (!strstr(glextstring, Extension_Name)) // Není podporováno?
    {
        // free(glextstring); // Překl.: Uvolnění alokované paměti !!!
        return FALSE;
    }

    free(glextstring); // Uvolnění alokované paměti
}
```

Na samém začátku programu jsme deklarovali proměnnou glFogCoordfEXT jako ukazatel na funkci. Protože už s jistotou víme, že grafická karta toto rozšíření podporuje, můžeme ho pomocí wglGetProcAddress() nastavit na správnou adresu. Od této chvíle máme k dispozici novou funkci glFogCoordfEXT(), které se předává jedna GLfloat hodnota.

```
glFogCoordfEXT = (PFNGLFOGCOORDFEXTPROC) wglGetProcAddress("glFogCoordfEXT"); //
Nastaví ukazatel na funkci

return TRUE; // OK
}
```

Při vstupu do Initialize() má program k dispozici renderovací kontext, takže se můžeme dotázat na podporu rozšíření. Pokud není dostupné, ukončíme program. Texturu nahráváme pomocí nového IPicture kódu. Pokud se z nějakého důvodu loading nezdaří, opět ukončíme program. Následuje obvyklá inicializace OpenGL.

```
BOOL Initialize(GL_Window* window, Keys* keys) // Inicializace
{
    g_window = window; // Okno
    g_keys = keys; // Klávesnice

    if (!Extension_Init()) // Je rozšíření podporováno?
    {
        return FALSE; // Konec
    }

    if (!BuildTexture("data/wall.bmp", texture[0])) // Nahrání textury
    {
        return FALSE; // Konec
    }

    glEnable(GL_TEXTURE_2D); // Zapne mapování textur
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // Černé pozadí
    glClearDepth(1.0f); // Nastavení hloubkového bufferu
    glDepthFunc(GL_LEQUAL); // Typ testování hloubky
    glEnable(GL_DEPTH_TEST); // Zapne testování hloubky
    glShadeModel(GL_SMOOTH); // Jemné stínování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // Nejlepší perspektivní korekce
}
```

Dále potřebujeme nastavit mlhu. Nejdříve ji zapneme, potom určíme lineární renderovací mód (vypadá lépe) a definujeme barvu na tmavší odstín oranžové. Startovní pozice mlhy je místo, kde bude nejméně hustá. Abychom udrželi

věci jednoduché předáme číslo 0.0f. Naopak nejvíce hustá bude s hodnotou 1.0f. Podle všech dokumentací, které jsem kdy četl, nastavení hintu na GL_NICEST způsobí, že se bude působit mlhy určovat zvlášť pro každý pixel. Předáte-li GL_FASTEST, bude se počítat pro jednotlivé vertexy, nicméně nejde vidět žádný rozdíl. Poslední glFogi() příkaz oznámí OpenGL, že chceme nastavovat mlhu v závislosti na koordinátech vertexů. To způsobí, že ji budeme moci umístit kamkoli na scénu bez toho, že bychom tak ovlivnili její zbytek.

```
// Nastavení mlhy
glEnable(GL_FOG); // Zapne mlhu
glFogi(GL_FOG_MODE, GL_LINEAR); // Lineární přechody
glFogfv(GL_FOG_COLOR, fogColor); // Barva
glFogf(GL_FOG_START, 0.0f); // Počátek
glFogf(GL_FOG_END, 1.0f); // Konec
glHint(GL_FOG_HINT, GL_NICEST); // Výpočty na jednotlivých pixelech
glFogi(GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT); // Mlha v závislosti na
souřadnicích vertexů
```

Počáteční hodnotu proměnné camz určíme na -19.0f. Protože chodbu renderujeme od -19.0f do +14.0f, bude to přesně na začátku.

```
camz = -19.0f; // Pozice kamery

return TRUE; // OK
}
```

Funkce zajišťující stisky kláves je dnes opravdu jednoduchá. Pomocí šipek nahoru a dolů nastavujeme pozici kamery ve scéně. Zároveň musíme ošetřit "přetečení", abychom se neocitli venku z chodby.

```
void Update(DWORD milliseconds) // Aktualizace scény
{
    if (g_keys->keyDown[VK_ESCAPE]) // ESC
    {
        TerminateApplication(g_window); // Konec programu
    }

    if (g_keys->keyDown[VK_F1]) // F1
    {
        ToggleFullscreen(g_window); // Změna fullscreen/okno
    }

    if (g_keys->keyDown[VK_UP] && camz < 14.0f) // Šipka nahoru
    {
        camz += (float)(milliseconds) / 100.0f; // Pohyb dopředu
    }

    if (g_keys->keyDown[VK_DOWN] && camz > -19.0f) // Šipka dolů
    {
        camz -= (float)(milliseconds) / 100.0f; // Pohyb dozadu
    }
}
```

Jsem si jistý, že už netrpělivě čekáte na vykreslování. Smažeme buffery, resetujeme matici a v závislosti na hodnotě camz se přesuneme do hloubky.

```
void Draw(void) // Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku a hloubkový
buffer
    glLoadIdentity(); // Reset matice

    glTranslatef(0.0f, 0.0f, camz); // Translace v hloubce
```

Kamera je umístěna, takže zkusíme vykreslit první quad. Bude jím zadní stěna, která by měla být kompletně ponořená v mlze. Z inicializace si jistě pamatujete, že nejhustší mlhu nastavuje hodnota GL_FOG_END; určili jsme ji na 1.0f. Mlha se aplikuje podobně jako texturové koordináty, pro nejmenší viditelnost předáme funkci glFogCoordfEXT() číslo 1.0f a pro největší 0.0f. Zadní stěna je kompletně ponořená v mlze, takže předáme všem jejím vertexům jedničku.

```
glBegin(GL_QUADS); // Zadní stěna
glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f, -2.5f, -15.0f);
glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f, -2.5f, -15.0f);
glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f,  2.5f, -15.0f);
glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f,  2.5f, -15.0f);
glEnd();
```

První dva body podlahy navazují na vertexy zadní stěny, a proto také zde uvedeme 1.0f. Přední body jsou už naopak z mlhy venku, tudíž je musíme nastavit na 0.0f. Místa ležící mezi okraji se automaticky interpolují, a tak vznikne plynulý přechod. Všechny ostatní stěny budou analogické.

```
glBegin(GL_QUADS); // Podlaha
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f);glVertex3f(-2.5f,-2.5f,-15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f);glVertex3f( 2.5f,-2.5f,-15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 1.0f);glVertex3f( 2.5f,-2.5f, 15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f);glVertex3f(-2.5f,-2.5f, 15.0f);
glEnd();

glBegin(GL_QUADS); // Strop
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f);glVertex3f(-2.5f, 2.5f,-15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f);glVertex3f( 2.5f, 2.5f,-15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 1.0f);glVertex3f( 2.5f, 2.5f, 15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f);glVertex3f(-2.5f, 2.5f, 15.0f);
glEnd();

glBegin(GL_QUADS); // Pravá stěna
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 0.0f);glVertex3f( 2.5f,-2.5f, 15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f);glVertex3f( 2.5f, 2.5f, 15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f);glVertex3f( 2.5f, 2.5f,-15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f);glVertex3f( 2.5f,-2.5f,-15.0f);
glEnd();

glBegin(GL_QUADS); // Levá stěna
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 0.0f);glVertex3f(-2.5f,-2.5f, 15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f);glVertex3f(-2.5f, 2.5f, 15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f);glVertex3f(-2.5f, 2.5f,-15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 0.0f);glVertex3f(-2.5f,-2.5f,-15.0f);
glEnd();

glFlush(); // Vyprázdnění renderovací pipeline
}
```

Doufám, že nyní už rozumíte, jak věci pracují. Čím vzdálenější je objekt, tím by měl být více ponořen v mlze a tudíž musí být nastavena hodnota 1.0f. Vždycky si také můžete pohrát s `GL_FOG_START` a `GL_FOG_END` a pozorovat, jak ovlivňují scénu. Efekt nebude pracovat podle očekávání, pokud prohodíte hodnoty. Iluze se vytvořila tím, že je zadní stěna kompletně oranžová. nejvýhodnější použití spočívá u temných koutů, kde se hráč nemůže dostat za mlhu.

Plánujete-li tento typ mlhy ve svém 3D enginu, bude možná vhodné upravovat počáteční a koncové hodnoty podle toho, kde hráč stojí, kterým směrem se dívá a podobně.

Doufám, že jste si užili tento tutoriál. Vytvářel jsem ho přes tři dny, čtyři hodiny denně. Většinu času zabralo psaní textů, které právě čtete. Původně jsme chtěli vytvořit kompletní 3D místnost s mlhou v jednom rohu, ale naneštěstí jsem měl velmi málo času na kódování. Přestože zamlžená chodba je velmi jednoduchá, vypadá perfektně a modifikace kódu pro váš projekt by také neměla být moc složitá.

Je důležité poznamenat, že toto je pouze jednou z nejrůznějších možností, jak vytvořit volumetrickou mlhu. Podobný efekt může být naprogramován pomocí blendingu, částicových systémů, maskování a podobných technologií. Pokud modifikujete pohled na scénu tak, aby byla kamera umístěna ne v chodbě, ale venku, zjistíte, že se mlha nachází uvnitř chodby.

Originální myšlenka tohoto tutoriálu ke mně dorazila už hodně dávno, což je jedním z důvodů, že jsem ztratil email. Osobě, která mi nápad zaslala, děkuji.

napsal: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 42 - Více viewportů

Tento tutoriál byl napsán pro všechny z vás, kteří se chtěli dozvědět, jak do jednoho okna zobrazit více pohledů na jednu scénu, kdy v každém probíhá jiný efekt. Jako bonus přidám získávání velikosti OpenGL okna a velice rychlý způsob aktualizace textury bez jejího znovuvytváření.

Vítejte do dalšího perfektního tutoriálu. Tentokrát se pokusíme v jednom okně zobrazit čtyři viewporty, které se budou při změně velikosti okna bez problémů zmenšovat i zvětšovat. Ve dvou z nich zapneme světla, jeden bude používat pravouhlou projekci a tři perspektivní. Abychom demu zajistili kvalitní efekty, budeme do textury postupně generovat půdorys bludiště a mapovat ji na objekty v jednotlivých viewportech.

Jakmile jednou porozumíte tomuto tutoriálu, nebudete mít nejmenší problémy při vytváření her pro více hráčů s rozdělenými scénami nebo 3D aplikací, ve kterých potřebujete několik pohledů na modelovaný objekt (půdorys, nárys, bokorys, drátěný model ap.).

Jako základní kód můžete použít buď nejnovější NeHeGL nebo IPicture. Je to, dá se říct, jedno, ale provedeme v něm několik úprav. Nejdůležitější změnu najdete ve funkci ReshapeGL(), ve které se definují dimenze scény (hlavní viewport). Všechna nastavení přesuneme do vykreslovací smyčky, zůstane zde pouze definování rozměrů hlavního okna.

```
void ReshapeGL(int width, int height)// Volá se při změně velikosti okna
{
    glViewport(0, 0, (GLsizei)(width), (GLsizei)(height));// Reset aktuálního viewportu
}
```

Druhá změna spočívá v ošetření systémové události WM_ERASEBKGD. Ukončením funkce zamezíme různému mihotání a blikání scény při roztahování okna, kdy systém automaticky maže pozadí. Pokud nerozumíte, odstraňte oba řádky a porovnejte chování okna při změně jeho velikosti.

```
// Funkce WindowProc()
switch (uMsg)// Větvení podle došlé zprávy
{
    case WM_ERASEBKGD://Okno zkouší smazat pozadí
        return 0;// Zákaz mazání (prevence blikání)
}
```

Nyní přejdeme k opravdovému kódu tohoto tutoriálu. Začneme deklarací globálních proměnných. Mx a my specifikují místnost v bludišti, ve které se právě nacházíme. Width a height definují rozměry textury, každému pixelu bludiště odpovídá jeden pixel na textuře. Pokud vaše grafická karta podporuje větší textury, zkuste zvětšit toto číslo na následující násobky dvou např. 256, 512, 1024. Ujistěte se ale, že ho nezvětšíte příliš mnoho. Má-li například okno šířku 1024 pixelů, viewporty budou poloviční, takže nemá cenu, aby textura byla větší než 512, protože by se stejně zmenšovala. To samé samozřejmě platí i pro výšku.

```
int mx, my;// Řídící proměnné cyklů

const width = 128;// Šířka textury (musí být mocninou čísla 2)
const height = 128;// Výška textury (musí být mocninou čísla 2)
```

Done povede záznam o tom, jestli už bylo generování bludiště dokončeno. Více podrobností se dozvíte později. Sp používáme k ošetření toho, aby program nebral dlouhý stisk mezerníku za několik spolu nesouvisejících stisků. Po jeho zmáčknutí resetujeme texturu a začneme kreslit bludiště od znova.

```
BOOL done;// Bludiště vygenerováno?
BOOL sp;// Flag stisku mezerníku
```

Čtyřprvková pole r, g, b ukládají složky barev pro jednotlivé viewporty. Používáme datový typ BYTE, protože se lépe získávají náhodná čísla od 0 do 255 než od 0.0f do 1.0f. Tex_data ukazuje na paměť dat textury.

```
BYTE r[4], g[4], b[4];// Čtyři náhodné barvy
BYTE* tex_data;// Data textury
```

Xrot, yrot a zrot specifikují úhel rotace 3D objektu na jednotlivých souřadnicových osách. Quadratic použijeme pro kreslení koule a válce.

```
GLfloat xrot, yrot, zrot;// Úhly rotací objektů
GLUquadricObj *quadric;// Objekt quadraticu
```

Pomocí následující funkce budeme moci snadno zabílit pixel textury na souřadnicích dmX, dmy. Tex_data představuje

ukazatel na data textury. Lokaci pixelu získáme vynásobením y pozice (dmy) šířkou řádku (width) a přičtením pozice na řádku (dmx). Protože se každý pixel skládá ze tří bytů násobíme výsledek třemi. Aby konečná barva byla bílá, musíme přiřadit číslo 255 všem třem barevným složkám.

```
void UpdateTex(int dmx, int dmy) // Zabílí určený pixel na textuře
{
    tex_data[0 + ((dmx + (width * dmy)) * 3)] = 255; // Červená složka
    tex_data[1 + ((dmx + (width * dmy)) * 3)] = 255; // Zelená složka
    tex_data[2 + ((dmx + (width * dmy)) * 3)] = 255; // Modrá složka
}
```

Reset má na starosti několik relativně důležitých úkonů. V první řadě kompletně začerní texturu a tím odstraní dosavadní bludiště, dále přiřazuje nové barvy viewportům a reinitializuje pozici v bludišti. První řádkou kódu nulujeme data textury, což ve výsledku znamená, že všechny pixely budou černé.

```
void Reset(void) // Reset textury, barev, aktuální pozice v bludišti
{
    ZeroMemory(tex_data, width * height * 3); // Nuluje paměť textury
}
```

Potřebujeme nastavit náhodnou barvu viewportů. Pro ty z vás, kteří to ještě neví, random není zase tak náhodný, jak by se mohl na první pohled zdát. Pokud vytvoříte jednoduchý program, který má vypsat deset náhodných čísel, tak samozřejmě vypíše deset náhodných čísel, která nemáte šanci předem odhadnout. Ale při příštím spuštění se bude všech deset "náhodných" čísel opakovat. Abychom tento problém odstranili, inicializujeme generátor. Pokud bychom ho ale nastavili na konstantní hodnotu (1, 2, 3...), výsledkem by opět byla při více spuštěních stejná čísla. Proto předáváme funkci srand() hodnotu aktuálního času (Překl.: počet milisekund od spuštění OS), který se samozřejmě vždy mění.

Překl.: Bývá zvykem inicializovat generátor náhodných čísel pouze jednou a to někde na začátku funkce main() a ne, jak děláme zde, při každém volání Reset() - není to špatně, ale je to zbytečné.

```
srand(GetTickCount()); // Inicializace generátoru náhodných čísel
```

V cyklu, který projde všechny čtyři viewporty nastavujeme pro každý náhodnou barvu. Mohli bychom generovat číslo v plném rozsahu (0 až 255), ale neměli bychom tak zaručeno, že nezískáme nějakou nízkou hodnotu (aby na černé byla vidět). Přičtením 128 získáme světlejší barvy.

```
for (int loop = 0; loop < 4; loop++) // Generuje čtyři náhodné barvy
{
    r[loop] = rand() % 128 + 128; // Červená složka
    g[loop] = rand() % 128 + 128; // Zelená složka
    b[loop] = rand() % 128 + 128; // Modrá složka
}
```

Nakonec nastavíme počáteční bod v bludišti - opět náhodný. Výsledkem musí být sudé číslo (zajistí násobení dvěma), protože liché pozice označují stěny mezi místnostmi.

```
mx = int(rand() % (width / 2)) * 2; // Náhodná x pozice
my = int(rand() % (height / 2)) * 2; // Náhodná y pozice
}
```

Prvním řádkem v inicializaci alokujeme dynamickou paměť pro uložení textury.

```
BOOL Initialize (GL_Window* window, Keys* keys) // Inicializace OpenGL
{
    tex_data = new BYTE[width * height * 3]; // Alokace paměti pro texturu

    g_window = window; // Okno
    g_keys = keys; // Klávesy
}
```

Voláme Reset(), abychom ji začernili a nastavili barvy viewportů.

```
Reset(); // Reset textury, barev, pozice
```

Inicializaci textury začneme nastavením clamp parametrů do rozmezí [0; 1]. Tímto odstraníme možné artefakty v podobě tenkých linek, které vznikají na okrajích textury. Příčina jejich zobrazování spočívá v lineárním filtrování, které se pokouší vyhladit texturu, ale zahrnuje do ní i její okraje. Zkuste odstranit první dva řádky a uvidíte, co myslím. Jak už jsem zmínil, nastavíme lineární filtrování a vytvoříme texturu.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP); // Clamp parametry
textury
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // Lineární
filtrování
```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
tex_data); // Vytvoří texturu

```

Pozadí budeme mazat černou barvou a hloubku jedničkou. Dále nastavíme testování hloubky na menší nebo rovno a zapneme ho.

```

glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Černé pozadí
glClearDepth(1.0f); // Nastavení depth bufferu

glDepthFunc(GL_LEQUAL); // Typ hloubkového testování
glEnable(GL_DEPTH_TEST); // Zapne testování hloubky

```

Povolení GL_COLOR_MATERIAL umožní měnit barvu textury použitím funkce glColor3f(). Také zapínáme mapování textur.

```

glEnable(GL_COLOR_MATERIAL); // Zapne vybarvování materiálů
glEnable(GL_TEXTURE_2D); // Zapne mapování textur

```

Vytvoříme a inicializujeme objekt quadratic tak, aby obsahoval normálové vektory pro světlo a texturové koordináty.

```

quadric = gluNewQuadric(); // Vytvoří objekt quadraticu
gluQuadricNormals(quadric, GLU_SMOOTH); // Normály pro světlo
gluQuadricTexture(quadric, GL_TRUE); // Texturové koordináty

```

I když ještě nemáme povoleny světla globálně, zapneme světlo 0.

```

glEnable(GL_LIGHT0); // Zapne světlo 0

return TRUE; // Vše v pořádku
}

```

Po jakékoli alokaci dynamické paměti musí přijít její uvolnění. Tuto akci vložíme do funkce Deinitialize(), která se volá před ukončením programu.

```

void Deinitialize(void) // Deinicializace
{
    delete [] tex_data; // Smaže data textury
}

```

Update má na starosti aktualizaci zobrazované scény, stisky kláves, pohyby, rotace a podobně. Celočíslnou proměnnou dir využijeme k pohybu náhodným směrem.

```

void Update(float milliseconds) // Aktualizace scény
{
    int dir; // Ukládá aktuální směr pohybu
}

```

V první fázi ošetříme klávesnici. Při stisku Esc ukončíme program, F1 přepíná mód fullscreen/okno a mezerník resetuje bludiště.

```

if (g_keys->keyDown[VK_ESCAPE]) // Klávesa Esc
{
    TerminateApplication (g_window); // Konec programu
}

if (g_keys->keyDown[VK_F1]) // Klávesa F1
{
    ToggleFullscreen (g_window); // Přepne fullscreen/okno
}

if (g_keys->keyDown[' '] && !sp) // Mezerník
{
    sp = TRUE;
    Reset(); // Resetuje scénu
}

if (!g_keys->keyDown[' ']) // Uvolnění mezerníku
{
    sp = FALSE;
}

```

Rot proměnné zvětšíme v závislosti na počtu uplynulých milisekund od minulého volání této funkce. Tím zajistíme rotaci objektů.

```
xrot += (float)(milliseconds) * 0.02f;// Aktualizace úhlů natočení
yrot += (float)(milliseconds) * 0.03f;
zrot += (float)(milliseconds) * 0.015f;
```

Kód níže zjišťuje, jestli bylo kreslení bludiště ukončeno (textura kompletně zaplněna). Nejdříve nastavíme flag done na true. Předpokládáme tedy, že už vykresleno bylo. Ve dvou vnořených cyklech procházíme jednotlivé řádky i sloupce a kontrolujeme, zda byl náš odhad správný. Pokud ne, nastavíme done na false.

Jak pracuje kód? Řídící proměnné cyklů zvyšujeme o dva, protože nám jde jen o sudé indexy v poli. Každé bludiště se skládá ze stěn (liché) a místností (sudé). Když otevřeme dveře, dostaneme se do místnosti a právě ty tedy musíme testovat. Kontroly stěn jsou samozřejmě zbytečné. Pokud se hodnota v poli rovná nule, znamená to, že jsme do něj ještě nekreslili a místnost nebyla navštívena.

```
done = TRUE;// Předpokládá se, že je už bludiště kompletní
for (int x = 0; x < width; x += 2)// Prochází všechny místnosti na ose x
{
    for (int y = 0; y < height; y += 2)// Prochází všechny místnosti na ose y
    {
        if (tex_data[((x + (width * y)) * 3)] == 0)// Pokud má pixel černou barvu
        {
            done = FALSE;// Bludiště ještě není hotové
        }
    }
}
```

Pokud byly všechny místnosti objeveny, změníme titulek okna na ...Maze Complete!, potom počkáme pět sekund, aby si ho stihl uživatel přečíst, vrátíme titulek zpět a resetujeme bludiště.

```
if (done)// Je bludiště hotové?
{
    // Změna titulku okna
    SetWindowText(g_window->hWnd, "Lesson 42: Multiple Viewports... 2003 NeHe
    Productions... Maze Complete!");
    Sleep(5000);// Zastavení na pět sekund

    SetWindowText(g_window->hWnd, "Lesson 42: Multiple Viewports... 2003 NeHe
    Productions... Building Maze!");
    Reset();// Reset bludiště a scény
}
```

Předpokládám, že pro vás následující podmínka vypadá totálně šíleně, ale vůbec není těžká. skládá se ze čtyř AND-ovaných podpodmínek a každá z nich ze dvou dalších. Všechny čtyři hlavní části jsou skoro stejné a dohromady zjišťují, jestli existuje místnost okolo aktuální pozice, která ještě nebyla navštívena. Vše si vysvětlíme na první z podpodmínek: Nejprve se ptáme, jestli jsme v místnosti vpravo už byli a potom, jestli jsou vpravo ještě nějaké místnosti (kvůli okraji textury). Pokud se červená složka pixelu rovná 255, podmínka platí. Okraj textury v daném směru nalezneme také snadno.

To samé vykonáme pro všechny směry a pokud nemáme kam jít, musíme vygenerovat novou pozici. Vše si ztřííme tím, že chceme, abychom se objevili na pozici, která už byla navštívena. Pokud ne, vygenerujeme v cyklu další souřadnici. Možná se ptáte, proč hledáme navštívenou místnost? Protože nechceme spoustu malých oddělených částí bludiště, ale jedno obrovské. Dokážete si to představit?

Zdá se vám to moc složité? Abychom udrželi velikost kódu na minimum, nekontrolujeme, jestli je mx-2 menší než nula a podobně pro všechny směry. Pokud si přejete 100% ošetření chyb, modifikujte podmínku tak, aby netestovala paměť, která už nepatří textuře.

```
// Máme kam jít?
if (((tex_data[((mx+2)+(width*my))*3]) == 255) || mx > (width-4)) && ((tex_data
[(((mx-2)+(width*my))*3)] == 255) || mx < 2) && ((tex_data[((mx+(width*(my+2)))*3)] ==
255) || my > (height-4)) && ((tex_data[((mx+(width*(my-2)))*3)] == 255) || my < 2))
{
    do
    {
        mx = int(rand() % (width / 2)) * 2;// Nová pozice
        my = int(rand() % (height / 2)) * 2;
    }
    while (tex_data[((mx + (width * my)) * 3)] == 0);// Hledá se navštívená místnost
}
```

Do proměnné dir vygenerujeme náhodné číslo od nuly do tří, které vyjadřuje směr, kterým se pokusíme jít.

```
dir = int(rand() % 4); // Náhodný směr pohybu
```

Pokud se rovná nule (směr doprava) a pokud nejsme na okraji bludiště (textury), zkontrolujeme, jestli už byla místnost vpravo navštívena. Pokud ne, označíme dveře (pixel stěny, ne místnosti) jako navštívené a projdeme do další místnosti.

```
if ((dir == 0) && (mx <= (width-4))) // Směr doprava; vpravo je místo
{
    if (tex_data[((mx+2) + (width*my)) * 3] == 0) // Místnost vpravo ještě nebyla
        navštívena
    {
        UpdateTex(mx+1, my); // Označí průchod mezi místnostmi
        mx += 2; // Posunutí doprava
    }
}
```

Analogicky ošetříme všechny další směry.

```
if ((dir == 1) && (my <= (height-4))) // Směr dolů; dole je místo
{
    if (tex_data[((mx + (width * (my+2))) * 3] == 0) // Místnost dole ještě nebyla
        navštívena
    {
        UpdateTex(mx, my+1); // Označí průchod mezi místnostmi
        my += 2; // Posunutí dolů
    }
}

if ((dir == 2) && (mx >= 2)) // Směr doleva; vlevo je místo
{
    if (tex_data[((mx-2) + (width*my)) * 3] == 0) // Místnost vlevo ještě nebyla
        navštívena
    {
        UpdateTex(mx-1, my); // Označí průchod mezi místnostmi
        mx -= 2; // Posunutí doleva
    }
}

if ((dir == 3) && (my >= 2)) // Směr nahoru; nahoře je místo
{
    if (tex_data[((mx + (width * (my-2))) * 3] == 0) // Místnost nahoře ještě nebyla
        navštívena
    {
        UpdateTex(mx, my-1); // Označí průchod mezi místnostmi
        my -= 2; // Posunutí nahoru
    }
}
```

Po přesunutí se do nové místnosti ji musíme označit.

```
UpdateTex(mx, my); // Označení nové místnosti
}
```

Vykreslování začneme netradičně. Potřebujeme zjistit velikost klientské oblasti okna, abychom mohli jednotlivé viewporty roztahovat korektně. Deklarujeme objekt struktury obdélníku a nagrabujeme do něj souřadnice okna. Šířku a výšku spočítáme jednoduchým odečtením.

```
void Draw(void) // Vykreslování
{
    RECT rect; // Struktura obdélníku

    GetClientRect(g_window->hWnd, &rect); // Grabování rozměrů okna

    int window_width = rect.right - rect.left; // Šířka okna
    int window_height = rect.bottom - rect.top; // Výška okna
}
```

Texturu musíme aktualizovat při každém překreslení scény. Nejrychlejší metodou je příkaz `glTexSubImage2D()`, který namapuje jakoukoli část obrázku na objekt ve scéně jako texturu. První parametr označuje, že chceme použít 2D texturu. Číslo úrovně detailů nastavíme na nulu a také nechceme žádný x ani y offset. Šířka a výška je určena rozměry obrázku. Každý pixel se skládá z RGB složek a data jsou ve formátu beznaménkových bytů. Poslední parametr představuje ukazatel na začátek dat.

Jak jsem již napsal, funkci `glTexSubImage2D()` velmi rychle aktualizujeme texturu bez nutnosti jejího opakovaného

smazání a sestavení. Tento příkaz je ale NEVYTVAŘÍ!!! Musíte ji tedy sestavit před první aktualizací, v našem případě se jedná o `glTexImage2D()` ve funkci `Initialize()`.

```
// Zvolí aktualizovanou texturu
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE,
tex_data);
```

Všimněte si následujícího řádku, je opravdu důležitý. Smažeme jím kompletně celou scénu. Z toho plyne, že nemažeme podscény jednotlivých viewportů postupně, ale VŠECHNY NAJEDNOU před tím, než cokoli vykreslíme. Také si všimněte, že v tuto chvíli k volání nepřidáváme mazání depth bufferu. Ten naopak ošetříme u každého viewportu zvlášť.

```
glClear(GL_COLOR_BUFFER_BIT); // Smaže obrazovku
```

Chceme vykreslit čtyři rozdílné viewporty, takže založíme cyklus od nuly do tří, pomocí řídicí proměnné nastavíme barvu.

```
for (int loop = 0; loop < 4; loop++) // Prochází viewporty
{
    glColor3ub(r[loop], g[loop], b[loop]); // Barva
```

Předtím než cokoli vykreslíme, potřebujeme nastavit viewporty. První bude umístěn vlevo nahoře. Na ose x tedy začíná na nule a na ose y v polovině okna. Šířku i výšku nastavíme na polovinu rozměrů okna. Pokud se nacházíme ve fullscreenu s rozlišením obrazovky 1024x768, bude tento viewport začínat na souřadnicích [0; 384]. Šířka se bude rovna 512 a výška 384.

```
if (loop == 0) // První scéna
{
    // Levý horní viewport, velikost poloviny okna
    glViewport(0, window_height / 2, window_width / 2, window_height / 2);
```

Po definování viewportu zvolíme projekční matici, resetujeme ji a nastavíme kolmou 2D projekci, která kompletně zaplňuje celý viewport. Levý roh spočívá na nule a pravý na polovině velikosti okna (šířka viewportu). Spodní bod je také polovinou okna a hornímu předáme nulu. Souřadnice [0; 0] tedy odpovídá levému hornímu rohu.

```
glMatrixMode(GL_PROJECTION); // Projekční matice
glLoadIdentity(); // Reset projekční matice

gluOrtho2D(0, window_width / 2, window_height / 2, 0); // Pravoúhlá projekce
}
```

Druhý viewport leží v pravém horním rohu. Opět zvolíme projekční matici a resetujeme ji. Tentokrát nenastavujeme pravoúhlou, ale perspektivní scénu.

```
if (loop == 1) // Druhá scéna
{
    // Pravý horní viewport, velikost poloviny okna
    glViewport(window_width / 2, window_height / 2, window_width / 2,
window_height / 2);
    glMatrixMode(GL_PROJECTION); // Projekční matice
    glLoadIdentity(); // Reset projekční matice

    // Perspektivní projekce
    gluPerspective(45.0, (GLfloat)(width) / (GLfloat)(height), 0.1f, 500.0);
}
```

Třetí viewport umístíme vpravo a čtvrtý vlevo dolů.

```
if (loop == 2) // Třetí scéna
{
    // Pravý dolní viewport, velikost poloviny okna
    glViewport(window_width / 2, 0, window_width / 2, window_height / 2);
    glMatrixMode(GL_PROJECTION); // Projekční matice
    glLoadIdentity(); // Reset projekční matice

    // Perspektivní projekce
    gluPerspective(45.0, (GLfloat)(width) / (GLfloat)(height), 0.1f, 500.0);
}

if (loop == 3) // Čtvrtá scéna
{
    // Levý dolní viewport, velikost poloviny okna
    glViewport(0, 0, window_width / 2, window_height / 2);
    glMatrixMode(GL_PROJECTION); // Projekční matice
```



```

glLoadIdentity();// Reset projekční matice

// Perspektivní projekce
gluPerspective(45.0, (GLfloat)(width) / (GLfloat)(height), 0.1f, 500.0);
}

```

Zvolíme matici modelview, resetujeme ji a smažeme hloubkový buffer.

```

glMatrixMode(GL_MODELVIEW);// Matice modelview
glLoadIdentity();// Reset matice

glClear(GL_DEPTH_BUFFER_BIT);// Smaže hloubkový buffer

```

Scéna prvního viewportu bude obsahovat plochý otexturovaný obdélník. Protože se nacházíme v pravoúhlé projekci, nepotřebujeme zadávat souřadnice na ose z. Objekty by se stejně nezmenšily. Vertexům předáme rozměry viewportu, který tudíž bude kompletně vyplněn.

```

if (loop == 0)// První scéna, bludiště přes celý viewport
{
    glBegin(GL_QUADS);
        glTexCoord2f(1.0f, 0.0f); glVertex2i(window_width / 2, 0);
        glTexCoord2f(0.0f, 0.0f); glVertex2i(0, 0);
        glTexCoord2f(0.0f, 1.0f); glVertex2i(0, window_height / 2);
        glTexCoord2f(1.0f, 1.0f); glVertex2i(window_width / 2, window_height /
        2);
    glEnd();
}

```

Jako druhý objekt nakreslíme kouli. Máme zapnutou perspektivu, takže se nejdříve přesuneme o 14 jednotek do obrazovky. Potom objekt natočíme o daný úhel na všech třech souřadnicových osách, zapneme světla, vykreslíme kouli o poloměru 4.0f jednotky a vypneme světla.

```

if (loop == 1)// Druhá scéna, koule
{
    glTranslatef(0.0f, 0.0f, -14.0f);// Přesun do hloubky

    glRotatef(xrot, 1.0f, 0.0f, 0.0f);// Rotace
    glRotatef(yrot, 0.0f, 1.0f, 0.0f);
    glRotatef(zrot, 0.0f, 0.0f, 1.0f);

    glEnable(GL_LIGHTING);// Zapne světlo
    gluSphere(quadric, 4.0f, 32, 32);// Koule
    glDisable(GL_LIGHTING);// Vypne světlo
}

```

Třetí viewport se velmi podobá prvnímu, ale na rozdíl od něj používá perspektivu. Přesuneme obdélník o dvě jednotky do hloubky a natočíme matici o 45 stupňů. Horní hrana se tím pádem vzdálí a spodní přiblíží. Abychom ještě přidali nějaký ten efekt, rotujeme jím také na ose z.

```

if (loop == 2)// Třetí scéna, bludiště na rovině
{
    glTranslatef(0.0f, 0.0f, -2.0f);// Přesun do hloubky

    glRotatef(-45.0f, 1.0f, 0.0f, 0.0f);// Rotace o 45 stupňů
    glRotatef(zrot / 1.5f, 0.0f, 0.0f, 1.0f);// Rotace na ose z

    glBegin(GL_QUADS);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 0.0f);
    glEnd();
}

```

Čtvrtým a posledním objektem je válec, který se nachází sedm jednotek hluboko ve scéně a rotuje na všech třech osách. Zapneme světla a potom se ještě posuneme o dvě jednotky (o polovinu jeho délky) na ose z. Chceme, aby se otáčel okolo svého středu a ne konce. Vykreslíme ho a vypneme světla.

```

if (loop == 3)// Třetí scéna, válec
{
    glTranslatef(0.0f,0.0f,-7.0f);// Přesun do hloubky

    glRotatef(-xrot/2,1.0f,0.0f,0.0f);// Rotace

```

```
glRotatef(-yrot/2,0.0f,1.0f,0.0f);
glRotatef(-zrot/2,0.0f,0.0f,1.0f);

glEnable(GL_LIGHTING); // Zapne světlo
glTranslatef(0.0f,0.0f,-2.0f); // Vycentrování
gluCylinder(quadric,1.5f,1.5f,4.0f,32,16); // Válec
glDisable(GL_LIGHTING); // Vypne světlo
    }
}
```

Na konci vykreslování flushneme renderovací pipeline.

```
glFlush(); // Vyprázdnění pipeline
}
```

Doufám, že tento tutoriál zodpověděl všechny vaše otázky ohledně více viewportů v jednom okně. Nyní také znáte jeden z mnoha způsobů generování bludiště a umíte upravit texturu bez jejího komplikovaného mazání a znovuvytváření. Co víc si přát?

napсал: Jeff Molofee - NeHe <nehe (zavináč) connect.ab.ca>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 43 - FreeType Fonty v OpenGL

Použitím knihovny *FreeType Font rendering library* můžete snadno vypisovat vyhlazené znaky, které vypadají mnohem lépe než písmena u bitmapových fontů z lekce 13. Náš text bude mít ale i jiné výhody - bezproblémová rotace, dobrá spolupráce s OpenGL vybíracími (*picking*) funkcemi a víceřádkové řetězce.

Motivace: Tady máte ukázky bitmapového fontu vytvořeného pomocí WGL a FreeType fontu. Oba jsou Arial Black Kurzíva.



Základní problém s použitím bitmapových fontů je, že OpenGL bitmapy jsou binárními obrázky. To znamená, že si OpenGL pamatuje pouze 1 bit na 1 pixel. Zoomujeme-li na text vytvořený pomocí WGL, výsledek vypadá přibližně takto:



Protože jsou bitmapy binární obrázky, nejsou okolo nich šedé pixely a to znamená, že vypadají hůř. Naštěstí je velmi jednoduché pomocí GNU FreeType knihovny vytvořit dobře vypadající fonty. Tuto knihovnu používá i Blizzard ve svých hrách, takže musí být opravdu dobrá :-)). Opět ukázka zvětšeného textu, tentokrát s knihovnou FreeType:



Jak můžeme vidět, okolo okrajů se nachází spousta šedých pixelů, které jsou typické pro vyhlazené (anti-aliasované) fonty. Šedé pixely vylepšují znaky při pohledu z dálky.

Knihovnu GNU FreeType si můžete stáhnout na adrese <http://gnuwin32.sourceforge.net/packages/freetype.htm>. Konkrétně se jedná o binární a vývojářské soubory. Při instalaci si určitě všimnete licenční podmínky. Hovoří se v ní, že při použití ve vlastních programech, musíte někde v dokumentaci uvést kredit.

Po instalaci potřebujeme nastavit MSVC, aby umělo používat FreeType. V menu Project - Settings - Link se ujistěte, že jste spolu s `opengl32.lib`, `glu32.lib`, `glaux.lib` a podobnými přidali do Object/libraries i `libfreetype.lib`.

Dále potřebujeme v Tools - Options - Directories přidat cesty k hlavičkovým souborům. Pod Show Directories for vybereme Include Files a poklikáme na prázdný řádek dole v seznamu. Objeví se tlačítko se třemi tečkami, které použijeme pro výběr adresáře. Takto přidáme:

```
C:\PROGRAM FILES\GNUWIN32\INCLUDE\FREETYPE32
```

a

```
C:\PROGRAM FILES\GNUWIN\INCLUDE
```

Pod Show Directories For vybereme Library Files a přidáme

```
C:\PROGRAM FILES\GNUWIN32\LIB
```

Na tomto místě bychom měli být schopni kompilovat programy, které používají FreeType, ale nepůjdou spustit bez dynamické knihovny `freetype-6.dll`. Kopii tohoto souboru naleznete v `GNUWIN32\BIN`. Je třeba ji umístit do adresáře, ve kterém systém při spouštění programů knihovny hledá (např. `C:\PROGRAM FILES\MICROSOFT\VISUAL STUDIO\VC98\BIN` nebo `C:\WINDOWS\SYSTEM` pro WIN9x, `C:\WINNT\SYSTEM32` pro WIN NT/2000/XP). Překl.: Osobně doporučuji podobné DLL knihovny výhradně vkládat do adresáře, ve kterém se nachází spouštěný exe soubor, protože až budete váš program někomu kopírovat, nikdy na ně nezapomenete (Tak ten tvůj program mi nešel spustit!).

OK, tak teď už konečně můžeme začít programovat. Jako základ vezmeme lekci 13. Zkopírujeme soubor lesson13.cpp a přidáme ho do projektu. Stejně tak zkopírujeme dva nové soubory freetype.h a freetype.cpp, do kterých budeme přidávat všechny FreeTypový kód. Až skončíme, budeme mít jednoduchou FreeType knihovnu, kterou budeme moci využít i v jiných OpenGL programech.

Začneme vytvářením freetype.h. Samozřejmě musíme nadefinovat hlavičky FreeType a OpenGL. Také přidáme pár užitečných částí ze Standard Template Library (STL). Konkrétně se jedná o třídy vyjímek, které nám zjednoduší vytváření pěkných debugových zpráv. Použití STL zvyšuje šanci, že někdo jiný, kdo používá náš kód, úspěšně zachytí všechny poslané výjimky.

```
#ifndef FREE_NEHE_H
#define FREE_NEHE_H

#include <windows.h>

// FreeType hlavičky
#include <ft2build.h>
#include <freetype/freetype.h>
#include <freetype/ftglyph.h>
#include <freetype/ftoutln.h>
#include <freetype/fttrigon.h>

// OpenGL hlavičky
#include <GL/gl.h>
#include <GL/glu.h>

// STL hlavičky
#include <vector>
#include <string>

// STL vyjímky
#include <stdexcept>
```

Následující pragma MSVC zabráni, aby oznamovalo zbytečná varování o vektorech řetězců.

```
#pragma warning(disable: 4786)
```

Všechny informace, které každý font potřebuje, dáme do jedné struktury (toto ulehčí práci s více písmi). Jak jsme se naučili v lekci 13, když WGL vytváří font, generuje sadu display listů s postupně se zvyšujícím ID. To je šikovné, protože díky tomu můžeme pro vypsání celého řetězce použít jediný příkaz glCallLists(). V naší knihovně nastavíme všechno úplně stejně, což znamená, že pole list_base bude ukládat prvních 128 display listů jednotlivých znaků. Protože se chystáme pro vykreslování použít textury, potřebujeme také uložit 128 asociovaných textur. Poslední, co uděláme, je deklarování proměnné označující výšku vytvářeného fontu v pixelech, která nám umožní vypisovat i zalomení řádků označené \n v řetězci.

```
namespace freetype// Proti vícenásobnému použití stejných identifikátorů
{
    using std::vector;// Možnost psát jen vector namísto std::vector
    using std::string;// To samé pro string

    struct font_data// Zapouzdření všeho do struktury
    {
        float h;// Výška fontu
        GLuint* textures;// ID textur
        GLuint list_base;// ID prvního display listu

        void init(const char* fname, unsigned int h);// Vytvoření písma s výškou h ze
        souboru fname
        void clean();// Uvolnění všech prostředků spojených s fontem
    };
};
```

Funkce print() vykreslí zadaný text na souřadnicích x, y. Modelview matice bude také aplikovaná na text.

```
void print(const font_data &ft_font, float x, float y, const char *fmt, ...);//
Vykreslí text
}

#endif
```

To je konec hlavičkového souboru freetype.h, teď otevřeme freetype.cpp.

```
#include "freetype.h"// Vloží freetype.h
```

```
namespace freetype
{
```

Pro vykreslení znaků budeme používat textury, které musí samozřejmě mít rozměry mocniny čísla 2. Následující funkce vrátí první mocninu dvojky, která se rovná nebo je větší než předané číslo.

```
inline int next_p2(int a) // Vrátí následující mocninu čísla 2
{
    int rval = 1; // Nastaví bit vpravo do jedničky
    while(rval < a) // Dokud je nalezená mocnina menší než minimum
    {
        rval <<= 1; // Získání další mocniny (rotace bitů doleva, rychlejší způsob,
            jak napsat rval *= 2)
    }
    return rval; // Vrácení výsledku
}
```

Další funkce, kterou budeme potřebovat, je srdcem tohoto kódu. Make_dlist() vytvoří display list podle poslaného znaku, parametr FT_Face představuje objekt, který FreeType používá pro uchování informací o fontu. Funkci se dále předává ID základního display listu a ukazatel na texturu.

```
void make_dlist(FT_Face face, char ch, GLuint list_base, GLuint* tex_base) // Vytvoří
display list pro daný znak
{
```

Na začátku požádáme FreeType o vykreslení daného znaku do bitmapy.

```
if(FT_Load_Glyph(face, FT_Get_Char_Index(face, ch), FT_LOAD_DEFAULT)) // Načte
glyph znaku
{
    throw std::runtime_error("FT_Load_Glyph failed");
}

FT_Glyph glyph; // Glyph objekt
if(FT_Get_Glyph(face->glyph, &glyph)) // Přesun glyphu do glyph objektu
{
    throw std::runtime_error("FT_Get_Glyph failed");
}

FT_Glyph_To_Bitmap(&glyph, ft_render_mode_normal, 0, 1); // Konvertování glyphu
na bitmapu
FT_BitmapGlyph bitmap_glyph = (FT_BitmapGlyph)glyph;

FT_Bitmap& bitmap = bitmap_glyph->bitmap; // Reference ulehčí přístup k bitmapě
```

Ted, když máme pomocí FreeType vytvořenu bitmapu, potřebujeme do ní doplnit prázdné pixely, abychom ji mohli použít v OpenGL. Je důležité zapamatovat si, že zatímco OpenGL používá termín bitmapa k označení binárních obrázků, ve FreeType bitmapa ukládá 8 bitů informace na pixel, takže může ukládat i šedé složky, které jsou potřebné pro vyhlazený text.

```
int width = next_p2(bitmap.width); // Velikost textury - mocnina čísla 2
int height = next_p2(bitmap.rows);

GLubyte* expanded_data = new GLubyte[2 * width * height]; // Alokace paměti
```

Všimněte si, že používáme dvoukanalovou bitmapu, první kanál pro zářivost a druhý pro alfu. Oba přiřadíme hodnotě, kterou jsme našli ve FreeType bitmapě. Ternární operátor ? : použijeme pro určení nulové hodnoty, pokud se nacházíme v okrajové zóně (zvětšení pro mocninu 2), v ostatních případech platí hodnota převzatá z FreeType bitmapy.

```
for(int j = 0; j < height; j++)
{
    for(int i = 0; i < width; i++)
    {
        expanded_data[2 * (i + j*width)] = expanded_data[2 * (i + j*width) + 1]
            = (i >= bitmap.width || j >= bitmap.rows) ? 0 : bitmap.buffer[i +
                bitmap.width*j];
    }
}
```

Jsme hotovi, takže se můžeme pustit do vytváření OpenGL textury. Protože zahrneme alfa kanál, černé části bitapy

budou vždy průhledné a okraje textu budou plynule průsvitné. Text by měl vypadat správně na jakémkoli podkladě. Jak už jsem napsal, texturu vytváříme ze složek luminance a alfa.

```
// Nastavení parametrů textury
glBindTexture(GL_TEXTURE_2D, tex_base[ch]);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_LUMINANCE_ALPHA,
GL_UNSIGNED_BYTE, expanded_data);

delete [] expanded_data;// Uvolnění paměti bitmapy
```

Na vykreslení znaku použijeme otexturované čtyřúhelníky. To znamená, že bude jednoduché otáčet, zvětšovat i zmenšovat text a dokonce bude od OpenGL dědit barvu.

```
glNewList(list_base + ch, GL_COMPILE);// Vytvoření display listu
glBindTexture(GL_TEXTURE_2D, tex_base[ch]);
```

Nejdřív pohneme kamerou trochu doprava, aby byl znak vycentrovaný mezi minulým a následujícím. Uložíme matici a pokud pracujeme se znakem typu g nebo y, posuneme se trochu dolů.

```
glTranslatef(bitmap_glyph->left, 0, 0);// Vycentrování znaku mezi minulým a
následujícím

glPushMatrix();
glTranslatef(0, bitmap_glyph->top - bitmap.rows, 0);// Posun o trochu
dolů
```

Musíme počítat s faktem, že mnoho textur je na okraji vyplněných prázdným místem. Zjistíme, jaká část textury je znakem používána a tuto hodnotu uložíme do pomocných proměnných x a y, kterou před kreslením předáme funkci glTexCoord2d().

```
float x = (float)bitmap.width / (float)width;
float y = (float)bitmap.rows / (float)height;
```

Na tomto místě vykreslíme otexturovaný obdélník. Bitmapa, kterou jsme získali pomocí FreeType není orientovaná přesně tak, jak by měla, ale to nám nevadí, protože můžeme explicitně určit polohu pro správné zarovnání.

```
glBegin(GL_QUADS);// Vykreslení znaku
glTexCoord2d(0, 0); glVertex2f(0, bitmap.rows);
glTexCoord2d(0, y); glVertex2f(0, 0);
glTexCoord2d(x, y); glVertex2f(bitmap.width, 0);
glTexCoord2d(x, 0); glVertex2f(bitmap.width, bitmap.rows);
glEnd();
glPopMatrix();

glTranslatef(face->glyph->advance.x >> 6, 0, 0);
```

Inkrementujeme pozici v rastru stejně, jako bychom pracovali s bitmapovým fontem. To je nutné pouze, pokud bychom chtěli spočítat aktuální délku textu. Proto jsem řádek zakomentoval.

```
// glBitmap(0, 0, 0, 0, face->glyph->advance.x >> 6, 0, NULL);

glEndList();// Ukončíme display list
}
```

Další funkce, kterou se chystáme vytvořit, bude používat make_dlist() pro vytvoření množiny display listů odpovídajících danému souboru s fontem a výšce v pixelech. FreeType používá truetype fonty, takže budeme potřebovat nějaký .ttf soubor s fontem. Truetypeová písma jsou velmi běžná, existuje spousta míst na internetu, kde si je můžete stáhnout. Jednodušší bude ale podívat se do adresáře windows/fonts.

```
void font_data::init(const char * fname, unsigned int h)// Vytvoření fontu
{
    textures = new GLuint[128];// Paměť pro ID textur

    this->h = h;

    FT_Library library;// Vytvoření FreeType

    if (FT_Init_FreeType(&library))// Inicializace FreeType
    {
        throw std::runtime_error("FT_Init_FreeType failed");
    }
}
```

```
FT_Face face;// Objekt pro informace o fontu
```

Na tomto místě se pokusíme načíst ze souboru data fontu. Ze všech míst, kde se kód může zaseknout je právě toto nejčastější, protože soubor např. nemusí existovat nebo může být nějakým způsobem poškozen.

```
if (FT_New_Face(library, fname, 0, &face))// Načtení fontu ze souboru
{
    throw std::runtime_error("FT_New_Face failed (there is probably a problem
        with your font file)");
}
```

Z nějakým nevysvětlitelných důvodů měří FreeType velikost písma v 1/64-nách pixelů. Proto, abychom měli font vysoký h pixelů, musíme předávat velikost násobenou číslem 64. $H \ll 6$ je jen rychlejší způsob psaní této operace.

```
FT_Set_Char_Size(face, h << 6, h << 6, 96, 96);
```

Překl.: Anglické znakové sadě stačí pouze 128 znaků, ale čeština obsahuje navíc háčky a čárky, takže pokud je chcete používat, musíte upravit kód.

```
list_base = glGenLists(128);// 128 display listů a textur
glGenTextures(128, textures);

for(unsigned char i = 0; i < 128; i++)// Vytvoření display listů znaků
{
    make_dlist(face, i, list_base, textures);
}
```

Protože všechna data máme uložena v display listech a texturách, můžeme uvolnit použité zdroje FreeType.

```
FT_Done_Face(face);// Uvolnění zdrojů
FT_Done_FreeType(library);
}
```

Vytvoříme clean() funkci, která uvolní všechny prostředky spojené s display listy a texturami.

```
void font_data::clean()
{
    glDeleteLists(list_base, 128);
    glDeleteTextures(128, textures);
    delete [] textures;
}
```

Následují dvě pomocné funkce pro print(), která bude chtít operovat ne v OpenGL jednotkách, ale v pixelových souřadnicích okna. Nula se bude nacházet v levém horním rohu.

```
inline void pushScreenCoordinateMatrix()// Přepne do pravoúhlé projekce
{
    glPushAttrib(GL_TRANSFORM_BIT);
    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(viewport[0], viewport[2], viewport[1], viewport[3]);
    glPopAttrib();
}

inline void pop_projection_matrix()// Obnoví perspektivu
{
    glPushAttrib(GL_TRANSFORM_BIT);
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glPopAttrib();
}
```

Nová funkce print() vypadá velmi podobně jako ta z lekce 13, ale je tu pár rozdílů. Nastavíme jiné OpenGL flagy, protože používáme pouze dvoukanalové textury namísto bitmap. Abychom zajistili přechody na nové řádky, přidáme několik extra výpočtů a samozřejmě nesmíme zapomenout na zajištění toho, aby všechna nastavení byla po výstupu z funkce ve stejném stavu jako před vstupem.

```
void print(const font_data &ft_font, float x, float y, const char *fmt, ...)//
Rendering textu
```

```

{
    pushScreenCoordinateMatrix();// Souřadná soustava v pixelech
    GLuint font = ft_font.list_base;
    float h = ft_font.h / 0.63f;// Větší mezera mezi řádky
    char text[256];// Výsledný řetězec
    va_list ap;// Ukazatel na argumenty funkce

    if (fmt == NULL)// Byl předán text?
    {
        *text = 0;// Nic nedělat
    }
    else
    {
        va_start(ap, fmt);// Analýza řetězce na proměnné
        vsprintf(text, fmt, ap);// Konvertování symbolů na čísla
        va_end(ap);// Výsledky jsou uloženy do text
    }
}

```

Následující kód rozdělí daný text na sadu řádků. Velmi jednoduše by se to dalo provést pomocí regulárních výrazů, jedna taková knihovna je dostupná např. na boost.org, ale my nic takového používat nebudeme - jednoduše se snažím udržet kód bez zbytečných závislostí na méně nutných knihovnách.

```

// Rozdělení řetězce na jednotlivé řádky
const char *start_line = text;
vector<string> lines;

for(const char *c = text; *c; c++)
{
    if(*c == '\n')
    {
        string line;

        for(const char *n = start_line; n < c; n++)
        {
            line.append(1, *n);
        }

        lines.push_back(line);
        start_line = c + 1;
    }
}

if(start_line)
{
    string line;
    for(const char *n = start_line; n < c; n++)
    {
        line.append(1, *n);
    }

    lines.push_back(line);
}

```

Zálohujeme všechny OpenGL parametry a potom je nastavíme na nutné hodnoty.

```

glPushAttrib(GL_LIST_BIT | GL_CURRENT_BIT | GL_ENABLE_BIT | GL_TRANSFORM_BIT);

glMatrixMode(GL_MODELVIEW);
glDisable(GL_LIGHTING);
glEnable(GL_TEXTURE_2D);
glDisable(GL_DEPTH_TEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glListBase(font);

```

Všechny transformace, které provedeme na modelview matici před voláním této funkce, se projeví i na textu samotném. To znamená, že při výpisu textu máme možnost rotovat nebo měnit jeho velikost. Nejpřirozenější cestou by bylo ponechat původní matici, tak jak byla, ale to nebude pracovat, protože chceme mít kontrolu nad pozicí textu. Další možností by bylo vytvořit kopii matice a mezi `glTranslatef()` a `glCallLists()` ji aplikovat, nicméně měřítko projekční matice teď už není v OpenGL jednotkách, ale v pixelech, takže bychom získali trošku odlišný efekt, než by někdo mohl očekávat. Přes toto bychom se také mohli dostat neresetován projekční matice uvnitř `print()`. To je v některých situacích

docela dobrý nápad, ale pokud to budete zkoušet, zajistěte, že fonty budou mít odpovídající velikost (jsou nastaveny na 32x32, ale vy pravděpodobně budete potřebovat něco kolem 0,01x0,01). Zkuste uhodnout, kterou cestou jdeme my :-)

```
float modelview_matrix[16];
glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix);
```

Zobrazování textu se děje právě na tomto místě. Pro každý řádek resetujeme matici, aby začínal na správné pozici. Všimněte si, že místo posunu dolů o výšku h, ji raději rovnou resetujeme. To proto, že se při vykreslení každého znaku posouváme doprava na pozici znaku za ním.

```
for(int i = 0; i < lines.size(); i++)// Prochází jednotlivé řádky textu
{
    glPushMatrix();// Záloha matice
    glLoadIdentity();// Resetování matice
    glTranslatef(x, y - h*i, 0);// Přesun na odpovídající pozici
    glMultMatrixf(modelview_matrix);
```

Pokud byste potřebovali zjistit délku textu, který vytváříte odkomentujete následující řádky. Pokud se tak rozhodnete, musíte odkomentovat i příkaz glBitmap() v make_dlist().

```
    // glRasterPos2f(0, 0);

    glCallLists(lines[i].length(), GL_UNSIGNED_BYTE, lines[i].c_str());//
    Vykreslí řádek textu

    // float rpos[4];
    // glGetFloatv(GL_CURRENT_RASTER_POSITION, rpos);
    // float len = x - rpos[0];

    glPopMatrix();// Obnovení matice
}

glPopAttrib();// Obnovení OpenGL flagů

pop_projection_matrix();// Obnovení perspektivy
}
} // Konec namespace
```

Knihovna je teď kompletní. Abychom mohli vidět výsledek, otevřeme soubor lesson13.cpp a provedeme v něm několik menších změn. Za includování hlavičkových souborů vložte i freetype.h.

```
#include "freetype.h"// Vložení freetype
```

A když už jsme tu, deklaruje i globální objekt font_data.

```
freetype::font_data our_font;// Informace pro vytvářený font
```

Dále potřebujeme font inicializovat..

```
// InitGL()
our_font.init("test.TTF", 16);// Vytvoření fontu
```

... a při skončení programu odstranit.

```
// KillGLWindow()
our_font.clean();
```

Do funkce DrawGLScene() doplníme výpis FreeType fontu, který bude navíc rotovat a měnit svou velikost.

```
int DrawGLScene(GLvoid)// Všechno vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smazání bufferů
    glLoadIdentity();// Reset matice

    glTranslatef(0.0f, 0.0f, -1.0f);// Posun o jednotku do obrazovky
    glColor3ub(0, 0, 0xff);// Modrý text

    // Vykreslení WGL textu
    glRasterPos2f(-0.40f, 0.35f);
    glPrint("Active WGL Bitmap Text With NeHe - %7.2f", cnt1);

    // Vykreslení FreeType fontu
    glColor3ub(0xff,0,0);// Červený text
```

```

glPushMatrix();
glLoadIdentity();
glRotatef(cnt1, 0, 0, 1);
glScalef(1, 0.8 + 0.3 * cos(cnt1 / 5), 1);
glTranslatef(-180, 0, 0);

freetype::print(our_font, 320, 240, "Active FreeType Text - %7.2f", cnt1);

glPopMatrix();

```

Chcete-li otestovat i přechody na nové řádky, odstraňte komentář.

```

// freetype::print(our_font, 320, 200, "Here\nthere\nbe\n\nnewlines\n.", cnt1);

cnt1 += 0.051f;// Zvětšení hodnot v čítačích
cnt2 += 0.005f;

return TRUE;// Vše OK
}

```

Nakonec musíme přidat kód pro odchyťávání vyjímek. Přejdeme do WinMain() a na začátku vyhledáme sekci try { }.

```

// WinMain()
MSG msg;// Struktura zprávy
BOOL done = FALSE;// Proměnná pro ukončení cyklu

try// Sekce, ve které se budou zachytávat vyjímky
{

```

Konec funkce modifikujeme přidáním catch { }, které vypíše text vyjímky.

```

    KillGLWindow();// Zrušení okna
}
catch (std::exception &e)// Ošetření vyjímek
{
    MessageBox(NULL, e.what(), "CAUGHT AN EXCEPTION", MB_OK | MB_ICONINFORMATION);
}

```

Tak a teď, když v programu nastane vyjímka, se zobrazí text oznamující uživateli, co se stalo. Pozor, tento kód může zpomalit váš program, takže se možná při kompilování konečné verze bude hodit vypnutí odchyťávání vyjímek (Project->Settings->C/C++, "C++ Language").

Zkompilujte program. Po spuštění byste měli vidět pěkný text renderovaný pomocí FreeType, který se pohybuje okolo originálního textu z lekce 13.

Obecné poznámky

Pravděpodobně budete chtít právě vytvořenou FreeType knihovnu ještě dále vylepšit. Konkrétně se může jednat např. o zarovnávání textu na střed. K tomu budete potřebovat nějakým způsobem zjistit jeho délku. Jedním způsobem může být vložení příkazu glBitmap() do display listu, který bude modifikovat pozici v rastru. Prakticky všechno už je v kódu připraveno, stačí odkomentářovat příslušné příkazy.

FreeType fonty zabírají také mnohem více místa než obyčejný WGL bitmapový font. Pokud z nějakého důvodu potřebujete šetřit texturovací paměť, zkuste vytvořit jednu texturu, která bude obsahovat matici všech znaků, stejnou jaká je v lekci 13.

Na rozdíl od bitmap obdélníky s namapovanou texturou reprezentující text dobře spolupracují s OpenGL picking funkcemi (lekce 32), což velmi usnadňuje zjištění, jestli někdo na text klikl myší nebo přes něj přešel.

Nakonec uvádím odkazy na několik knihoven fontů pro OpenGL. Záleží pouze na vás, jestli je budete chtít použít místo tohoto kódu.

GLTT - tato knihovna je už relativně stará. Je založena na FreeType1. Předpokládám, že pro kompilaci v MSVC6 budete potřebovat nalézt kopii staré distribuce FreeType1.

OGLFT - je založeno na FreeType2, kompilace pod MSVC dá možná trochu práce, protože byla zaměřena především pro Linux...

FTGL - třetí knihovna založená na FreeType, vyvíjena pro OS X.

FNT - knihovna, která není založena na FreeType, je částí PLIB, má hezký interface, používá vlastní formát fontů, kompilace pod MSVC6 s minimem potíží...

napsal: Sven Olsen <sven (zavináč) sccs.swarthmore.edu>
do slovenštiny přeložil: Pavel Hradský - PcMaster <pcmaster (zavináč) stonline.sk>
do češtiny přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 44 - Čočkové efekty

Čočkové efekty vznikají po dopadu paprsku světla např. na objektiv kamery nebo fotoaparátu. Podíváte-li se na zářivou vyvolanou čočkou, zjistíte, že jednotlivé útvary mají jednu společnou věc. Pozorovateli se zdá, jako by se všechny pohybovaly skrz střed scény. S tímto na mysli můžeme osu zjednoduše odstranit a vytvářet vše ve 2D. Jediný problém související s nepřítomností z souřadnice je, jak zjistit, jestli se zdroj světla nachází ve výhledu kamery nebo ne. Připravte se proto na trochu matematiky.

Ahoj všichni, jsem tu s dalším tutoriálem. Rozšíříme naši třídu `glCamera` o čočkové efekty (Překl.: v originále Lens Flare - čočková záře), které jsou sice náročné na množství výpočtů, ale vypadají opravdu realisticky. Jak už jsem napsal, do třídy kamery přidáme možnost, jak zjistit, jestli se bod nebo koule nachází ve výhledu kamery na scénu. Neměli bychom však při tom odrovnat procesor.

Jsem na rozpacích, ale musím zmínit, že třída kamery obsahuje chybu. Před tím, než začneme, musíme ji záplatovat. Funkci `SetPerspective()` upravte podle následujícího vzoru.

```
void glCamera::SetPerspective()
{
    GLfloat Matrix[16]; // Pole pro modelview matici
    glVector v; // Směr a rychlost kamery

    glRotatef(m_HeadingDegrees, 0.0f, 1.0f, 0.0f); // Výpočet směrového vektoru
    glRotatef(m_PitchDegrees, 1.0f, 0.0f, 0.0f);

    glGetFloatv(GL_MODELVIEW_MATRIX, Matrix); // Získání matice

    m_DirectionVector.i = Matrix[8]; // Směrový vektor
    m_DirectionVector.j = Matrix[9];
    m_DirectionVector.k = -Matrix[10]; // Musí být invertován

    glLoadIdentity(); // Reset matice

    glRotatef(m_PitchDegrees, 1.0f, 0.0f, 0.0f); // Správná orientace scény
    glRotatef(m_HeadingDegrees, 0.0f, 1.0f, 0.0f);

    v = m_DirectionVector; // Aktualizovat směr podle rychlosti
    v *= m_ForwardVelocity;

    m_Position.x += v.i; // Inkrementace pozice vektorem
    m_Position.y += v.j;
    m_Position.z += v.k;

    glTranslatef(-m_Position.x, -m_Position.y, -m_Position.z); // Přesun na novou pozici
}
```

Před tím, než se pustíme do kódování, si nakreslíme čtyři textury pro čočkovou záři. První představuje mlhavou zářivou nebo sálání a bude vždy umístována na pozici světelného zdroje. Pomocí další můžeme vytvářet záblesky zářící ven ze světla. Opět ji umístíme na jeho pozici. Třetí se vzhledem podobá první textuře, ale uprostřed je mnohem více definovaná. Budeme jí dynamicky pohybovat přes scénu. Poslední textura je zářící, dutě vypadající kruh, který budeme přesunovat v závislosti na pozici a orientaci kamery. Existují samozřejmě i další typy textur; pro další informace se podívejte na reference uvedené na konci tutoriálu.



Teď byste už měli mít alespoň představu, co budeme vykreslovat. Obecně se dá říci, že se čočkový efekt nikdy neobjeví, dokud se nepodíváme do zdroje světla nebo alespoň jeho směrem, a proto potřebujeme najít cestu, jak zjistit,

jestli se daný bod (pozice světla) nachází ve výhledu kamery nebo ne. Můžeme vynásobit modelview a projekční matici a potom nalézt ořezávací roviny, které OpenGL používá. Druhá možnost je použít rozšíření GL_HP_occlusion_test nebo GL_NV_occlusion_query, ale ne každá grafická karta je implementuje. My použijeme věc, která funguje vždy a všude - matematiku.

Překl.: Občas, když je něco málo vysvětlené, přidávám vlastní texty, ale teď to po mně prosím nechtějte :-)

```
void glCamera::UpdateFrustum() // Získání ořezávacích rovin
{
    GLfloat clip[16]; // Pomocná matice
    GLfloat proj[16]; // Projekční matice
    GLfloat modl[16]; // Modelview matice
    GLfloat t; // Pomocná

    glGetFloatv(GL_PROJECTION_MATRIX, proj); // Získání projekční matice
    glGetFloatv(GL_MODELVIEW_MATRIX, modl); // Získání modelview matice

    // Vynásobí projekční matici pomocí modelview
    clip[ 0] = modl[ 0] * proj[ 0] + modl[ 1] * proj[ 4] + modl[ 2] * proj[ 8] + modl
    [ 3] * proj[12];
    clip[ 1] = modl[ 0] * proj[ 1] + modl[ 1] * proj[ 5] + modl[ 2] * proj[ 9] + modl
    [ 3] * proj[13];
    clip[ 2] = modl[ 0] * proj[ 2] + modl[ 1] * proj[ 6] + modl[ 2] * proj[10] + modl
    [ 3] * proj[14];
    clip[ 3] = modl[ 0] * proj[ 3] + modl[ 1] * proj[ 7] + modl[ 2] * proj[11] + modl
    [ 3] * proj[15];

    clip[ 4] = modl[ 4] * proj[ 0] + modl[ 5] * proj[ 4] + modl[ 6] * proj[ 8] + modl
    [ 7] * proj[12];
    clip[ 5] = modl[ 4] * proj[ 1] + modl[ 5] * proj[ 5] + modl[ 6] * proj[ 9] + modl
    [ 7] * proj[13];
    clip[ 6] = modl[ 4] * proj[ 2] + modl[ 5] * proj[ 6] + modl[ 6] * proj[10] + modl
    [ 7] * proj[14];
    clip[ 7] = modl[ 4] * proj[ 3] + modl[ 5] * proj[ 7] + modl[ 6] * proj[11] + modl
    [ 7] * proj[15];

    clip[ 8] = modl[ 8] * proj[ 0] + modl[ 9] * proj[ 4] + modl[10] * proj[ 8] + modl
    [11] * proj[12];
    clip[ 9] = modl[ 8] * proj[ 1] + modl[ 9] * proj[ 5] + modl[10] * proj[ 9] + modl
    [11] * proj[13];
    clip[10] = modl[ 8] * proj[ 2] + modl[ 9] * proj[ 6] + modl[10] * proj[10] + modl
    [11] * proj[14];
    clip[11] = modl[ 8] * proj[ 3] + modl[ 9] * proj[ 7] + modl[10] * proj[11] + modl
    [11] * proj[15];

    clip[12] = modl[12] * proj[ 0] + modl[13] * proj[ 4] + modl[14] * proj[ 8] + modl
    [15] * proj[12];
    clip[13] = modl[12] * proj[ 1] + modl[13] * proj[ 5] + modl[14] * proj[ 9] + modl
    [15] * proj[13];
    clip[14] = modl[12] * proj[ 2] + modl[13] * proj[ 6] + modl[14] * proj[10] + modl
    [15] * proj[14];
    clip[15] = modl[12] * proj[ 3] + modl[13] * proj[ 7] + modl[14] * proj[11] + modl
    [15] * proj[15];

    m_Frustum[0][0] = clip[ 3] - clip[ 0]; // Získání pravé roviny
    m_Frustum[0][1] = clip[ 7] - clip[ 4];
    m_Frustum[0][2] = clip[11] - clip[ 8];
    m_Frustum[0][3] = clip[15] - clip[12];

    // Normalizace výsledku
    t = GLfloat(sqrt( m_Frustum[0][0] * m_Frustum[0][0] + m_Frustum[0][1] * m_Frustum[0]
    [1] + m_Frustum[0][2] * m_Frustum[0][2] ));
    m_Frustum[0][0] /= t;
    m_Frustum[0][1] /= t;
    m_Frustum[0][2] /= t;
    m_Frustum[0][3] /= t;

    m_Frustum[1][0] = clip[ 3] + clip[ 0]; // Získání levé roviny
    m_Frustum[1][1] = clip[ 7] + clip[ 4];
    m_Frustum[1][2] = clip[11] + clip[ 8];
    m_Frustum[1][3] = clip[15] + clip[12];
}
```

```

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[1][0] * m_Frustum[1][0] + m_Frustum[1][1] * m_Frustum[1][1] + m_Frustum[1][2] * m_Frustum[1][2] ));
m_Frustum[1][0] /= t;
m_Frustum[1][1] /= t;
m_Frustum[1][2] /= t;
m_Frustum[1][3] /= t;

m_Frustum[2][0] = clip[ 3] + clip[ 1]; // Získání dolní roviny
m_Frustum[2][1] = clip[ 7] + clip[ 5];
m_Frustum[2][2] = clip[11] + clip[ 9];
m_Frustum[2][3] = clip[15] + clip[13];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[2][0] * m_Frustum[2][0] + m_Frustum[2][1] * m_Frustum[2][1] + m_Frustum[2][2] * m_Frustum[2][2] ));
m_Frustum[2][0] /= t;
m_Frustum[2][1] /= t;
m_Frustum[2][2] /= t;
m_Frustum[2][3] /= t;

m_Frustum[3][0] = clip[ 3] - clip[ 1]; // Získání horní roviny
m_Frustum[3][1] = clip[ 7] - clip[ 5];
m_Frustum[3][2] = clip[11] - clip[ 9];
m_Frustum[3][3] = clip[15] - clip[13];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[3][0] * m_Frustum[3][0] + m_Frustum[3][1] * m_Frustum[3][1] + m_Frustum[3][2] * m_Frustum[3][2] ));
m_Frustum[3][0] /= t;
m_Frustum[3][1] /= t;
m_Frustum[3][2] /= t;
m_Frustum[3][3] /= t;

m_Frustum[4][0] = clip[ 3] - clip[ 2]; // Získání zadní roviny
m_Frustum[4][1] = clip[ 7] - clip[ 6];
m_Frustum[4][2] = clip[11] - clip[10];
m_Frustum[4][3] = clip[15] - clip[14];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[4][0] * m_Frustum[4][0] + m_Frustum[4][1] * m_Frustum[4][1] + m_Frustum[4][2] * m_Frustum[4][2] ));
m_Frustum[4][0] /= t;
m_Frustum[4][1] /= t;
m_Frustum[4][2] /= t;
m_Frustum[4][3] /= t;

m_Frustum[5][0] = clip[ 3] + clip[ 2]; // Získání přední roviny
m_Frustum[5][1] = clip[ 7] + clip[ 6];
m_Frustum[5][2] = clip[11] + clip[10];
m_Frustum[5][3] = clip[15] + clip[14];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[5][0] * m_Frustum[5][0] + m_Frustum[5][1] * m_Frustum[5][1] + m_Frustum[5][2] * m_Frustum[5][2] ));
m_Frustum[5][0] /= t;
m_Frustum[5][1] /= t;
m_Frustum[5][2] /= t;
m_Frustum[5][3] /= t;
}

```

Tato funkce byla opravdu náročná! Jsem si jistý, že už víte, proč vznikají nejrůznější OpenGL rozšíření. Ačkoli je matematika celkem přímočará, její strašná délka zobrazuje věci složitě. Použili jsme celkem 190 základních operací (násobení, dělení, sčítání, odčítání), plus šest druhých odmocnin. Protože ji budeme volat při každém překreslení scény, mohla by se snaha o optimalizaci vyplatit. Dokud nemodifikujeme projekční matici translací nebo rotací, můžeme používat její rychlejší ekvivalent `UpdateFrustumFaster()`.

```

void glCamera::UpdateFrustumFaster() // Získání ořezávacích rovin (optimalizovaná funkce)
{
    GLfloat clip[16]; // Pomocná matice

```

```

GLfloat proj[16]; // Projekční matice
GLfloat modl[16]; // Modelview matice
GLfloat t; // Pomocná

glGetFloatv(GL_PROJECTION_MATRIX, proj); // Získání projekční matice
glGetFloatv(GL_MODELVIEW_MATRIX, modl); // Získání modelview matice

// Vynásobí projekční matici pomocí modelview (nesmí být před tím použita rotace ani
translace)
clip[ 0] = modl[ 0] * proj[ 0];
clip[ 1] = modl[ 1] * proj[ 5];
clip[ 2] = modl[ 2] * proj[10] + modl[ 3] * proj[14];
clip[ 3] = modl[ 2] * proj[11];

clip[ 4] = modl[ 4] * proj[ 0];
clip[ 5] = modl[ 5] * proj[ 5];
clip[ 6] = modl[ 6] * proj[10] + modl[ 7] * proj[14];
clip[ 7] = modl[ 6] * proj[11];

clip[ 8] = modl[ 8] * proj[ 0];
clip[ 9] = modl[ 9] * proj[ 5];
clip[10] = modl[10] * proj[10] + modl[11] * proj[14];
clip[11] = modl[10] * proj[11];

clip[12] = modl[12] * proj[ 0];
clip[13] = modl[13] * proj[ 5];
clip[14] = modl[14] * proj[10] + modl[15] * proj[14];
clip[15] = modl[14] * proj[11];

m_Frustum[0][0] = clip[ 3] - clip[ 0]; // Získání pravé roviny
m_Frustum[0][1] = clip[ 7] - clip[ 4];
m_Frustum[0][2] = clip[11] - clip[ 8];
m_Frustum[0][3] = clip[15] - clip[12];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[0][0] * m_Frustum[0][0] + m_Frustum[0][1] * m_Frustum[0][1] + m_Frustum[0][2] * m_Frustum[0][2] ));
m_Frustum[0][0] /= t;
m_Frustum[0][1] /= t;
m_Frustum[0][2] /= t;
m_Frustum[0][3] /= t;

m_Frustum[1][0] = clip[ 3] + clip[ 0]; // Získání levé roviny
m_Frustum[1][1] = clip[ 7] + clip[ 4];
m_Frustum[1][2] = clip[11] + clip[ 8];
m_Frustum[1][3] = clip[15] + clip[12];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[1][0] * m_Frustum[1][0] + m_Frustum[1][1] * m_Frustum[1][1] + m_Frustum[1][2] * m_Frustum[1][2] ));
m_Frustum[1][0] /= t;
m_Frustum[1][1] /= t;
m_Frustum[1][2] /= t;
m_Frustum[1][3] /= t;

m_Frustum[2][0] = clip[ 3] + clip[ 1]; // Získání spodní roviny
m_Frustum[2][1] = clip[ 7] + clip[ 5];
m_Frustum[2][2] = clip[11] + clip[ 9];
m_Frustum[2][3] = clip[15] + clip[13];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[2][0] * m_Frustum[2][0] + m_Frustum[2][1] * m_Frustum[2][1] + m_Frustum[2][2] * m_Frustum[2][2] ));
m_Frustum[2][0] /= t;
m_Frustum[2][1] /= t;
m_Frustum[2][2] /= t;
m_Frustum[2][3] /= t;

m_Frustum[3][0] = clip[ 3] - clip[ 1]; // Získání horní roviny
m_Frustum[3][1] = clip[ 7] - clip[ 5];
m_Frustum[3][2] = clip[11] - clip[ 9];

```

```

m_Frustum[3][3] = clip[15] - clip[13];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[3][0] * m_Frustum[3][0] + m_Frustum[3][1] * m_Frustum[3][1] + m_Frustum[3][2] * m_Frustum[3][2] ));
m_Frustum[3][0] /= t;
m_Frustum[3][1] /= t;
m_Frustum[3][2] /= t;
m_Frustum[3][3] /= t;

m_Frustum[4][0] = clip[ 3] - clip[ 2]; // Získání zadní roviny
m_Frustum[4][1] = clip[ 7] - clip[ 6];
m_Frustum[4][2] = clip[11] - clip[10];
m_Frustum[4][3] = clip[15] - clip[14];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[4][0] * m_Frustum[4][0] + m_Frustum[4][1] * m_Frustum[4][1] + m_Frustum[4][2] * m_Frustum[4][2] ));
m_Frustum[4][0] /= t;
m_Frustum[4][1] /= t;
m_Frustum[4][2] /= t;
m_Frustum[4][3] /= t;

m_Frustum[5][0] = clip[ 3] + clip[ 2]; // Získání přední roviny
m_Frustum[5][1] = clip[ 7] + clip[ 6];
m_Frustum[5][2] = clip[11] + clip[10];
m_Frustum[5][3] = clip[15] + clip[14];

// Normalizace výsledku
t = GLfloat(sqrt( m_Frustum[5][0] * m_Frustum[5][0] + m_Frustum[5][1] * m_Frustum[5][1] + m_Frustum[5][2] * m_Frustum[5][2] ));
m_Frustum[5][0] /= t;
m_Frustum[5][1] /= t;
m_Frustum[5][2] /= t;
m_Frustum[5][3] /= t;
}

```

Operací se provádí stále mnoho, ale oproti předchozí verzi, je jich pouze něco přes polovinu (102). Optimalizace byla celkem jednoduchá, odstranil jsem pouze všechna násobení, která se díky nule vykrátí. Pokud chcete kompletní optimalizaci, použijte již zmíněná rozšíření, která za vás udělají stejnou práci a navíc mnohem rychleji, protože všechny výpočty proběhnou na hardwaru grafické karty. Ačkoli volání obou `UpdateFrustum()` funkcí navyšuje výkonnostní ztrátu, můžeme nyní snadno zjistit, jestli se libovolný bod nachází ve výhledu kamery. Obsahuje-li scéna více objektů náročných na rendering, bude určitě výhodné vykreslovat pouze ty, které půjdou vidět - například u rozsáhlého terénu.

`PointInFrustum()` vrácením `true` oznámí, že se bod předaný v parametru nachází ve viditelné oblasti okna. Druhá funkce je prakticky stejná, ale jedná se o kouli.

```

BOOL glCamera::PointInFrustum(glPoint p) // Bude bod vidět na scéně?
{
    int i;

    for(i = 0; i < 6; i++) // Bod se musí nacházet mezi všemi šesti ořezávacími rovinami
    {
        if(m_Frustum[i][0] * p.x + m_Frustum[i][1] * p.y + m_Frustum[i][2] * p.z +
           m_Frustum[i][3] <= 0)
        {
            return FALSE;
        }
    }

    return TRUE;
}

BOOL glCamera::SphereInFrustum(glPoint p, GLfloat Radius) // Bude koule vidět na scéně?
{
    int i;

    for(i = 0; i < 6; i++) // Koule se musí nacházet mezi všemi šesti ořezávacími rovinami
    {
        if(m_Frustum[i][0] * p.x + m_Frustum[i][1] * p.y + m_Frustum[i][2] * p.z +

```

```

        m_Frustum[i][3] <= -Radius)
    {
        return FALSE;
    }
}
return TRUE;
}

```

Ve funkci `IsOccluded()` požádáme `gluProject()` o zjištění, do které části viewportu bude zadaný bod projektován. Pozice ve viewportu odpovídá souřadnicím v depth bufferu. Pokud bude hloubka pixelu v bufferu menší než hloubka našeho bodu, je jasné, že se už něco nachází před ním.

```

bool glCamera::IsOccluded(glPoint p)// Je před bodem něco vykresleno?
{
    GLint viewport[4];// Data viewportu
    GLdouble mvmatrix[16], projmatrix[16];// Transformační matice
    GLdouble winx, winy, winz;// Výsledné souřadnice
    GLdouble flareZ;// Hloubka záře v obrazovce
    GLfloat bufferZ;// Hloubka z bufferu

    glGetIntegerv(GL_VIEWPORT, viewport);// Získání viewportu
    glGetDoublev(GL_MODELVIEW_MATRIX, mvmatrix);// Získání modelview matice
    glGetDoublev(GL_PROJECTION_MATRIX, projmatrix);// Získání projekční matice

    // Kam do viewportu (2D) se vykreslí bod (3D)
    gluProject(p.x, p.y, p.z, mvmatrix, projmatrix, viewport, &winx, &winy, &winz);
    flareZ = winz;

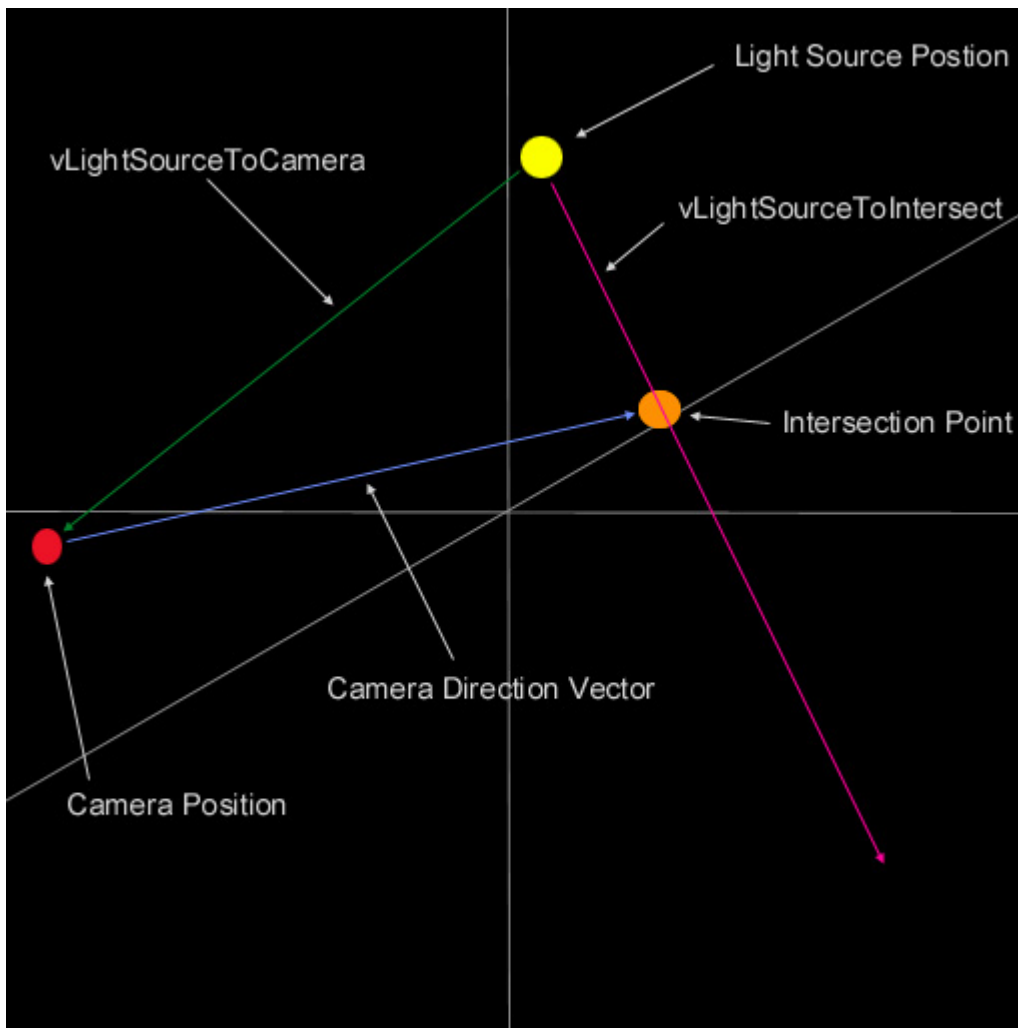
    glReadPixels(winx, winy, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &bufferZ);// Hloubka v
    depth bufferu

    if (bufferZ < flareZ)// Před bodem se nachází objekt
    {
        return true;
    }
    else// Nic před bodem není
    {
        return false;
    }
}

```

Všechny obdélníky objektů čočkového efektu by měly být vykreslovány na rovinu rovnoběžnou s obrazovkou monitoru, ale může se stát, že budou kvůli rotacím nakloněné. To je problém, protože by se měly zobrazit ploché i v případě, že se díváme na zdroj světla ze strany. Namísto otexturovaného quadu bychom mohli s výhodou využít point sprity. Když chceme nakreslit "klasický" obdélník, předáme OpenGL souřadnice čtyř bodů, texturovací koordináty a normálové vektory. Na rozdíl od toho point sprite vyžaduje pouze x, y, z souřadnice a nic jiného. Grafická karta vykreslí kolem těchto souřadnic obdélník, který bude vždy orientován k obrazovce. Možná se vám při programování částicových systémů stalo, že po natočení scény o 90 stupňů všechny částice zmizely, protože byly vykreslovány kolmo k ploše obrazovky. Právě pro ně se hodí point sprity nejvíce, ale pro čočkové efekty také. Jejich velká nevýhoda spočívá v implementaci, existují pouze jako rozšíření (`GL_NV_point_sprite`), takže se může stát, že je grafická karta nebude podporovat. Ani zde tedy rozšíření nepoužijeme. Řešení může spočívat v invertování všech rotací, nicméně problémy nastanou, pokud se kamera dostane za zdroj světla. Proto, abychom tomu předešli, budeme při pohybu kamerou zároveň měnit také polohu světla. Získáme i vedlejší efekt, zdroj světla se bude jevit jakoby stále ve stejné vzdálenosti a také dovolí čočkovým efektům o trochu vylepšit pohybování po přímé lince.

Vypočteme vzdálenost kamery od světla a přes směrový vektor kamery získáme průsečík, jehož vzdálenost od kamery musí být stejná jako vzdálenost kamery a světla. Máme-li průsečík, můžeme nalézt vektor, přes který vykreslíme všechny části čočkového efektu. Obrázek bude možná názornější...



```
void glCamera::RenderLensFlare() // Vykreslení čočkových objektů
{
    GLfloat Length = 0.0f;

    if(SphereInFrustum(m_LightSourcePos, 1.0f) == TRUE) // Pouze pokud kamera směřuje ke
    světlu
    {
        vLightSourceToCamera = m_Position - m_LightSourcePos; // Vektor od kamery ke
        světlu
        Length = vLightSourceToCamera.Magnitude(); // Vzdálenost kamery od světla

        ptIntersect = m_DirectionVector * Length; // Bod průsečíku
        ptIntersect += m_Position;

        vLightSourceToIntersect = ptIntersect - m_LightSourcePos; // Vektor mezi světlem
        a průsečíkem
        Length = vLightSourceToIntersect.Magnitude(); // Vzdálenost světla a průsečíku
        vLightSourceToIntersect.Normalize(); // Normalizace vektoru
    }
}
```

Na získaném směrovém vektoru vykreslíme záblesky. Posuneme se o x jednotek dolů po vektoru `vLightSourceToIntersect` a následným přičtením k pozici světla získáme nový požadovaný bod.

```
// Nastavení OpenGL
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
glDisable(GL_DEPTH_TEST);
glEnable(GL_TEXTURE_2D);

if (!IsOccluded(m_LightSourcePos)) // Před středem záře nesmí být žádný objekt
{
    // Vykreslení záře
    RenderBigGlow(0.60f, 0.60f, 0.8f, 1.0f, m_LightSourcePos, 16.0f);
}
```

```

RenderStreaks(0.60f, 0.60f, 0.8f, 1.0f, m_LightSourcePos, 16.0f);
RenderGlow(0.8f, 0.8f, 1.0f, 0.5f, m_LightSourcePos, 3.5f);

pt = vLightSourceToIntersect * (Length * 0.1f); // Bod ve 20% vzdálenosti od
světla ve směru průsečíku
pt += m_LightSourcePos;

RenderGlow(0.9f, 0.6f, 0.4f, 0.5f, pt, 0.6f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.15f); // Bod ve 30% vzdálenosti od
světla ve směru průsečíku
pt += m_LightSourcePos;

RenderHalo(0.8f, 0.5f, 0.6f, 0.5f, pt, 1.7f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.175f); // Bod ve 35% vzdálenosti
od světla ve směru průsečíku
pt += m_LightSourcePos;

RenderHalo(0.9f, 0.2f, 0.1f, 0.5f, pt, 0.83f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.285f); // Bod ve 57% vzdálenosti
od světla ve směru průsečíku
pt += m_LightSourcePos;

RenderHalo(0.7f, 0.7f, 0.4f, 0.5f, pt, 1.6f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.2755f); // Bod ve 55.1%
vzdálenosti od světla ve směru průsečíku
pt += m_LightSourcePos;

RenderGlow(0.9f, 0.9f, 0.2f, 0.5f, pt, 0.8f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.4775f); // Bod ve 95.5%
vzdálenosti od světla ve směru průsečíku
pt += m_LightSourcePos;

RenderGlow(0.93f, 0.82f, 0.73f, 0.5f, pt, 1.0f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.49f); // Bod ve 98% vzdálenosti od
světla ve směru průsečíku
pt += m_LightSourcePos;

RenderHalo(0.7f, 0.6f, 0.5f, 0.5f, pt, 1.4f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.65f); // Bod ve 130% vzdálenosti
od světla ve směru průsečíku
pt += m_LightSourcePos;

RenderGlow(0.7f, 0.8f, 0.3f, 0.5f, pt, 1.8f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.63f); // Bod ve 126% vzdálenosti
od světla ve směru průsečíku
pt += m_LightSourcePos;

RenderGlow(0.4f, 0.3f, 0.2f, 0.5f, pt, 1.4f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.8f); // Bod ve 160% vzdálenosti od
světla ve směru průsečíku
pt += m_LightSourcePos;

RenderHalo(0.7f, 0.5f, 0.5f, 0.5f, pt, 1.4f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 0.7825f); // Bod ve 156.5%
vzdálenosti od světla ve směru průsečíku
pt += m_LightSourcePos;

RenderGlow(0.8f, 0.5f, 0.1f, 0.5f, pt, 0.6f); // Vykreslení záře

pt = vLightSourceToIntersect * (Length * 1.0f); // Bod ve 200% vzdálenosti od
světla ve směru průsečíku
pt += m_LightSourcePos;

RenderHalo(0.5f, 0.5f, 0.7f, 0.5f, pt, 1.7f); // Vykreslení záře

```

```

        pt = vLightSourceToIntersect * (Length * 0.975f); // Bod ve 195% vzdálenosti
        od světla ve směru průsečíku
        pt += m_LightSourcePos;

        RenderGlow(0.4f, 0.1f, 0.9f, 0.5f, pt, 2.0f); // Vykreslení záře
    }

    // Obnovení nastavení OpenGL
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
    glDisable(GL_TEXTURE_2D);
}
}

```

Následuje výpis kódu pro rendering čočkové záře. Máme celkem čtyři různé funkce, které se ale liší pouze texturou objektu, jinak jsou identické.

```

void glCamera::RenderHalo(GLfloat r, GLfloat g, GLfloat b, GLfloat a, glPoint p, GLfloat
scale) // Vykreslení záře
{
    glPoint q[4]; // Pomocný bod

    q[0].x = (p.x - scale); // Výpočet pozice
    q[0].y = (p.y - scale);

    q[1].x = (p.x - scale);
    q[1].y = (p.y + scale);

    q[2].x = (p.x + scale);
    q[2].y = (p.y - scale);

    q[3].x = (p.x + scale);
    q[3].y = (p.y + scale);

    glPushMatrix(); // Uložení matice
    glTranslatef(p.x, p.y, p.z); // Přesun na pozici
    glRotatef(-m_HeadingDegrees, 0.0f, 1.0f, 0.0f); // Odstranění rotací
    glRotatef(-m_PitchDegrees, 1.0f, 0.0f, 0.0f);
    glBindTexture(GL_TEXTURE_2D, m_HaloTexture); // Textura
    glColor4f(r, g, b, a); // Nastavení barvy

    glBegin(GL_TRIANGLE_STRIP);
        glTexCoord2f(0.0f, 0.0f); glVertex2f(q[0].x, q[0].y);
        glTexCoord2f(0.0f, 1.0f); glVertex2f(q[1].x, q[1].y);
        glTexCoord2f(1.0f, 0.0f); glVertex2f(q[2].x, q[2].y);
        glTexCoord2f(1.0f, 1.0f); glVertex2f(q[3].x, q[3].y);
    glEnd();
    glPopMatrix(); // Obnovení matice
}

```

Tak to by z kódu bylo všechno. Pomocí kláves W, S, A, D můžete v programu měnit směr kamery. Klávesy 1 a 2 zapínají/vypínají výpisy informací. Z a C nastavují kameře konstantní rychlost a X ji zastavuje.

Samozřejmě nejsem první člověk, který vytvářel čočkové efekty, a proto můžete dole najít pár odkazů, které mi při psaní pomohly. Chtěl bych také poděkovat Davu Steerovi, Cameron Tidwell, Bertu Sammonsovi a Brannon Martidale za zpětnou vazbu a testování kódu na rozličném hardware.

- <http://www.gamedev.net/reference/articles/article874.asp>
- <http://www.gamedev.net/reference/articles/article813.asp>
- <http://www.opengl.org/developers/code/mjktips/lensflare/>
- <http://www.markmorley.com/opengl/frustumculling.html>
- http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt
- http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt

napsal: Vic Hollis <vichollis (zavináč) comcast.netVic>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Poznámky Daria Corna - rIO ze Spinning Kids

Přidal jsem několik testů pro zjištění objektů ve scéně před čočkovým efektem (na pozici zdroje světla). V takovém případě se záře vypíná. Nový kód by měl být dobře okomentován a je označen řetězcem # New Stuff #. Jeho případné odstranění by nemělo činit problémy. Modifikace jsou následující:

- Nová metoda třídy glCamera nazvaná IsOccluded(), která vrací true v případě, že se před světlem nachází nějaký objekt
- Několik proměnných pro gluCylinder (použit jako objekt stínící světlu)
- Změny v glDraw() pro vykreslení stínícího objektu
- Deinicializační kód pro quadratic

Doufám, že se vám modifikovaná verze bude líbit více. Jako domácí úkol si můžete zkusit testovat více než jeden bod na souřadnicích světla, aby se záře skokově nevypínala, ale postupně mizela.

Lekce 45 - Vertex Buffer Object (VBO)

Jeden z největších problémů jakékoli 3D aplikace je zajištění její rychlosti. Vždy byste měli limitovat množství aktuálně renderovaných polygonů buď řazením, cullingem nebo nějakým algoritmem na snižování detailů. Když nic z toho nepomáhá, můžete zkusit například vertex arrays. Moderní grafické karty nabízejí rozšíření nazvané vertex buffer object, které pracuje podobně jako vertex arrays kromě toho, že nahrává data do vysoce výkonné paměti grafické karty, a tak podstatně snižuje čas potřebný pro rendering. Samozřejmě ne všechny karty tato nová rozšíření podporují, takže musíme implementovat i verzi založenou na vertex arrays.

V tomto tutoriálu budeme

- nahrávat data výškové mapy
- používat vertex arrays k efektivnímu posílání dat vertexů do OpenGL
- prostřednictvím VBO nahrávat data do paměti grafické karty

Jako vždy nejdříve nadefinujeme parametry aplikace. První dvě symbolické konstanty představují rozlišení výškové mapy a měřítko pro vertikální roztáhnutí (viz. tutoriál 34 o výškových mapách). Když nadefinujete třetí konstantu, v programu se vypne používání VBO... abyste snadno mohli porovnat rychlostní rozdíl.

```
// Parametry výškové mapy
#define MESH_RESOLUTION 4.0f// Počet pixelů na vertex
#define MESH_HEIGHTSCALE 1.0f// Měřítko vyvýšení
// #define NO_VBOS// Vypíná VBO
```

K definicím také musíme přidat konstanty, datové typy a ukazatele na funkce pro VBO rozšíření. Zahrnul jsem jen parametry nutné pro toto demo. Pokud potřebujete více funkcionalitu, doporučuji z <http://www.opengl.org/> stáhnout nejnovější glx.h a použít definice obsažené v něm. Pro kód to jistě bude čistější metoda.

```
// Rozšíření VBO z glx.h
#define GL_ARRAY_BUFFER_ARB 0x8892
#define GL_STATIC_DRAW_ARB 0x88E4
typedef void (APIENTRY * PFNGLBINDBUFFERARBPROC) (GLenum target, GLuint buffer);
typedef void (APIENTRY * PFNGLDELETEBUFFERSARBPROC) (GLsizei n, const GLuint *buffers);
typedef void (APIENTRY * PFNGLGENBUFFERSARBPROC) (GLsizei n, GLuint *buffers);
typedef void (APIENTRY * PFNGLBUFFERDATAARBPROC) (GLenum target, int size, const GLvoid *data, GLenum usage);

// Ukazatele na funkce pro VBO
PFNGLGENBUFFERSARBPROC glGenBuffersARB = NULL;// Generování VBO jména
PFNGLBINDBUFFERARBPROC glBindBufferARB = NULL;// Zvolení VBO bufferu
PFNGLBUFFERDATAARBPROC glBufferDataARB = NULL;// Nahrávání dat VBO
PFNGLDELETEBUFFERSARBPROC glDeleteBuffersARB = NULL;// Mazání VBO
```

Deklarujeme jednoduché třídy vertexu a texturových koordinátů. CMesh je kompletní třídou, která může zapouzdřit základní data meshe. V našem případě se jedná o výškovou mapu. Kód vysvětluje sám sebe, všimněte si akorát, že data vertexů jsou oddělená od texturových koordinátů do vlastního pole. Jak bude vysvětleno dále, není to úplně nutné.

```
class CVert// Třída vertexu
{
public:
    float x;
    float y;
    float z;
};
typedef CVert CVec;// Definice jsou synonymní

class CTexCoord// Třída texturových koordinátů
{
public:
    float u;
    float v;
};
```

```

class CMesh// Třída meshe (výškové mapy)
{
public:
    int m_nVertexCount;// Počet vertexů
    CVert* m_pVertices;// Souřadnice vertexů
    CTexCoord* m_pTexCoords;// Texturové koordináty
    unsigned int m_nTextureId;// ID textury

    unsigned int m_nVBOVertices;// Jméno (ID) VBO pro vertexy
    unsigned int m_nVBOTexCoords;// Jméno (ID) VBO pro texturové koordináty

    AUX_RGBImageRec* m_pTextureImage;// Data výškové mapy

public:
    CMesh();// Konstruktor
    ~CMesh();// Destruktor

    bool LoadHeightmap(char* szPath, float flHeightScale, float flResolution);// Loading
    výškové mapy
    float PtHeight(int nX, int nY);// Hodnota na indexu výškové mapy
    void BuildVBOS();// Vytvoření VBO
};

```

Globální proměnná `g_bVBOSupported` indikuje podporu VBO ze strany grafické karty. Nastavíme ji v inicializačním kódu. `G_pMesh` bude ukládat data výškové mapy a `g_flyRot` určuje úhel natočení scény. Proměnná `g_nFPS` bude obsahovat počet snímků za sekundu a `g_nFrames` je čítač jednotlivých snímků. Poslední proměnná ukládá čas minulého výpočtu FPS.

```

bool g_fVBOSupported = false;// Flag podpory VBO
CMesh* g_pMesh = NULL;// Data meshe
float g_flyRot = 0.0f;// Rotace
int g_nFPS = 0, g_nFrames = 0;// FPS a čítač pro FPS
DWORD g_dwLastFPS = 0;// Čas minulého testu FPS

```

Funkce `Loadheightmap()` nahrává data výškové mapy. Pro ty z vás, kteří o ničem takovém ještě neslyšeli (Překl.: v originále - kdo žijete pod skálou :-). Výšková mapa je dvou dimenzionální sada dat, většinou obrázek, který hodnotami jednotlivých pixelů specifikuje vertikální výšku dané části terénu. Existuje mnoho různých způsobů, jak ji vytvořit. Moje implementace načítá tříkanalovou RGB bitmapu a ke zjištění výšky používá výpočet luminance. Výsledná hodnota bude díky tomu stejná pro barevný i černobílý obrázek. Osobně doporučuji čtyřkanalový formát vstupních dat, jako je například targa (.TGA) obrázek, u kterého alfa kanál může specifikovat výšku. Nicméně pro účely tohoto tutoriálu bude dostačovat obyčejná bitmapa.

Ujistíme se, že soubor obrázku existuje a pokud ano, loadujeme ho pomocí knihovny `glaux`. Víím, existují mnohem lepší cesty nahrávání obrázků...

```

bool CMesh::LoadHeightmap(char* szPath, float flHeightScale, float flResolution)
{
    FILE* fTest = fopen(szPath, "r");// Otevření pro čtení

    if (!fTest)
    {
        return false;
    }

    fclose(fTest);// Uvolní handle

    m_pTextureImage = auxDIBImageLoad(szPath);// Nahraje obrázek

```

Věci začínají být trochu zajímavější. Ze všeho nejdříve bych chtěl poukázat, že pro každý trojúhelník generuji tři vertexy - jednotlivé body nejsou sdílené. Měli byste to vědět už před načítáním.

Abychom mohli alokovat paměť pro data, potřebujeme znát její velikost. Výpočet je celkem jednoduchý ((šířka terénu / rozlišení) * (délka terénu / rozlišení) * 3 vertexy na trojúhelník * 2 trojúhelníky na čtverec). alokujeme paměť pro vertexy i texturové koordináty, deklarujeme pomocné proměnné a ve třech vnořených cyklech nastavíme obě pole.

```

// Generování pole vertexů
m_nVertexCount = (int)(m_pTextureImage->sizeX * m_pTextureImage->sizeY * 6 /
(flResolution * flResolution));

m_pVertices = new CVec[m_nVertexCount];// Alokace paměti
m_pTexCoords = new CTexCoord[m_nVertexCount];

int nX, nZ, nTri, nIndex = 0;// Pomocné

```

```

float flX, flZ;

for (nZ = 0; nZ < m_pTextureImage->sizeY; nZ += (int)flResolution)
{
    for (nX = 0; nX < m_pTextureImage->sizeX; nX += (int)flResolution)
    {
        for (nTri = 0; nTri < 6; nTri++)
        {
            // Výpočet x a z pozice bodu
            flX = (float)nX + ((nTri == 1 || nTri == 2 || nTri == 5) ?
            flResolution : 0.0f);
            flZ = (float)nZ + ((nTri == 2 || nTri == 4 || nTri == 5) ?
            flResolution : 0.0f);

            // Nastavení vertexu v poli
            m_pVertices[nIndex].x = flX - (m_pTextureImage->sizeX / 2);
            m_pVertices[nIndex].y = PtHeight((int)flX, (int)flZ) * flHeightScale;
            m_pVertices[nIndex].z = flZ - (m_pTextureImage->sizeY / 2);

            // Nastavení texturových koordinátů v poli
            m_pTexCoords[nIndex].u = flX / m_pTextureImage->sizeX;
            m_pTexCoords[nIndex].v = flZ / m_pTextureImage->sizeY;

            nIndex++; // Inkrementace indexu
        }
    }
}

```

Z obrázku výškové mapy vytvoříme OpenGL texturu a potom uvolníme jeho paměť.

```

glGenTextures(1, &m_nTextureId); // OpenGL ID
glBindTexture(GL_TEXTURE_2D, m_nTextureId); // Zvolí texturu
glTexImage2D(GL_TEXTURE_2D, 0, 3, m_pTextureImage->sizeX, m_pTextureImage->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, m_pTextureImage->data); // Nahraje texturu do OpenGL
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Lineární
filtrování
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

if (m_pTextureImage) // Uvolnění paměti
{
    if (m_pTextureImage->data)
    {
        free(m_pTextureImage->data);
    }

    free(m_pTextureImage);
}

return true;
}

```

Funkce PtHeight() vypočítá index do pole s daty, přitom ošetří přístup do nealokované paměti a vrátí výšku na daném indexu. Aby mohl být obrázek barevný i černobílý, použijeme vzorec pro luminanci. Opravdu nic složitého.

```

float CMesh::PtHeight(int nX, int nY) // Výška na indexu
{
    // Výpočet pozice v poli, ošetření přetečení
    int nPos = ((nX % m_pTextureImage->sizeX) + ((nY % m_pTextureImage->sizeY) *
    m_pTextureImage->sizeX)) * 3;

    float flR = (float)m_pTextureImage->data[nPos]; // Grabování složek barvy
    float flG = (float)m_pTextureImage->data[nPos + 1];
    float flB = (float)m_pTextureImage->data[nPos + 2];

    return (0.299f * flR + 0.587f * flG + 0.114f * flB); // Výpočet luminance
}

```

V následující funkci začneme konečně pracovat s vertex arrays a VBO. Takže, co to jsou pole vertexů? V základu je to systém, díky kterému můžeme ukázat OpenGL na pole geometrických dat a potom je několika málo příkazy vykreslit. Výsledkem je, že odpadají spousty výskytů funkcí typu glVertex3f() a jiných, které svým mnohonásobným voláním zbytečně zpomalují rendering. Systém vertex buffer object (VBO) jde ještě dále, namísto standardní paměti aplikace alokované v RAM používá vysoce výkonnou paměť grafické karty. Čas renderingu se zkracuje také proto, že data

nemusí putovat "po celém počítači", ale jsou uložena přímo na zařízení, kde se používají.

Takže teď se chystáme vytvořit Vertex Buffer Object. Pro tuto operaci existuje několik možných způsobů realizace, jeden z nich se nazývá "mapování" paměti. Myslím, že na tomto místě bude nejlepší jít tou nejsnadnější cestou. Nejprve pomocí `glGenBuffersARB()` získáme validní jméno VBO. Je to vlastně číslo ID, které OpenGL asociuje s našimi daty. Dále, podobně jako u textur, musíme VBO nastavit jako aktivní, čili říct OpenGL, že s ním chceme pracovat. K tomu slouží funkce `glBindBufferARB()`. Nakonec nahrajeme data do grafické karty. Funkci se předává velikost dat v bytech a ukazatel na ně. Protože už po této operaci nebudou potřeba, můžeme je smazat z RAM.

```
void CMesh::BuildVBOs()// Vytvoření VBO
{
    // VBO pro vertexy
    glGenBuffersARB(1, &m_nVBOVertices);// Získání jména (ID)
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, m_nVBOVertices);// Zvolení bufferu
    glBufferDataARB(GL_ARRAY_BUFFER_ARB, m_nVertexCount * 3 * sizeof(float),
        m_pVertices, GL_STATIC_DRAW_ARB);

    // VBO pro texturové koordináty
    glGenBuffersARB(1, &m_nVBOTexCoords);// Získání jména (ID)
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, m_nVBOTexCoords);// Zvolení bufferu
    glBufferDataARB(GL_ARRAY_BUFFER_ARB, m_nVertexCount * 2 * sizeof(float),
        m_pTexCoords, GL_STATIC_DRAW_ARB);

    // Data v RAM už jsou zbytečná
    delete [] m_pVertices;
    delete [] m_pTexCoords;
    m_pVertices = NULL;
    m_pTexCoords = NULL;
}
```

Tak to bychom měli, teď je čas na inicializaci. Vytvoříme dynamický objekt výškové mapy a pokusíme se ji vygenerovat ze souboru `terrain.bmp`. Není-li nadefinovaná symbolická konstanta `NO_VBOS`, zjistíme, jestli grafická karta podporuje rozšíření `GL_ARB_vertex_buffer_object`. Pokud ano, pomocí `wglGetProcAddress()` nagrabujeme ukazatele na potřebné funkce a vytvoříme VBO. Všimněte si, že se ve funkci `BuildVBOs()` mažou data výškové mapy, která se volá pouze, pokud je VBO podporováno.

```
BOOL Initialize(GL_Window* window, Keys* keys)// Inicializace
{
    g_window = window;
    g_keys = keys;

    g_pMesh = new CMesh();// Instance výškové mapy

    if(!g_pMesh->LoadHeightmap("terrain.bmp", MESH_HEIGHTSCALE, MESH_RESOLUTION))//
    Nahrání
    {
        MessageBox(NULL, "Error Loading Heightmap", "Error", MB_OK);
        return false;
    }

#ifdef NO_VBOS
    g_fVBOSupported = IsExtensionSupported("GL_ARB_vertex_buffer_object");// Test
    podpory VBO

    if(g_fVBOSupported)// Je rozšíření podporováno?
    {
        // Ukazatele na GL funkce
        glGenBuffersARB = (PFNGLGENBUFFERSARBPROC) wglGetProcAddress("glGenBuffersARB");
        glBindBufferARB = (PFNGLBINDBUFFERARBPROC) wglGetProcAddress("glBindBufferARB");
        glBufferDataARB = (PFNGLBUFFERDATAARBPROC) wglGetProcAddress("glBufferDataARB");
        glDeleteBuffersARB = (PFNGLDELETEBUFFERSARBPROC) wglGetProcAddress
            ("glDeleteBuffersARB");

        g_pMesh->BuildVBOs();// Poslat data vertexů do paměti grafické karty
    }
#else
    g_fVBOSupported = false;// Bez VBO
#endif

    // Klasické nastavení OpenGL
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
```



```

glClearDepth(1.0f);
glDepthFunc(GL_EQUAL);
glEnable(GL_DEPTH_TEST);
glShadeModel(GL_SMOOTH);
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
glEnable(GL_TEXTURE_2D);
glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

return TRUE;// Inicializace úspěšná
}

```

Funkci `IsExtensionSupported()`, která zjišťuje podporu rozšíření, můžete získat na OpenGL.org, ale moje varianta je o trochu čistší. Někteří lidé sice pomocí `strstr()` hledají pouze přítomnost podřetězce v řetězci, nicméně zdá se, že OpenGL.org moc nedůvěřuje konzistentnosti řetězce s rozšířeními.

```

bool IsExtensionSupported(char* szTargetExtension)// Je rozšíření podporováno?
{
    const unsigned char *pszExtensions = NULL;
    const unsigned char *pszStart;
    unsigned char *pszWhere, *pszTerminator;

    // Jméno by nemělo mít mezery
    pszWhere = (unsigned char *)strchr(szTargetExtension, ' ');

    if (pszWhere || *szTargetExtension == '\\0')
    {
        return false;// Nepodporováno
    }

    pszExtensions = glGetString(GL_EXTENSIONS);// Řetězec s názvy rozšíření
    // Vyhledávání podřetězce se jménem rozšíření
    pszStart = pszExtensions;

    for (;;)
    {
        pszWhere = (unsigned char *) strstr((const char *) pszStart, szTargetExtension);

        if (!pszWhere)
        {
            break;
        }

        pszTerminator = pszWhere + strlen(szTargetExtension);

        if (pszWhere == pszStart || *(pszWhere - 1) == ' ')
        {
            if (*pszTerminator == ' ' || *pszTerminator == '\\0')
            {
                return true;// Podporováno
            }
        }

        pszStart = pszTerminator;
    }

    return false;// Nepodporováno
}

```

Většina věcí je už hotová, zbývá vykreslování.

```

void Draw(void)// Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

```

Existuje několik možností, jak získat FPS. Asi nejjednodušší je čítat po dobu jedné sekundy průchody vykreslovací funkcí.

```

// Získání FPS
if(GetTickCount() - g_dwLastFPS >= 1000)// Uběhla sekunda?
{
    g_dwLastFPS = GetTickCount();// Aktualizace času pro další měření

```

```

g_nFPS = g_nFrames;// Uložení FPS
g_nFrames = 0;// Reset čítače

char szTitle[256] = {0};// Řetězec titulků okna
sprintf(szTitle, "Lesson 45: NeHe & Paul Frazee's VBO Tut - %d Triangles, %d
FPS", g_pMesh->m_nVertexCount / 3, g_nFPS);

if(g_fVBOSupported)// Používá/nepoužívá VBO
{
    strcat(szTitle, ", Using VBOs");
}
else
{
    strcat(szTitle, ", Not Using VBOs");
}

SetWindowText(g_window->hWnd, szTitle);// Nastaví titulek
}

g_nFrames++;// Inkrementace čítače FPS

```

Přesuneme kameru nad terén a natočíme scénu okolo osy y. Proměnnou g_flyRot inkrementujeme ve funkci Update().

```

glTranslatef(0.0f, -220.0f, 0.0f);// Přesun nad terén
glRotatef(10.0f, 1.0f, 0.0f, 0.0f);// Naklonění kamery
glRotatef(g_flyRot, 0.0f, 1.0f, 0.0f);// Rotace kamery

```

Abychom mohli pracovat s vertex arrays (a také VBO), musíme zapnout GL_VERTEX_ARRAY a GL_TEXTURE_COORD_ARRAY. Protože máme pouze jednu výškovou mapu, nemuseli bychom to dělat po každé, ale bývá to dobrým zvykem.

```

glEnableClientState(GL_VERTEX_ARRAY);// Zapne vertex arrays
glEnableClientState(GL_TEXTURE_COORD_ARRAY);// Zapne texture coord arrays

```

Dále musíme specifikovat pole, ve kterých má OpenGL hledat data. Začnu nejprve vertex arrays (část else), protože jsou jednodušší. Vše, co potřebujeme udělat, je zavolání funkce glVertexPointer(), které se předává počet prvků na jeden vertex (2, 3 nebo 4), typ dat, prokládání (v případě, že nejsou vertexy v samostatné struktuře) a ukazatel na pole. To samé platí i pro texturové koordináty, ale mají svoji vlastní funkci. Také bychom mohli uložit všechna data do jednoho velkého paměťového bufferu a použít glInterleavedArrays(), ale necháme je oddělené, abyste viděli, jak použít více VBO najednou.

Jediný rozdíl mezi vertex arrays a VBO je na tomto místě pouze v tom, že u VBO zavoláme glBindBufferARB() a do glVertexPointer() předáme místo ukazatele hodnotu NULL.

```

if(g_fVBOSupported)// Podporuje grafická karta VBO?
{
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, g_pMesh->m_nVBOVertices);
    glVertexPointer(3, GL_FLOAT, 0, (char *) NULL);// Předat NULL
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, g_pMesh->m_nVBOTexCoords);
    glTexCoordPointer(2, GL_FLOAT, 0, (char *) NULL);// Předat NULL
}
else// Obyčejné vertex arrays
{
    glVertexPointer(3, GL_FLOAT, 0, g_pMesh->m_pVertices);// Ukazatel na data
    vertexů
    glTexCoordPointer(2, GL_FLOAT, 0, g_pMesh->m_pTexCoords);// Ukazatel na
    texturové koordináty
}

```

Samotný rendering je ještě snazší. Pomocí glDrawArrays() řekneme OpenGL, aby vykreslil trojúhelníky ve formátu GL_TRIANGLES. Jako počáteční index v poli předáme nulu, celkový počet vertexů by měl být jasný. Funkce pomocí client state sama detekuje, co všechno má při renderingu použít (textury, světlo...). Existuje mnohem více způsobů, jak poslat data OpenGL. Jako příklad uvedu glVertexElement(), ale naše verze je ze všech nejrychlejší. Všimněte si také, že nespecifikujeme žádné glBegin() a glEnd(). Zde nejsou nutné.

Funkce glDrawArrays() je také důvodem, proč jsem zvolil nesdílet jeden vertex mezi několika trojúhelníky - není to možné. Co vím, nejlepší cestou, jak optimalizovat paměťové nároky, je použít triangle strip. V případě světel byste měli zajistit, aby měl k sobě každý vertex odpovídající normálový vektor. Je to sice nutnost, bez které by tato funkce nefungovala, na druhou stranu se však obrovsky zlepšil vzhled renderovaného objektu.

```

glDrawArrays(GL_TRIANGLES, 0, g_pMesh->m_nVertexCount);// Vykreslení vertexů

```

Zbývá vypnout client state a máme hotovo.

```
glDisableClientState(GL_VERTEX_ARRAY); // Vypne vertex arrays
glDisableClientState(GL_TEXTURE_COORD_ARRAY); // Vypne texture coord arrays
}
```

Pokud byste chtěli získat více informací o vertex buffer object, doporučuji prostudovat si dokumentaci ve SGI registru rozšíření (SGI's extensions registry) na <http://oss.sgi.com/projects/ogl-sample/registry/> . Je to sice trochu těžší čtení než tento tutoriál, ale budete znát mnohem více možností implementace a detailů.

napsal: Paul Frazee <paulfrazee (zavináč) cox.net>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 46 - Fullscreenový antialiasing

Chtěli byste, aby vaše aplikace vypadaly ještě lépe než doposud? Fullscreenové vyhlazování, nazývané též multisampling, by vám mohlo pomoci. S výhodou ho používají ne-realtimové renderovací programy, nicméně s dnešním hardwarem ho můžeme dosáhnout i v reálném čase. Bohužel je implementováno pouze jako rozšíření ARB_MULTISAMPLE, které nebude pracovat, pokud ho grafická karta nepodporuje.

V tomto zajímavém tutoriálu zkusíme posunout grafický vzhled aplikací ještě dále. O antialiasingu jste už četli v minulých tutoriálech, multisampling, narozdíl od něj, neoperuje s jednotlivými objekty zvlášť, ale pracuje až s vykreslovanými pixely. Ve výsledném obrázku se pokouší najít a odstranit ostré hrany. Protože se musí vzít v úvahu každý zobrazovaný pixel, bez hardwarové akcelerace grafické karty by velice snížil výkon aplikace.

```
Vid_mem = sizeof(Front_buffer) + sizeof(Back_buffer) + num_samples * (sizeof
(Front_buffer) + sizeof(ZS_buffer))
```

Pro více informací prosím zkuste tyto odkazy:

GDC2002 - OpenGL Multisample OpenGL Pixel Formats and Multisample Antialiasing

Po tomto nutném úvodu se konečně můžeme pustit do práce. Narozdíl od jiných rozšíření, která OpenGL při renderingu využívá, musíme s ARB_MULTISAMPLE počítat už při vytváření okna. Postupujeme tedy následovně:

- Vytvoříme okno úplně stejně jako obyčejně
- Dotážeme se, jestli můžeme vyhlazovat pixely
- Pokud je multisampling dostupný, zrušíme okno a vytvoříme ho s novým pixel formátem
- Pro části, které chceme vyhlazovat, jednoduše zavoláme glEnable(GL_ARB_MULTISAMPLE)

Začneme v souboru arb_multisample.cpp. Jako vždy inkludujeme hlavičkové soubory pro OpenGL a knihovnu GLU. O arb_multisample.h se budeme bavit později.

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>

#include "arb_multisample.h"
```

Symbolické konstanty použijeme při definování atributů pixel formátu. Podporuje-li grafická karta multisampling, bude logická proměnná arbMultisampleSupported obsahovat true.

```
#define WGL_SAMPLE_BUFFERS_ARB 0x2041// Symbolické konstanty pro multisampling
#define WGL_SAMPLES_ARB 0x2042
```

```
bool arbMultisampleSupported = false;// Je multisampling dostupný?
int arbMultisampleFormat = 0;// Formát multisamplingu
```

Následující funkce testuje, zda je WGL OpenGL rozšíření na systému dostupné v daném formátu.

```
bool WGLisExtensionSupported(const char *extension)// Je rozšíření podporováno?
{
    const size_t extlen = strlen(extension);
    const char *supported = NULL;

    // Pokud je to možné, pokusí se wglGetExtensionStringARB použít na aktuální DC
    PROC wglGetExtString = wglGetProcAddress("wglGetExtensionsStringARB");

    if (wglGetExtString)// WGL OpenGL rozšíření
    {
        supported = ((char*(__stdcall*)(HDC))wglGetExtString)(wglGetCurrentDC());
    }

    if (supported == NULL)// Zkusí ještě standardní OpenGL řetězec s rozšířeními
    {
        supported = (char*)glGetString(GL_EXTENSIONS);
    }
}
```

```

}

if (supported == NULL)// Pokud selže i toto, není řetězec dostupný
{
    return false;
}

for (const char* p = supported; ; p++)// Testování obsahu řetězce
{
    p = strstr(p, extension);// Hledá podřetězec

    if (p == NULL)// Podřetězec není v řetězci
    {
        return false;// Rozšíření nebylo nalezeno
    }

    // Okolo podřetězce se musí vyskytovat oddělovač (mezera nebo NULL)
    if ((p == supported || p[-1] == ' ') && (p[extlen] == '\0' || p[extlen] == ' '))
    {
        return true;// Rozšíření bylo nalezeno
    }
}
}

```

Funkce `InitMultisample()` je svým způsobem jádrem programu. Dotážeme se na podporu potřebného rozšíření a pokud ji máme, získáme požadovaný pixel formát.

```

bool InitMultisample(HINSTANCE hInstance, HWND hWnd, PIXELFORMATDESCRIPTOR pfd)//
Inicializace multisamplingu
{
    if (!WGLisExtensionSupported("WGL_ARB_multisample"))// Existuje řetězec ve WGL
    {
        arbMultisampleSupported = false;
        return false;
    }

    PFNWGLCHOOSEPIXELFORMATARBPROC wglChoosePixelFormatARB =
    (PFNWGLCHOOSEPIXELFORMATARBPROC)wglGetProcAddress("wglChoosePixelFormatARB");//
    Získání pixel formátu

    if (!wglChoosePixelFormatARB)// Daný pixel formát není dostupný
    {
        arbMultisampleSupported = false;
        return false;
    }

    HDC hdc = GetDC(hWnd);// Získání kontextu zařízení

    int pixelFormat;
    int valid;
    UINT numFormats;
    float fAttributes[] = {0, 0};

```

Následující pole atributů slouží pro definování vlastností pixel formátu. Všechny položky kromě `WGL_SAMPLE_BUFFERS_ARB` a `WGL_SAMPLE_ARB` jsou standardní, a proto by nám neměly činit potíže. Pokud uspěje hlavní test podpory multisamplingu, který reprezentuje `wglChoosePixelFormatARB()`, máme vyhráno.

```

int iAttributes[] =// Atributy
{
    WGL_DRAW_TO_WINDOW_ARB, GL_TRUE,
    WGL_SUPPORT_OPENGL_ARB, GL_TRUE,
    WGL_ACCELERATION_ARB, WGL_FULL_ACCELERATION_ARB,
    WGL_COLOR_BITS_ARB, 24,
    WGL_ALPHA_BITS_ARB, 8,
    WGL_DEPTH_BITS_ARB, 16,
    WGL_STENCIL_BITS_ARB, 0,
    WGL_DOUBLE_BUFFER_ARB, GL_TRUE,
    WGL_SAMPLE_BUFFERS_ARB, GL_TRUE,
    WGL_SAMPLES_ARB, 4,
    0, 0
};

```

```

valid = wglChoosePixelFormatARB(hdc, iAttributes, fAttributes, 1, &pixelFormat,
&numFormats); // Pixel formát pro čtyři vzorkování

if (valid && numFormats >= 1) // Vráceno true a počet formátů je větší než jedna
{
    arbMultisampleSupported = true;
    arbMultisampleFormat = pixelFormat;
    return arbMultisampleSupported;
}

iAttributes[19] = 2; // Čtyři vzorkování nejsou dostupná, test dvou

valid = wglChoosePixelFormatARB(hdc, iAttributes, fAttributes, 1, &pixelFormat,
&numFormats);

if (valid && numFormats >= 1)
{
    arbMultisampleSupported = true;
    arbMultisampleFormat = pixelFormat;
    return arbMultisampleSupported;
}

return arbMultisampleSupported; // Vrácení validního formátu
}

```

Kód pro detekci multisamplingu máme hotov, teď modifikujeme vytváření okna. Inkludujeme hlavičkový soubor `arb_multisample.h` a vytvoříme funkční prototypy.

```

#include "arb_multisample.h" // Hlavičkový soubor pro multisampling

BOOL DestroyWindowGL(GL_Window* window); // Funkční prototypy
BOOL CreateWindowGL(GL_Window* window);

```

Následující výpis kódu patří do funkce `CreateWindowGL()`. Původní kód povětšinou zůstane, ale uděláme v něm několik změn. V základu potřebujeme vyřešit problém, který spočívá v tom, že nemůžeme položit dotaz na pixel formát (detekovat přítomnost multisamplingu), dokud není vytvořeno okno. Nicméně naproti tomu nemůžeme vytvořit okno s vyhlazováním, dokud nemáme pixel formát, který ho podporuje. Trochu se to podobá otázce, zda bylo první vejce nebo slepice. Implementujeme dvouprůchodový systém - nejprve vytvoříme obyčejné okno, dotážeme se na pixel formát a pokud je multisampling podporován, zrušíme okno a vytvoříme správné. Trochu těžkopádné, ale neznám jiný způsob.

```

// Funkce CreateWindowGL()
window->hDC = GetDC(window->hWnd); // Grabování kontextu zařízení

if (window->hDC == 0) // Podařilo se ho získat?
{
    DestroyWindow(window->hWnd); // Zrušení okna
    window->hWnd = 0; // Nulování handle

    return FALSE; // Neúspěch
}

```

Při prvním průchodu touto funkcí (další průchody např. při přepínání do/z fullscreenu) není možné multisampling natvrdo zapnout, takže jsme vytvořili pouze obyčejné okno. Pokud máme jistotu, že ho můžeme použít, nastavíme pixel formát na `arbMultiSampleFormat`.

```

if(!arbMultisampleSupported) // Multisampling není podporován
{
    // Vytvoření normálního okna
    PixelFormat = ChoosePixelFormat(window->hDC, &pf); // Získá kompatibilní pixel
    formát

    if (PixelFormat == 0) // Podařilo se ho získat?
    {
        ReleaseDC(window->hWnd, window->hDC); // Uvolnění kontextu zařízení
        window->hDC = 0; // Nulování proměnné
        DestroyWindow(window->hWnd); // Zrušení okna
        window->hWnd = 0; // Nulování handle

        return FALSE; // Neúspěch
    }
}
else // Multisampling je podporován

```

```

{
    PixelFormat = arbMultisampleFormat;
}

if (SetPixelFormat(window->hDC, PixelFormat, &pfid) == FALSE) // Zkusí nastavit pixel
formát
{
    ReleaseDC(window->hWnd, window->hDC);
    window->hDC = 0;
    DestroyWindow(window->hWnd);
    window->hWnd = 0;

    return FALSE;
}

window->hRC = wglCreateContext(window->hDC); // Zkusí získat rendering kontext

if (window->hRC == 0) // Podařilo se ho získat?
{
    ReleaseDC(window->hWnd, window->hDC);
    window->hDC = 0;
    DestroyWindow(window->hWnd);
    window->hWnd = 0;

    return FALSE;
}

if (wglMakeCurrent(window->hDC, window->hRC) == FALSE) // Aktivuje rendering kontext
{
    wglDeleteContext(window->hRC);
    window->hRC = 0;
    ReleaseDC(window->hWnd, window->hDC);
    window->hDC = 0;
    DestroyWindow(window->hWnd);
    window->hWnd = 0;

    return FALSE;
}

```

Okno bylo vytvořeno, takže máme k dispozici handle pro dotaz na multisampling. Pokud je podporován, zrušíme okno a vytvoříme ho s novým pixel formátem.

```

if(!arbMultisampleSupported && CHECK_FOR_MULTISAMPLE) // Je multisampling dostupný?
{
    if(InitMultisample(window->init.application->hInstance, window->hWnd, pfd)) //
Inicializace multisamplingu
    {
        DestroyWindowGL(window);
        return CreateWindowGL(window);
    }
}

ShowWindow(window->hWnd, SW_NORMAL); // Zobrazí okno
window->isVisible = TRUE;

ReshapeGL(window->init.width, window->init.height); // Oznámí rozměry okna OpenGL
ZeroMemory(window->keys, sizeof(Keys)); // Nulování pole indikující stisk kláves
window->lastTickCount = GetTickCount(); // Inicializuje časovou proměnnou

return TRUE; // Vše v pořádku
}

```

OK, nastavování je kompletní, dostáváme se k zábavnější části, pro kterou jsme se tak snažili. Naštěstí se sdružení ARB rozhodlo učinit multisampling dynamickým, což nám ho umožňuje kdykoli zapnout nebo vypnout. Stačí jednoduché glEnable() a glDisable().

```

glEnable(GL_MULTISAMPLE_ARB);
// Vykreslení vyhlazovaných objektů
glDisable(GL_MULTISAMPLE_ARB);

```

A to je vše. Až spustíte ukázkové demo, uvidíte, jak kvalitně vyhlazování zlepšuje celkový vzhled scény.

napsal: Colt McAnlis - MainRoach <duhroach (zavináč) hotmail.com>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 47 - CG vertex shader

Používání vertex a fragment (pixel) shaderů ke "špinavé práci" při renderingu může mít nespočet výhod. Nejvíce je vidět např. pohyb objektů do teď výhradně závislý na CPU, který neběží na CPU, ale na GPU. Pro psaní velice kvalitních shaderů poskytuje CG (přiměřeně) snadné rozhraní. Tento tutoriál vám ukáže jednoduchý vertex shader, který sice něco dělá, ale nebude předvádět ne nezbytné osvětlení a podobné složitější nadstavby. Tak jako tak je především určen pro začátečníky, kteří už mají nějaké zkušenosti s OpenGL a zajímají se o CG.

Hned na začátku uvedu dvě internetové adresy, které by se vám mohli hodit. Jedná se o <http://developer.nvidia.com/> a <http://www.cgshaders.org/>.

Překl.: Perfektní článek o vertex a pixel shaderech vyšel v časopise CHIP 01/2004: Hardwarový Fotorealismus - Možnosti moderních 3D grafických akceleratorů (str. 96 - 100).

Poznámka: účelem tohoto tutoriálu není naučit úplně všechno o psaní vertex shaderů používajících CG. Má v úmyslu vysvětlit, jak úspěšně nahrát a spustit vertex shader v OpenGL.

Nastavení

První krok spočívá v downloadu CG kompilátoru od nVidie. Protože existují rozdíly mezi verzemi 1.0 a 1.1, dbejte na to, abyste si stáhli ten novější. Kód přeložený pro jeden nemusí pracovat i s druhým. Rozdíly jsou např. v rozdílně pojmenovaných proměnných, nahrazených funkcích a podobně.

Dále musíme nahrát hlavičkové a knihovní soubory CG na místo, kde je může Visual Studio najít. Protože ze zásady nedůvěřuji instalátorům, které povětšinou pracují jinak, než se očekává, osobně dávám přednost ručnímu kopírování knihovních souborů

```
z: C:\Program Files\NVIDIA Corporation\Cg\lib
do: C:\Program Files\Microsoft Visual Studio\VC98\Lib
```

a hlavičkových souborů

```
z: C:\Program Files\NVIDIA Corporation\Cg\include
do: C:\Program Files\Microsoft Visual Studio\VC98\Include
```

CG Tutoriál

Informace o CG uvedené v tomto tutoriálu byly většinou získány z CG uživatelského manuálu (CG Toolkit User's Manual).

Existuje několik podstatných bodů, které byste si měli provždy zapamatovat. První a nejdůležitější je, že se vertex program provede na KAŽDÉM vertexu, který předáte grafické kartě. Jediná možnost, jak ho spustit nad několika zvolenými vertexy je buď ho nahrávat/mazat individuálně pro každý vertex nebo posílat vertexy do proudu, ve kterém budou ovlivněny a do proudu, kde nebudou. Výstup vertex programu je předán fragment (pixel) shaderu. To platí pouze tehdy, pokud je implementován a zapnut. Za poslední si zapamatujte, že se vertex program provede nad vertexy předtím, než se vytvoří primitiva. Fragment shader je na rozdíl od toho vykonán až po rasterizaci.

Pojďme se konečně podívat na tutoriál. Vytvoříme prázdný textový soubor a pojmenujeme ho wave.cg. Do něj budeme psát veškerý CG kód. Nejdříve vytvoříme datové struktury, které budou obsahovat všechny proměnné a informace potřebné pro shader.

Každá ze všech tří proměnných struktury (pozice, barva a hodnota vlny) je následována předdefinovaným jménem (POSITION, COLOR0, COLOR1). Tato předdefinovaná jména se vztahují k sémantice jazyka. Specifikují mapování vstupů do přesně určených hardwarových registrů. Mimochodem, jedinou opravdu požadovanou vstupní proměnnou do vertex programu je position.

```
struct appdata
{
    float4 position : POSITION;
    float4 color: COLOR0;
    float3 wave: COLOR1;
};
```

Dále vytvoříme strukturu vfconn. Ta bude obsahovat výstup vertex programu, který se po rasterizaci předá fragment shaderu. Stejně jako vstupy mají i výstupy předdefinovaná jména. HPos reprezentuje pozici transformovanou do

homogenního souřadnicového systému a Col0 určuje barvu vertexu změněnou v programu.

```
struct vfconn
{
    float4 HPos: POSITION;
    float4 Col0: COLOR0;
};
```

Zbývá nám pouze napsat vertex program. Funkce se definuje stejně jako v jazyce C. Má návratový typ (struktura vfconn), jméno (main, ale může jím být i jakékoli jiné) a parametry. V našem příkladě ze vstupu převezmeme strukturu appdata, která obsahuje pozici vertexu, jeho barvu a hodnotu výšky pro vytvoření sinusových vln. Dostaneme také uniformní parametr, kterým je aktuální modelview matice. Potřebujeme ji pro transformaci pozice do homogenního souřadnicového systému.

```
vfconn main(appdata IN, uniform float4x4 ModelViewProj)
{
```

Do proměnné OUT uložíme modifikované vstupní parametry a na konci programu ji vrátíme.

```
    vfconn OUT;// Výstup z vertex shaderu (posílá se na fragment shader, pokud je
    dostupný)
```

Vypočítáme pozici na ose y v závislosti na x a z pozici vertexu. X i z vydělíme pěti (respektive čtyřmi), přechody budou jemnější. Změňte hodnoty na 1.0, abyste viděli, co myslím. Proměnná IN.wave specifikovaná hlavním programem obsahuje stále se zvětšující hodnotu, která způsobí, že se sinusová vlna rozpohybuje přes celý mesh. Y pozici spočítáme z pozice v meshi jako sinus hodnoty vlny plus aktuální x nebo z pozice. Aby byla výsledná vlna vyšší, vynásobíme ještě výsledek číslem 2,5.

```
    // Změna y pozice v závislosti na sinusové vlně
    IN.position.y = (sin(IN.wave.x + (IN.position.x / 5.0)) + sin(IN.wave.x +
    (IN.position.z / 4.0))) * 2.5f;
```

Nastavíme výstupní proměnné našeho vertex programu. Nejdříve transformujeme novou pozici vertexu do homogenního souřadnicového systému a potom přiřadíme výstupní barvě hodnotu vstupní. Pomocí return předáme vše fragment shaderu (pokud je zapnutý).

```
    OUT.HPos = mul(ModelViewProj, IN.position);// Transformace pozice na homogenní
    souřadnice
    OUT.Col0.xyz = IN.color.xyz;// Nastavení barvy
    return OUT;
}
```

OpenGL Tutorial

V tuto chvíli máme vertex program běžící na grafické kartě hotov. Můžeme se pustit do hlavního programu. Vytvoříme v něm rovinný mesh poskládaný z trojúhelníků (triangle stripů), které budeme posílat na grafickou kartu. Na ní se ovlivní y pozice každého vertexu tak, aby ve výsledku vznikly pohyblivé se sinusové vlny.

V první řadě inkluujeme hlavičkové soubory, které v OpenGL umožní spustit CG shader. Musíme také říct Visual Studiu, aby přilinkovalo potřebné knihovny soubory.

```
#include <windows.h>// Windows
#include <gl\gl.h>// OpenGL
#include <gl\glu.h>// GLU

#include <cg\cg.h>// CG hlavičky
#include <cg\cggl.h>// CG hlavičky specifické pro OpenGL

#include "NeHeGL.h"// NeHe OpenGL

#pragma comment(lib, "opengl32.lib")// Přilinkování OpenGL
#pragma comment(lib, "glu32.lib")// Přilinkování GLU

#pragma comment(lib, "cg.lib")// Přilinkování CG
#pragma comment(lib, "cggl.lib")// Přilinkování OpenGL CG

#define TWO_PI 6.2831853071// PI * 2

GL_Window* g_window;// Struktura okna
Keys* g_keys;// Klávesnice
```

Symbolická konstanta SIZE určuje velikost meshe na osách x a z. Dále vytvoříme proměnnou cg_enable, která bude

oznamovat, jestli má být vertex program zapnutý nebo vypnutý. Pole mesh slouží pro uložení dat meshe a wave_movement pro vytvoření sinusové vlny.

```
#define SIZE 64// Velikost meshe

bool cg_enable = TRUE, sp;// Flag spuštění CG
GLfloat mesh[SIZE][SIZE][3];// Data meshe
GLfloat wave_movement = 0.0f;// Pro vytvoření sinusové vlny
```

Následují proměnné pro CG. CGcontext slouží jako kontejner pro několik CG programů. Obecně stačí pouze jeden CGcontext bez ohledu na počet vertex a fragment programů, které využíváme. Z jednoho kontextu můžete pomocí funkcí cgGetFirstProgram() a cgGetNextProgram() zvolit libovolný program. CG profile definuje profil vertexů. CG parametry zprostředkovávají vazbu mezi hlavním programem a CG programem běžícím na grafické kartě. Každý CG parameter je handle na korespondující proměnnou v shaderu.

```
CGcontext cgContext;// CG kontext
CGprogram cgProgram;// CG vertex program
CGprofile cgVertexProfile;// CG profil
CGparameter position, color, modelViewMatrix, wave;// Parametry pro shader
```

Deklaraci globálních proměnných máme za sebou, pojďme se podívat na inicializační funkci. Po obvyklých nastaveních zapneme vykreslování drátěných modelů. Používáme je z důvodu, že vyplněné polygony nevypadají bez světél dobře. Pomocí dvou vnořených cyklů inicializujeme pole mesh tak, aby se střed roviny nacházel v počátku souřadnicového systému. Pozici na ose y nastavíme u všech bodů na 0.0f, sinusovou deformaci má na starosti CG program.

```
BOOL Initialize(GL_Window* window, Keys* keys)// Inicializace
{
    g_window = window;// Okno
    g_keys = keys;// Klávesnice

    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Černé pozadí
    glClearDepth(1.0f);// Mazání hloubky
    glDepthFunc(GL_EQUAL);// Typ testování hloubky
    glEnable(GL_DEPTH_TEST);// Povolí testování hloubky
    glShadeModel(GL_SMOOTH);// Jemné stínování
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);// Nastavení perspektivy

    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);// Drátěný model

    for (int x = 0; x < SIZE; x++)// Inicializace meshe
    {
        for (int z = 0; z < SIZE; z++)
        {
            mesh[x][z][0] = (float) (SIZE / 2) - x;// Vycentrování na ose x
            mesh[x][z][1] = 0.0f;// Plochá rovina
            mesh[x][z][2] = (float) (SIZE / 2) - z;// Vycentrování na ose z
        }
    }
}
```

Musíme také inicializovat CG, jako první vytvoříme kontext. Pokud funkce vrátí NULL, něco selhalo, chyby většinou nastávají kvůli nepovedené alokaci paměti. Zobrazíme chybovou zprávu a vrátíme false, čímž ukončíme i celý program.

```
cgContext = cgCreateContext();// Vytvoření CG kontextu

if (cgContext == NULL)// OK?
{
    MessageBox(NULL, "Failed To Create Cg Context", "Error", MB_OK);
    return FALSE;
}
```

Pomocí cgGLGetLatestProfile() určíme minulý profil vertexů, za typ profilu předáme CG_GL_VERTEX. Kdybychom vytvářeli fragment shader, předávali bychom CG_GL_FRAGMENT. Pokud není žádný vhodný profil dostupný, vrátí funkce CG_PROFILE_UNKNOWN. S validním profilem můžeme zavolat cgGLSetOptimalOptions(). Tato funkce se používá pokaždé, když se překládá nový CG program, protože podstatně optimalizuje kompilaci shaderu v závislosti na aktuálním grafickém hardwaru a jeho ovladačích.

```
cgVertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX);// Získání minulého profilu
vertexů

if (cgVertexProfile == CG_PROFILE_UNKNOWN)// OK?
{
    MessageBox(NULL, "Invalid profile type", "Error", MB_OK);
}
```

```

        return FALSE;
    }

    cgGLSetOptimalOptions(cgVertexProfile); // Nastavení profilu

```

Zavoláme funkci `cgCreateProgramFromFile()`, čímž načteme a zkompilujeme CG program. První parametr specifikuje CG kontext, ke kterému bude program připojen. Druhý parametr určuje, že soubor obsahuje zdrojový kód (CG_SOURCE) a ne objektový kód předkompilovaného programu (CG_OBJECT). Jako třetí položka se předává cesta k souboru, čtvrtý je minulým profilem pro konkrétní typ programu (vertex profil pro vertex program, fragment profil pro fragment program). Pátý parametr specifikuje vstupní funkci do programu, její jméno může být libovolné, ne pouze `main` (). Poslední parametr slouží pro předání přídatných argumentů kompilátoru. Většinou se dává NULL.

Pokud z nějakého důvodu funkce selže, získáme pomocí `cgGetError()` typ chyby. Do řetězcové podoby ho můžeme převést prostřednictvím `cgGetErrorString()`.

```

// Nahraje a zkompiluje vertex shader
cgProgram = cgCreateProgramFromFile(cgContext, CG_SOURCE, "CG/Wave.cg",
cgVertexProfile, "main", 0);

if (cgProgram == NULL) // OK?
{
    CGError Error = cgGetError(); // Typ chyby

    MessageBox(NULL, cgGetErrorString(Error), "Error", MB_OK);
    return FALSE;
}

```

Nahrajeme zkompilovaný program a připravíme ho pro zvolení (binding).

```

cgGLLoadProgram(cgProgram); // Nahraje program do grafické karty

```

Jako poslední krok inicializace získáme handle na proměnné, se kterými bude CG program manipulovat. Pokud daná proměnná neexistuje, `cgGetNamedParameter()` vrátí NULL. Neznáme-li jména parametrů, můžeme použít dvojici funkcí `cgGetFirstParameter()` a `cgGetNextParameter()`.

```

// Handle na proměnné
position = cgGetNamedParameter(cgProgram, "IN.position");
color = cgGetNamedParameter(cgProgram, "IN.color");
wave = cgGetNamedParameter(cgProgram, "IN.wave");
modelViewMatrix = cgGetNamedParameter(cgProgram, "ModelViewProj");

return TRUE;
}

```

Pomocí deinicializační funkce po sobě uklidíme. Jednoduše zavoláme `cgDestroyContext()` pro každý CGcontext proměnnou. Také bychom mohli smazat jednotlivé CG programy, k tomu slouží funkce `cgDestroyProgram()`, nicméně `cgDestroyContext()` je smaže automaticky.

```

void Deinitialize(void) // Deinicializace
{
    cgDestroyContext(cgContext); // Smaže CG kontext
}

```

Do aktualizací funkce přidáme kód pro ošetření stisku mezerníku, který zapíná/vypíná CG program běžící na grafické kartě.

```

void Update(float milliseconds) // Aktualizace
{
    if (g_keys->keyDown[VK_ESCAPE]) // Stisk Esc
    {
        TerminateApplication(g_window); // Konec programu
    }

    if (g_keys->keyDown[VK_F1]) // Stisk F1
    {
        ToggleFullscreen(g_window); // Přepnutí do/z fullscreenu
    }

    if (g_keys->keyDown[' '] && !sp) // Stisk mezerníku
    {
        sp = TRUE;
        cg_enable = !cg_enable; // Zapne/vypne CG program
    }
}

```

```

    if (!g_keys->keyDown[' '])
    {
        sp = FALSE;
    }
}

```

A jako poslední vykreslování. Kamerou se přesuneme o 45 jednotek před počátek souřadnicového systému a nahoru o 25 jednotek.

```

void Draw(void) // Vykreslování
{
    glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Smaže obrazovku
    glLoadIdentity(); // Reset matice

    gluLookAt(0.0f, 25.0f, -45.0f, 0.0f, 0.0f, 0.0f, 0, 1, 0); // Pozice kamery

```

Modelview matici vertex shaderu nastavíme na aktuální OpenGL matici. Bez toho bychom nemohli přepočítávat pozici vertexů do homogenních souřadnic.

```

    // Nastavení modelview matice v shaderu
    cgGLSetStateMatrixParameter(modelViewMatrix, CG_GL_MODELVIEW_PROJECTION_MATRIX,
    CG_GL_MATRIX_IDENTITY);

```

Pokud je flag `cg_enable` v `true`, voláním `cgGLEnableProfile()` aktivujeme předaný profil. Funkce `cgGLBindProgram()` zvolí náš program a dokud ho nevypneme, provede se nad každým vertexem poslaným na grafickou kartu. Také musíme poslat barvu vertexů.

```

    if (cg_enable) // Zapnout CG shader?
    {
        cgGLEnableProfile(cgVertexProfile); // Zapne profil
        cgGLBindProgram(cgProgram); // Zvolí program
        cgGLSetParameter4f(color, 0.5f, 1.0f, 0.5f, 1.0f); // Nastaví barvu (světle
        zelená)
    }

```

Tak teď jsme konečně připraveni na rendering meshe. Pro každou hodnotu souřadnice `x` v cyklu vykreslíme proužek roviny seskládaný triangle stripem.

```

    for (int x = 0; x < SIZE - 1; x++) // Vykreslení meshe
    {
        glBegin(GL_TRIANGLE_STRIP); // Každý proužek jedním triangle stripem

        for (int z = 0; z < SIZE - 1; z++)
        {

```

Současně s renderovanými vertexy dynamicky předáme i hodnotu `wave` parametru, díky kterému bude moci CG program z roviny vygenerovat sinusové vlny. Jakmile grafická karta dostane všechna data, automaticky spustí CG program. Všimněte si, že do triangle stripu posíláme dva body, to má za následek, že se nevykreslí pouze trojúhelník, ale rovnou celý čtverec.

```

            cgGLSetParameter3f(wave, wave_movement, 1.0f, 1.0f); // Parametr vlny

            glVertex3f(mesh[x][z][0], mesh[x][z][1], mesh[x][z][2]); // Vertex
            glVertex3f(mesh[x+1][z][0], mesh[x+1][z][1], mesh[x+1][z][2]); // Vertex

            wave_movement += 0.00001f; // Inkrementace parametru vlny

            if (wave_movement > TWO_PI) // Větší než dvě pí (6,28)?
            {
                wave_movement = 0.0f; // Vynulovat
            }
        }

        glEnd(); // Konec triangle stripu
    }

```

Po dokončení renderingu otestujeme, jestli je `cg_enable` rovno `true` a pokud ano, vypneme vertex profil. Dále můžeme kreslit cokoli chceme, aniž by to bylo ovlivněno CG programem.

```

    if (cg_enable) // Zapnutý CG shader?
    {
        cgGLDisableProfile(cgVertexProfile); // Vypne profil
    }

```

```
} glFlush();// Vyprázdnění renderovací pipeline
```

napsal: Owen Bourne <o.bourne (zavináč) griffith.edu.au>
přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>

Lekce 48 - ArcBall rotace

Nebylo by skvělé otáčet modelem pomocí myši jednoduchým drag & drop? S ArcBall rotacemi je to možné. Moje implementace je založená na myšlenkách Brettona Wadea a Kena Shoemakea. Kód také obsahuje funkci pro rendering toroidu - kompletně i s normálami.

ArcBall funguje tak, že mapuje okenní souřadnice kliknutí přímo do souřadnic ArcBallu. Zmenší poměr souřadnic myši z rozsahu [0..šířka, 0..výška] na rozsah [-1..1, 1..-1]. Pamatujte si, že aby v OpenGL dosáhl korektního výsledku, musí převrátit znamínko y souřadnice. Vzorec vypadá takto:

```
MousePt.X = ((MousePt.X / ((Width - 1) / 2)) - 1);  
MousePt.Y = -((MousePt.Y / ((Height - 1) / 2)) - 1);
```

Jediný důvod, proč jsme měnili měřítko souřadnic je, abychom zjednodušili matematiku, nicméně šťastnou shodou okolností to dovoluje kompilátoru kód trochu optimalizovat. Dále vypočítáme délku vektoru a určíme, jestli se nachází nebo nenachází uvnitř koule. Pokud ano, vrátíme vektor z jejího vnitřku, jinak normalizujeme bod a vrátíme nejbližší pozici k vnějšku koule. Poté, co máme oba vektory, získáme vektor současně kolmý na počáteční i koncový vektor, čímž dostaneme quaternion. S tímto v rukách máme dost informací na vygenerování rotační matice.

Konstruktoru třídy ArcBall budeme předávat rozměry okna.

```
ArcBall_t::ArcBall_t(GLfloat NewWidth, GLfloat NewHeight)
```

Když uživatel klikne myší, vypočítáme počáteční vektor podle toho, kam kliknul.

```
void ArcBall_t::click(const Point2fT* NewPt)
```

Když táhne myší (drag), aktualizujeme koncový vektor pomocí metody drag() a pokud je poskytnut i výstupní quaternion, aktualizujeme ho pomocí výsledné rotace.

```
void ArcBall_t::drag(const Point2fT* NewPt, Quat4fT* NewRot)
```

Při změně velikosti okna jednoduše aktualizujeme i rozměry ArcBallu.

```
void ArcBall_t::setBounds(GLfloat NewWidth, GLfloat NewHeight)
```

V projektu budeme potřebovat i několik dalších proměnných. Transformation je finální transformace, která určuje rotaci, ale také posunutí. LastRot představuje poslední zaznamenanou rotaci od konce dragu a ThisRot určuje rotaci v době táhnutí myší. Všechny tři na začátku inicializujeme na matici identity.

Při kliknutí se začíná z identického stavu rotace a když následně táhneme, rotace se počítá od pozice kliknutí až po bod táhnutí. I když na otáčení objektů ve scéně používáme tuto implementaci, je důležité poznamenat, že nerotujeme samotný ArcBall. S rostoucími (přírůstkovými) rotacemi se musíme vypořádat sami. To je úkol LastRot a ThisRot. LastRot si můžeme představit jako všechny rotace až do teď a ThisRot jako aktuální rotace. Vždy, když začne rotace, ThisRot se modifikuje pomocí originální rotace a potom se aktualizuje jako výsledek součinu s LastRot (a také se upraví konečná transformace). Po skončení dragu přiřadíme do LastRot hodnoty z ThisRot. Kdybychom neakumulovali rotace samotné, model by vypadal, jako by se při každém kliknutí přilepil na začátek souřadnic. Například při rotaci okolo osy x o 90 stupňů a potom o 45 stupňů, chceme získat 135 namísto posledních 45.

```
Matrix4fT Transform =// Finální transformace
```

```
{  
    1.0f, 0.0f, 0.0f, 0.0f,  
    0.0f, 1.0f, 0.0f, 0.0f,  
    0.0f, 0.0f, 1.0f, 0.0f,  
    0.0f, 0.0f, 0.0f, 1.0f  
};
```

```
Matrix3fT LastRot =// Minulá rotace
```

```
{  
    1.0f, 0.0f, 0.0f,  
    0.0f, 1.0f, 0.0f,  
    0.0f, 0.0f, 1.0f  
};
```

```
Matrix3fT ThisRot =// Současná rotace
```

```
{  
    1.0f, 0.0f, 0.0f,
```

```

    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f
};

```

Co se týče zbytku proměnných (kromě isDragged), všechno, co s nimi musíme udělat, je vždy je ve správný čas aktualizovat. ArcBall potřebuje, aby se jeho hranice při každé změně velikosti okna resetovaly. MousePt se aktualizuje při pohybu myši nebo stisknutí tlačítka a isClicked/isRClicked při stlačení levého/pravého tlačítka myši. Levé tlačítko slouží pro dragging a pravé pro resetování všech rotací do výchozího identity stavu.

```

ArcBallT ArcBall(640.0f, 480.0f); // Instance ArcBallu
Point2fT MousePt; // Pozice myši

bool isClicked = false; // Kliknuto myši?
bool isRClicked = false; // Kliknuto pravým tlačítkem myši?
bool isDragging = false; // Táhnuto myši?

```

Aktualizace proměnných vypadají takto:

```

// Konec ReshapeGL()
ArcBall.setBounds((GLfloat)width, (GLfloat)height); // Nastaví hranice pro ArcBall

// Funkce WindowProc() - ošetření zpráv myši
case WM_MOUSEMOVE: // Pohyb
    MousePt.s.X = (GLfloat)LOWORD(lParam);
    MousePt.s.Y = (GLfloat)HIWORD(lParam);
    isClicked = (LOWORD(wParam) & MK_LBUTTON) ? true : false;
    isRClicked = (LOWORD(wParam) & MK_RBUTTON) ? true : false;
    break;

case WM_LBUTTONDOWN: // Uvolnění levého tlačítka
    isClicked = false;
    break;

case WM_RBUTTONDOWN: // Uvolnění pravého tlačítka
    isRClicked = false;
    break;

case WM_LBUTTONDOWN: // Kliknutí levým tlačítkem
    isClicked = true;
    break;

case WM_RBUTTONDOWN: // Kliknutí pravým tlačítkem
    isRClicked = true;
    break;

```

Máme-li toto všechno, je na čase vypořádat se s klikací logikou.

```

void Update(DWORD milliseconds) // Aktualizace scény
{
    if (g_keys->keyDown [VK_ESCAPE] == TRUE) // Stisk ESC
    {
        TerminateApplication(g_window); // Ukončení programu
    }

    if (g_keys->keyDown [VK_F1] == TRUE) // Stisk F1
    {
        ToggleFullscreen(g_window); // Přepnutí do fullscreenu
    }

    if (isRClicked) // Kliknutí pravým tlačítkem - reset všech rotací
    {
        Matrix3fSetIdentity(&LastRot);
        Matrix3fSetIdentity(&ThisRot);
        Matrix4fSetRotationFromMatrix3f(&Transform, &ThisRot);
    }

    if (!isDragging) // Netáhne se myši?
    {
        if (isClicked) // Kliknutí?
        {
            isDragging = true; // Příprava na dragging
            LastRot = ThisRot; // Nastavení minulé statické rotace na tuto

```



```

        ArcBall.click(&MousePt);// Aktualizace startovního vektoru a příprava na
        dragging
    }
}
else// Už se táhne
{
    if (isClicked)// Je ještě stisknuto tlačítko?
    {
        Quat4fT ThisQuat;

        ArcBall.drag(&MousePt, &ThisQuat);// Aktualizace koncového vektoru a získání
        rotace jako quaternionu
        Matrix3fSetRotationFromQuat4f(&ThisRot, &ThisQuat);// Konvertování
        quaternionu na Matrix3fT
        Matrix3fMulMatrix3f(&ThisRot, &LastRot);// Akumulace minulé rotace do této
        Matrix4fSetRotationFromMatrix3f(&Transform, &ThisRot);// Nastavení koncové
        transformační rotace na tuto
    }
    else// Už není stisknuto
    {
        isDragging = false;// Konec draggingu
    }
}
}
}
}

```

Ted' už jenom potřebujeme aplikovat transformaci na naše modely a jsme hotovi.

```

void Draw(void)// Vykreslování
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);// Smazání bufferů
    glLoadIdentity();// Reset matice
    glTranslatef(-1.5f, 0.0f, -6.0f);// Translace doleva a do hloubky

    glPushMatrix();// Uložení matice
        glMultMatrixf(Transform.M);// Aplikování transformace
        glColor3f(0.75f, 0.75f, 1.0f);// Barva
        Torus(0.30f, 1.00f);// Vykreslení toroidu (speciální funkce)
    glPopMatrix();// Obnovení původní matice

    glLoadIdentity();// Reset matice
    glTranslatef(1.5f, 0.0f, -6.0f);// Translace doprava a do hloubky

    glPushMatrix();// Uložení matice
        glMultMatrixf(Transform.M);// Aplikování transformace
        glColor3f(1.0f, 0.75f, 0.75f);// Barva
        gluSphere(quadratic,1.3f,20,20);// Vykreslení koule
    glPopMatrix();// Obnovení matice

    glFlush();// Flushnutí renderovací pipeline
}

```

Přidal jsem i ukázkou kompletního kódu, který toto všechno demonstruje. Nemusíte používat moji matematiku a funkce stojící na pozadí, naopak, pokud si věříte, doporučuji vytvořit si vlastní. Nicméně i s mými vzorci a výpočty by všechno mělo bez problémů fungovat.

napsal: Terence J. Grant <tjgrant (zavináč) tatewake.com>
do slovenštiny přeložil: Pavel Hradský - PcMaster <pcmaster (zavináč) stonline.sk>
do češtiny přeložil: Michal Turek - Woq <WOQ (zavináč) email.cz>