Několik poznámek k tvorbě počítačových her

Bernard Lidický

28. října 2004

1 Předmluva

Tento článek by měl být shrnutím mých zkušeností s tvorbou počítačových her a mohl by usnadnit život začínajícím amatérským tvůrcům her a zejména programátorům.

Nejprve se podíváme na několik obecných věcí o hrách a pak se vrhneme na grafiku, klávesnici s myší a nakonec na čas. V každé části se pokusíme vytvořit nějaké příklady a na nich předvést o čem je řeč.

Předpoklad při psaní textu byl, že čtenář již umí trochu programovat a ovládá alespoň základy jazyka C.

Většina rad a srovnání v tomto textu pochází z mých vlastních zkušeností při tvorbě počítačových her. Možná se budou hodit i vám.

Místy se pokusím uvést příklady z mé tvorby. Všechny mnou vytvořené hry lze najít na `http://hippo.nipax.cz`. U drtivé většiny je dostupný i zdrojový kód.

Na adrese `http://hippo.nipax.cz/text` jsou ke stažení kompletní zdrojové kódy ke všem příkladům v tomto textu i text samotný.

Případné návrhy na zlepšení tohoto textu nebo objevené chyby posílejte, prosím, emailem na adresu `2berny@seznam.cz`.

2 Ještě před programováním

Zajímavá otázka hned ze začátku je, proč vlastně dělat hry? Možná není špatné si na ni odpovědět, ještě než se začne cokoli dělat. Snáze se pak dosahuje cíle, když se ví, jaký vlastně je.

Například já jsem začal dělat hry, abych se naučil programovat a tiše jsem doufal, že je někdo bude hrát. Někdo třeba může dělat hry, aby se proslavil. Mimochodem pokud bude vaším cílem dělat hry, které bude hrát co možná nejvíc lidí, dělejte hry erotické. Není to žádný žert. Erotické hry mají **několikanásobně** větší stahovanost než ostatní.

Ještě než se podíváme na výběr témat, přichází malé doporučení. Nesrovnávejte se s profesionálními tvůrci her. Většinou není šance udělat dokonalou grafiku, rozsáhlý příběh a engine se skriptováním jako mohou skupiny lidí, kteří sedí od rána do večera u počítače a nedělají nic jiného. Doporučil bych zaměřit se zejména na hratelnost. Tam můžete soutěžit s kýmkoli. Aby hra byla zábavná, není nutnou a ani postačující podmínkou záplava grafických efektů. Škoda, že na to dnes hodně lidí zapomíná.

Při výběru tématu si dejte hlavně pozor, ať si toho nevymyslíte příliš. Mnoho počítačových projektů[1] skončí neúspěšně velmi brzy po svém vzniku, protože si jejich autoři na začátku na svá bedra naloží více, než jsou nakonec schopní zvládnout.

Vhodné žánry pro tvorbu hry jsou hry akční, arkády nebo logické. U akčních her většinou není potřeba vymýšlet složitý příběh a když bude okolo stříkat krev nebo budou vybuchovat roboti, budou hráči spokojení. Logické hry jsou příjemné, protože není potřeba se příliš starat o rychlost programu[2] a různé podobné radosti jako v ostatních žánrech. Aby se všichni panáčci pohybovali stejně rychle a naráželi do zdí,...

Naopak bych nedoporučil dělat realtime-strategie. Je to příliš mnoho práce. Stačí se zamyslet nad tím, co vše hra musí umět.

Ještě než se pustíte do tvorby, může být dobrý nápad si rozvrhnout kdo bude co dělat. A pak se do toho těm druhým příliš nemontovat. Mně se to velice osvědčilo.

3 Začínáme programovat

První, co nás čeká a nemine, je výběr programovacího jazyka. Většina her je psaná v jazyce C/C++ a i já bych ho doporučil. Sežene se pro něj hodně příkladů a je dostatečně svižný.

[1] Teď se to netýká jen her.

[2] Pokud nepíšete šachy, kde má hrát i počítač.

Další věc kromě výběru jazyka je výběr knihovny pro tvorbu her. Když chceme psát hry, je zbytečná práce tvořit ovladače grafiky a podobné věci, protože to už dávno vytvořil někdo jiný a nakonec bychom si po velkém úsilí[3] vytvořili vlastní knihovničku. Ta by se ale stejně podobala těm již hotovým.

Výběr knihoven usnadňující tvorbu her, které já jsem použil, je asi následující.

- SDL, SDL a OpenGL
- Allegro, (Allegro a OpenGL)
- DirectX

3.1 DirectX

DirectX je API od Microsoftu. Původně bylo pouze pro hry, ale dnes jej používají i jiné programy pro přístup k multimédiím. Obsahuje prakticky vše, co je pro psaní hry potřeba. Umí si poradit s grafikou, videem, klávesnicí,... Přesto se DirectXu nebudeme dále věnovat kvůli několika nevýhodám a mému osobnímu odporu. Pokud přeci jen toužíte používat DirectX, zkuste se poohlédnout po nějaké jeho nadstavbě. Například CDX nebo NukeDX. Usnadní to mnoho práce.

Důvody proč ne DirectX

- API se mi zdá příliš komplikované a zabere hodně času se naučit s ním pracovat.
- Je potřeba napsat hodně kódu, aby program vůbec něco dělal.
- API se často mění. A mění se tak, že není zpětně kompatibilní. Tedy nová verze znamená nové ponoření se do manuálů a přepisování programu. Je otázka, zda je vývoj opravdu tak rychlý, nebo je API tak nevhodně navržené, že se musí s každou novou verzí celé změnit.
- U verzí menších než 7 je **lživá** dokumentace. Nezapomenu, když jsem týden od rána do večera seděl u počítače a marně se snažil použít rotaci nebo poloprůhlednost. V manuálu k DX7 bylo pak pod čarou napsáno: Currently not implemented.
- U verzí novějších než 7 pro změnu chybí podpora 2D grafiky. Tedy i 2D hra potřebuje pro svůj běh 3D akceleraci, aby se hýbala rychle.
- Poslední[4] nevýhoda je dostupnost pouze na Windows. Může se stát, že jednoho krásného dne opustíte Windows a přestěhujete se do Linuxu. Hru Kulič jsem pod Linux přeportoval za 3 hodiny, ale u Bombiče napsaného v DirectX8 jsem to raději ani nezkoušel. Znamenalo by to kompletní přepsání programu.

Možná výhoda je, že pro DirectX je hodně tutoriálů. Proč je jich vlastně hodně? Možná proto, že jsou potřeba, aby to normální programátoři pochopili.

3.2 SDL

SDL je knihovna domovem v Linuxu, ale funguje i na Win32, BeOS, MacOS, MacOS X, FreeBSD, Sloaris, Irix, EPOC a dalších. SDL je zkratka za Simple Direct Layer. Tomu odpovídá i struktura knihovny. Knihovna sama je malá a obsahuje jen základní věci. Díky tomu je přehledná a nezahrnuje programátora gigantickým API. Například sama umí načítat obrázky pouze ve formátu BMP. K SDL pak existuje řada rozšiřujících knihoven. Mezi skoro oficiální rozšíření patří

- `SDL_net` pro tvorbu síťových her.
- `SDL_image` pro načítání obrázků rozličných formátů

[3] Pokud by nás to nestihlo od tvorby her odradit.

[4] a pro mě zásadní, protože používám Linux

- `SDL_mixer` pro lepší práci se zvukem
- `SDL_ttf` pro podporu TTF fontů
- `SDL_bitmap` pro lepší práci s bitmapami
- A mnoho dalších je ke stažení na domovské stránce `http://www.libsdl.org`

3.3 Allegro

Allegro je podobně jako SDL dostupné na mnohých OS. Například Linux, Windows, BeOS, QNX, MacOS nebo i DOS. Na rozdíl od SDL je velké množství funkcí přímo v knihovně. Obsahuje i takové věci, jako je možnost zabalit bitmapy a zvuky do jednoho komprimovaného souboru. Dobrá zpráva pro odpůrce angličtiny je, že domácí stránky Allegra
`http://www.talula.demon.co.uk/allegro`. jsou i v češtině. Stránky přátel Allegra jsou
`http://www.allegro.cc` Nachází se tam ke stažení mnoho rozšíření knihovny, utilit a her vytvořených v Allegru.

Je těžké rozhodnout, zda je lepší Allegro nebo SDL. Proto se budeme dále věnovat oběma knihovnám.

4 Základ SDL a Allegra

Nejprve si ukážeme, jak napsat program v Allegru nebo SDL, který vlastně nic nedělá. Začneme s příkladem Allegra, protože je nepatrně jednodušší.

V příkladech nebudeme úplně přesně a do detailu zkoumat, co která funkce dělá a jaké přesně musí mít parametry. To je velice pěkně popsané v manuálech ke knihovnám. Naším cílem je udělat několik malých programů, na kterých je vidět, jak lze knihovny použít.

4.1 Příklad 1

V prvním příkladu si ukážeme, jak napsat program, který vlastně nic nedělá. Tedy nutné minimum pro to, aby program fungoval.

4.1.1 allegro_1.c

Nejprve je nutné přidat hlavičkový soubor `allegro.h`.

```
#include <stdio.h>
#include <allegro.h>

// Funkce main
int main( )
{
```

Na začátku programu je potřeba Allegro inicializovat.

```
   // Inicializace Allegra
   if ( allegro_init() < 0 ) {
      fprintf(stderr, "Selhala inicializace Allegra.\n");
      exit(1);
   }
```

A na konci programu ho zase vypnout. Jde spíš o slušnost. Pokud se na to zapomene, nic se neděje, Allegro se vypne samo.

```
   // ukonceni Allegra - neni nutne volat
   allegro_exit();

   return 0;
}
```

Za funkcí main musí následovat `END_OF_MAIN();` Je to makro, které zajistí správné vytvoření funkce main. Například ve Win32 se funkce `main` nejmenuje `main`, ale `WinMain`. Toto makro se o vše postará.

```
// toto musi byt za funkci main - zalezitost Allegra
END_OF_MAIN();
```

Tento příklad se v Linuxu zkompiluje například pomocí příkazu
`gcc allegro_1.c -o allegro_1 ‘allegro-config --cflags --libs‘`[5]
Další příklady se přeloží obdobně.

4.1.2 sdl_1.c

Knihovna SDL posílá programu zprávy a ty je vhodné odchytávat. Je to podobné jako WinAPI nebo jiné systémy se zprávami.

Na začátku programu je potřeba přidat soubor `SDL.h`

```
#include <stdio.h>
#include <SDL.h>
```

Ještě než začneme psát funkci `main`, napíšeme jednoduchou funkci na zpracovávání zpráv od SDL. Ve while cyklu budeme vybírat zprávy z fronty zpráv dokud tam nějaké jsou. Vybrané zprávy pak budeme zpracovávat. Bude stačit jen zpráva `SDL_QUIT`. Ta říká, že si někdo přeje, aby se náš program ukončil.[6]

```
// Zpracovava zpravy zasilane od SDL
int EventLoop()
{
   SDL_Event event;

   // z fronty zprav vybirame zpravy, pokud tam nejake jsou
   while ( SDL_PollEvent(&event) ) {
      switch (event.type) {

      // zprava s pozadavkem na ukonceni aplikace
      case SDL_QUIT :
         return 0;
         break;
      }
   }

   return 1;
}
```

Na začátku funkce main je potřeba provést inicializaci knihovny. Funkcí `atexit` nastavíme, že se má ukončovat SDL při konci programu. U menších programů je doporučené to takto dělat. U větších programů je lepší udělat vlastní čistící funkci, která bude volat funkci `SDL_Quit`

[5] Příkaz je uzavřen ve zpětných apostrofech. Zpětný apostrof bývá na klávesnici pod klávesou Esc.
[6] Například uživatel kliknutím na křížek v liště okna.

```
// Funkce main
int main( )
{
   // Inicializace SDL
   if ( SDL_Init(0) < 0 ) {
      fprintf(stderr, "Selhala inicializace SDL: %s\n", SDL_GetError());
      exit(1);
   }

   // Je vhodne pridat automaticke volani SDL_Quit pri ukonceni programu exitem
   atexit(SDL_Quit);
```

V hlavní smyčce nebudeme dělat nic. Budeme jen čekat, dokud si uživatel nebude přát program ukončit.

```
   // smycka zprav
   while ( EventLoop() ) ;
```

Na konci programu SDL ukončíme.

```
   // ukonceni SDL
   SDL_Quit();

   return 0;
}
```

Zdrojový kód můžeme přeložit například tímto příkazem
`gcc sdl_1.c -o sdl_1 ‘sdl-config --cflags --static-libs‘.`

5 Povídání o grafice

Grafika je velice důležitá věc. Je to to první, co na screenshotech z hry hráči uvidí. A podle čeho se rozhodnou, zda si hru zkusí zahrát nebo raději svůj čas stráví jinak. Budeme se jí proto věnovat nejvíce. V současné době je velice moderní 3D grafika a 3D hry.

3D má bohužel několik úskalí. Na 2D hru stačí obrázek na papíře a matematika ze základní školy. Ale ve 3D z matematiky bolí hlava a vysokoškolská přednáška o maticích rozhodně přijde vhod.

Také kolize jsou ve 2D celkem příjemná věc, ale ve 3D to je komplikovaná záležitost. Je zde ještě možnost udělat hru ve 2D světě, kde se ale budou pohybovat 3D modely. Tím se dosáhne moderního 3D vzhledu, ale počítání kolizí bude i nadále jednoduché.

Allegro ani SDL 3D API přímo nemají.[7] Obě ale umožňují použít knihovnu OpenGL. OpenGL je rozhraní, které umožňuje práci s 3D grafikou. Je to průmyslový standart, který původně s hrami nechtěl mít nic společného, ale dnes se pro hry používá.

3D grafika a OpenGL bohužel přesahují rámec tohoto textu, takže se jim dále nebudeme věnovat. Případní zájemci naleznou velice pěkné tutoriály pro OpenGL v češtině na
`http://nehe.opengl.cz`

Kdo bude hledat srovnání DirectX a OpenGL, možná časem narazí na zajímavou historku. Původně byla hra Quake X[8] napsána v DirectX. Jeden výrobce 3D karet měl kartu, která uměla OpenGL, ale neuměla DirectX. Přišel tedy za IDsoftem a zeptal se, zda by IDsoft nechtěl používat OpenGL místo DirectX. IDsoft pak za víkend přepsal zdrojové kódy do OpenGL a zjistil, že hra je rychlejší a kód je o třetinu menší. A tak u OpenGL zůstal.

[7] Allegro sice malý náznak má, ale nezdál se mi příliš použitelný.
[8] Bohužel neznám správné pořadové číslo.

5.1 Výlet do minulosti

V dobách, kdy byly počítače pomalé[9], si hry hlídaly, co je na obrazovce a překreslovaly jen změny. Příkladem takové hry je BoxMaster. Já tedy takové doby ještě pamatuji.

Dnes nás naštěstí nic podobného nečeká. Programátoři jsou líní optimalizovat a navíc počítače jsou dostatečně rychlé, aby stačily pokaždé nakreslit celou obrazovku od začátku. Program se tím zjednoduší a nestane se, že se při překreslování splete a na obrazovce budou zbývat kusy obrázků, které tam nemají co dělat.

V dobách nedostatku paměti se grafika řešila kreslením čar, koleček, obdélníčků a podobných věcí. Jako příklad opět poslouží BoxMaster.

Dnes se na sebe plácají různě otočené a ořezané bitmapy[10] a tím vznikne jeden frame. Grafické operace se tedy většinou realizují stěhováním kusů paměti po paměti[11].

5.2 Rozlišení a barevná hloubka

Když budeme chtít ve hře grafiku použít, je potřeba nastavit rozlišení a barevnou hloubku.

Co to je rozlišení jistě každý ví. Je to počet pixelů[12], které má obrazovka na šířku a na výšku. Při příliš malém rozlišení bude mít hra málo grafických detailů a při příliš velkém rozlišení bude pomalá, protože vykreslit velkou obrazovku trvá dlouho.[13]

Barevná hloubka[14] je počet bitů udávajících barvu na jeden pixel. Používá se několik různých hloubek: 8, 15, 16, 24 a 32. Barva se skládá ze složek červená, zelená a modrá[15].

Barevné formáty se liší v barevné hloubce a pořadí uložení barevných složek[16]. Většinou se při inicializaci grafiky nevybírá přímo barevný formát, ale jen barevná hloubka. Proto Allegro i SDL poskytují funkce, které zakódují tři hodnoty pro červenou, modrou a zelenou do aktuálního barevného formátu.

- 8 BPP: Barevná hloubka 8 BPP pracuje s paletou a dnes se téměř nepoužívá, protože její použití přináší více komplikací, než výhod. Jediná a hlavní výhoda, proč se dříve používala, je malá spotřeba paměti.

- 15 a 16 BPP: 15 a 16 bitů[17] na pixel sice nestačí na pokrytí všech barev, ale pro hry je to více než dost. Tyto barevné hloubky mají výhodu, že nepoužívají paletu a přitom nezabírají příliš paměti. Navíc, pokud je potřeba pixely nějakým způsobem zpracovávat procesorem, na 32 bitovém procesoru mohou být zpracovávány 2 pixely současně. Tedy oproti větším barevným hloubkám jsou 15 a 16 výhodnější jak z hlediska času, tak z hlediska paměti. Bohužel nedokáží zobrazit všechny barvy a plynulý barevný přechod.

  15 BPP vyhrazuje pro každou barevnou složku 5 bitů. Protože počítače raději pracují se 16 bity a 16 bitů se lépe rovná do paměti, nechává se jeden bit nevyužitý. Ten se může použít například pro určení, zda je barva pixelu skutečně platná a má se kreslit nebo se má pixel chovat jako průhledný.

  16 BPP má pro červenou a modrou složku 5 bitů a pro zelenou složku 6 bitů. Na zelenou barvu je lidské oko nejcitlivější, proto je logické 1 bit informace navíc z 15BPP modelu použít právě pro zelenou barvu.

[9] Přeskakujeme dobu hodně hodně pomalých strojů, které žádnou grafiku neměli.

[10] Nebo se grafické kartě sypou hromady trojúhelníků na renderování.

[11] Slyšel jsem, že většina programů vlastně jen stěhuje paměť z místa na místo a občas někam přičte 1.

[12] pixel = 1 barevný bod

[13] Doba bývá přímo úměrná počtu pixelů na obrazovce. Ten však roste s rostoucím rozlišením obrazovky kvadraticky.

[14] Zkratka je BPP - bites per pixel

[15] Tzv. barevný model RGB (Red-Green-Blue). Každá z barev může mít hodnotu 0 až 255. Používají ho monitory a grafické karty.

[16] Může být model 24BPP a pořadí barev RGB nebo BGR. Anglicky pixel format.

[17] Některé grafické karty umí jen 15 BPP a jiné jen 16 BPP. V Allegru 15 a 16 není to samé a je potřeba drobné ošetření při inicializaci grafiky.

Obrázek 1: 15 BPP

Obrázek 2: 16 BPP

- 24 BPP : Hloubka 24 BPP už je schopná popsat všechny barvy z RGB, protože pro každou složku má 8 bitů. nevýhoda 24 BPP je, že není příliš dobře zarovnána v paměti pro 32bitový procesor a některé operace s pixely jsou proto časově náročnější než u 32 BPP. Proto se také příliš nehodí pro použití ve hrách.

Obrázek 3: 24 BPP

- 32 BPP : 32 BPP vypadá podobně jako 24 BPP. Navíc ale přidává ještě dalších 8 bitů, aby se velikost jednoho pixelu zarovnala na 32 bitů. Těchto 8 bitů pak zůstává nevyužito nebo se mohou použít pro informaci o poloprůhlednosti pixelu. Takový barevný model se nazývá ARGB[18].

Použití v programu je velice jednoduché. Stačí říci, že chceme použít grafiku, vybrat rozlišení a barevnou hloubku.

V Allegru je globální proměnnou `screen`, která ukazuje na obrazovku, a SDL při vytvoření obrazovky vrátí ukazatel na bitmapu obrazovky. Když chceme něco dostat na obrazovku, jednoduše to na ní překopírujeme.

5.3 Příklad 2

Předvedeme si, jak zapnout grafický režim, načíst obrázek ze souboru a zobrazit ho na obrazovku. Program po svém spuštění vytvoří okno a v něm nakreslí obrázek rybičky, který pojede z levého horního rohu šikmo dolů.

5.3.1 allegro_2.c

Připravíme si proměnou na obrázek rybičky a proměnné pro souřadnice rybičky.

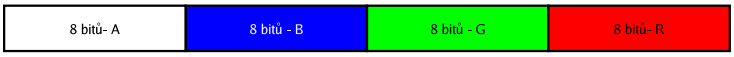
```
BITMAP *ryba;
int x = 0, y = 0;
```

Inicializaci grafiky provedeme tak, že nejprve vybereme barevnou hloubku a pak se pokusíme nastavit rozlišení. Ještě lepší by bylo, kdyby program po selhání funkce `set_gfx_mode` změnil nastavení barevné hloubky na 15 a znovu zkusil zavolat funkci `set_gfx_mode`.

```
// nastaveni barevne hloubky
set_color_depth(16);

// nastaveni grafickeho modu
```

[18] Alpha-Red-Green-Blue



Obrázek 4: 32 BPP

```
if ( set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0) != 0 ){
   fprintf(stderr, "Nepodarilo nastavit rozliseni 640x480x16\n");
   exit(1);
}
```

Načteme obrázek ryba.bmp. Funkce `load_bitmap` sama pozná formát souboru a vytvoří místo pro bitmapu.

```
// nacteme bitmaku
ryba = load_bitmap( "ryba.bmp", NULL );
if ( ryba == NULL ) {
   fprintf(stderr, "Nepodarilo se nacist soubor ryba.bmp.\n");
   exit(1);
}
```

V jednoduchém cyklu budeme posouvat obrázkem po obrazovce. Allegro automaticky nekreslí růžovou barvu okolo ryby, protože ji považuje za průhlednou. Lze samozřejmě nastavit i jinou barvu jako průhlednou.

```
while ( y < 480 ) {
   clear_to_color(screen, makecol(180, 200, 255)); //  smazani obrazovky
   draw_sprite(screen, ryba, x++, y++);  // nakresleni ryby
}
```

Na konci je potřeba uvolnit paměť, kterou obrázek zabíral.

```
// uvolneni nactene bitmapky
destroy_bitmap(ryba);
```

5.3.2 sdl_2.c

SDL nemá žádnou globální proměnnou `screen` jako Allegro. Pokud jí chceme, musíme si ji vytvořit sami.

Podobně jako v Allegru si připravíme proměnné na obrázek a pozici ryby. Pro pozici nepoužijeme x a y, ale použijeme `SDL_Rect`, se kterým se lépe pracuje.

```
SDL_Surface *screen; // obrazovka

// Funkce main
int main( )
{
   SDL_Surface *ryba; // obrazek ryby
   SDL_Rect pozice;   // pozice ryby
```

Nastavíme rozlišení a načteme obrázek ze souboru ryba.bmp

```
   // nastavime rozliseni okna a barevna hloubka nas nezajima
   if ( (screen = SDL_SetVideoMode(640, 480, 0, 0)) == NULL ) {
      fprintf(stderr, "Nepodarilo se vytvorit okno: %s\n", SDL_GetError());
      exit(2);
```

```
}

// nacteme obrazek ryby
ryba = SDL_LoadBMP("ryba.bmp");
if (ryba == NULL) {
   fprintf(stderr, "Nepodarilo se nacist ryba.bmp: %s\n", SDL_GetError());
   exit(2);
}
```

SDL nepovažuje automaticky žádnou barvu za průhlednou. Proto je potřeba říci, že chceme, aby byla růžová barva považována za průhlednou. Nakonec si ještě připravíme počáteční pozici ryby na obrazovce.

```
// nastavime, ze ruzova se nekresli
if ( SDL_SetColorKey(ryba,  SDL_SRCCOLORKEY,
                     SDL_MapRGB( ryba->format, 255, 0, 255)) < 0 ) {
   fprintf(stderr, "Nepodarilo se nastavit colorkey: %s\n", SDL_GetError());
   exit(2);
}

// priprava pozice ryby
pozice.x = 0;
pozice.y = 0;
pozice.w = ryba->w;
pozice.h = ryba->h;
```

Vytvoříme smyčku, ve které budeme posouvat obrázkem ryby po obrazovce. Když se něco nakreslí na `screen` je nutné zavolat `SDL_UpdateRect`, aby se změna projevila na obrazovce.

```
// nejaka smycka zprav
while ( EventLoop() ) {

   pozice.x++;
   pozice.y++;

   if ( pozice.y > 480 ) break;

   // vymazeme obrazovku do svetle modre
   SDL_FillRect(screen, NULL,  SDL_MapRGB( screen->format, 180, 200, 255));

   // nakreslime rybu
   SDL_BlitSurface(ryba, NULL, screen, &pozice);

   // aktualizujeme obrazovku
   SDL_UpdateRect(screen, 0, 0, 640, 480);
}
```

Na konci je potřeba uvolnit paměť zabranou obrázkem.

```
// uvolnime obrazek ryby
SDL_FreeSurface(ryba);
```

Pozorný a zvídavý čtenář si jistě všiml, že program v Allegru nedělá to samé, co jeho ekvivalent v SDL. Program v Allegru bliká. Problém je v tom, že kreslíme na obrazovku a neříkáme, kdy už je obrazovka připravená k zobrazení. Pak se samozřejmě stává, že obrázek ryby občas zmizí. V SDL funkcí `SDL_UpdateRect` říkáme, že obrazovka je připravená k zobrazení. V Allegru si musíme pomoci jinak. Existují dvě příjemná a použitelná řešení.

5.4 Příklad 3

Velice jednoduché a fungující řešení je vytvořit v paměti bitmapu velikosti obrazovky. Do ní vše nakreslit a jednorázově ji překopírovat na obrazovku.

5.4.1 allegro_3.c

Na začátku je potřeba vytvořit bitmapu velkou jako obrazovka.

```
// druha bitmapa velikosti obrazovky
BITMAP *back = NULL;

// vytvorime druhou obrazovku
back = create_bitmap(SCREEN_X, SCREEN_Y);
if ( back == NULL ) {
   fprintf(stderr, "Nepodarilo se vytvorit druhou bitmapu.\n");
   exit(1);
}
```

Vše co se dříve kreslilo přímo na `screen` teď nakreslíme na `back`. Až bude vše připravené, nakreslí se `back` na `screen`.

Na konci je samozřejmě nutné druhou bitmapu `back` uvolnit.

```
while ( y < SCREEN_Y ) {
   clear_to_color(back, makecol(180, 200, 255));    //  smazani obrazovky

   // nakreslime nekolik ryb
   draw_sprite(back, ryba, x, y);
   draw_sprite(back, ryba, x, SCREEN_Y-y);
   draw_sprite(back, ryba, x, SCREEN_Y/2);
   draw_sprite(back, ryba, SCREEN_Y/2, y);

   x++, y++; // posunuti rybou

   // aktualizace obrazovku
   draw_sprite(screen, back, 0, 0);
}

// uvolneni bitmapy s druhou obrazovkou
destroy_bitmap(back);
```

5.5 Příklad 4

Druhé řešení (tzv. doublebuffering, či page flipping) spočívá v použití dvou bitmap velikosti obrazovky, které jsou obě umístěné ve videopaměti. Zatímco jedna bitmapa je vidět na monitoru, druhou program překresluje. Když je druhá bitmapa připravená na zobrazení, začne grafická karta místo první bitmapy zobrazovat druhou bitmapu a program mezitím může překreslovat první bitmapu. Výhodu tohoto řešení je, že jednou nakreslená bitmapa s obrazovkou se už nikam nepřesouvá, jako v minulém řešení, ale přehazují se jen ukazatele mezi zobrazovanou a překreslovanou bitmapu. Nevýhodou tohoto řešení je, že v paměti grafické karty musíte mít najednou dvě obrazovky. Spotřebuje se tedy dvakrát tolik videopaměti, které nemusí být mnoho.

5.5.1 allegro_4.c

V Allegru budeme potřebovat dvě bitmapy ve videopaměti vytvořit sami. Proto si na ně připravíme místo.

```
BITMAP *back = NULL; // kam se ma kreslit - urcije aktualne nezobrazenou
BITMAP *page_1, *page_2; // bitmapy pro doublebuffering
```

Poté, co inicializujeme grafiku, můžeme bitmapy vytvořit. Jejich vytvoření se nemusí vždy povést. Příčinou může být málo videopaměti nebo to, že program běží v okně a ne přes celou obrazovku. V případě, že se to skutečně nepovede, by program měl zareagovat tak, že začne používat řešení z příkladu 3. Nicméně v našem ukázkovém příkladu při neúspěšném tvoření bitmap program ukončí svou činnost.

```
// vytvorime 1. obrazovku
page_1 = create_video_bitmap(SCREEN_X, SCREEN_Y);
if ( page_1 == NULL ) {
   fprintf(stderr, "Nepodarilo se vytvorit prvni video bitmapu.\n");
   exit(1);
}

// vytvorime 2. obrazovku
page_2 = create_video_bitmap(SCREEN_X, SCREEN_Y);
if ( page_2 == NULL ) {
   fprintf(stderr, "Nepodarilo se vytvorit druhou video bitmapu.\n");
   exit(1);
}
```

Jediná zajímavá změna v hlavní smyčce programu je prohazování bitmap. Na konci programu je potřeba obě bitmapy z videopaměti zrušit.

```
// 1. inicializace
back = page_1;

while ( y < SCREEN_Y ) {
   // kresleni hry na back
   MovaAndDrawGameWorld(back)

   // aktualizace obrazovky - prohozeni videobitmap
   show_video_bitmap(back);

   // prehozeni mista, kam se bude kreslit
   if ( back == page_1 ) back = page_2;
   else back = page_1;
}

// uvolneni bitmap z videopameti
destroy_bitmap(page_1);
destroy_bitmap(page_2);
```

5.5.2 sdl_4.c

Při použití SDL je zapnutí doublebufferingu velice snadné. Při inicializaci stačí nastavit navíc příznak `SDL_DOUBLEBUF` a místo funkce `SDL_UpdateRect` volat funkci `SDL_Flip`. Výhodou je, že pokud vytvoření dvou bitmap ve videopaměti selže, program v klidu dál poběží a funkce `SDL_Flip` se bude chovat stejně jako se chová funkce `SDL_UpdateRect`.

```
// Inicializace SDL a subsytemu s grafikou a doublebufferingem
if ( SDL_Init(SDL_INIT_VIDEO|SDL_DOUBLEBUF) < 0 ) {
   fprintf(stderr, "Selhala inicializace SDL: %s\n", SDL_GetError());
```

```
    exit(1);
}
...
while ( EventLoop() ) {
    // pohnuti hry a kresleni na screen
    MoveAndDrawGameWorld();

    // aktualizujeme obrazovku
    SDL_Flip(screen);
}
```

6 Klávesnice a myš

Jako programátoři v jazyce C jistě známe funkci `scanf`. Tou sice vstup od hráče získat můžeme, ale hodí se tak nanejvýš pro logické hry. Potvrzovat každý vstup stiskem klávesy Enter asi není to pravé, že.

O něco lepší by bylo použít funkci `getc`. Pak by se ale stávalo, že se hra bude zastavovat, pokud hráč nebude nic mačkat. To by šlo opravit použitím více vláken nebo nějakou funkcí `keypressed`, ale při akčních hrách je závažná nevýhoda, že nevíme, jak dlouho hráč klávesu držel.

Proto Allegro i SDL mají kromě normálního přístupu ke klávesnici ještě jeden, který je vhodný zejména pro akční hry.

Knihovny poskytují obyčejné pole indexované jménem klávesy. U každé klávesy je pak stisknuto/nestisknuto.

Když se hra čas od času dostane k pohybu panáčka ovládaného hráčem, jednoduše se podívá, co zrovna hráč má zmáčknutého a tím zjistí, co chce hráč dělat.

S myší je to velice podobné. Knihovny poskytují snadný způsob, jak zjistit aktuální pozici myši a stavy tlačítek.

6.1 Příklad 5

Ukážeme si jednoduchý příklad, který umožňuje pohybovat obrázkem ryby po obrazovce pomocí šipek na klávesnici a pomocí myši.

6.1.1 allegro_5.c

Na začátku musíme říci Allegru, že chceme inicializovat navíc i myš a klávesnici.

```
// inicializace Allegra a ostatniho
InitSystem();

// pridame klavesnici a mys
install_keyboard();
install_mouse();
```

Ke klávesnici se přistupuje pomocí globální proměnné `key`. Ta se chová jako pole indexované pomocí konstant definovaných Allegrem. Před čtením z pole je vhodné volat funkci `poll_keyboard`, která zajistí, že pole obsahuje aktuální hodnoty. Allegro se občas samo pokouší pole kláves aktualizovat, ale na některých systémech je nutné explicitně žádat o aktualizaci.

```
while ( !key[KEY_Q] ) { // koncime na stisk q

    poll_keyboard(); // pred ctenim z klavesnice je vhodne volat

    // pohybova cast
    if ( key[KEY_LEFT] )  x--;
```

```
    if ( key[KEY_RIGHT] ) x++;
    if ( key[KEY_UP] )    y--;
    if ( key[KEY_DOWN] )  y++;
```

U myši je podobná věc s aktualizací jako u klávesnice. To, že hra funguje na některých počítačích bez volání `poll_mouse` bohužel neznamená, že to bude fungovat všude. Je proto slušností funkce `poll_xx` volat. Souřadnice myši jsou v proměnných `mouse_x` a `mouse_y`. V proměnné `mouse_b` jsou pomocí bitových příznaků zaznamenaná stisknutá tlačítka.

```
    // pohyb mysi
    poll_mouse();

    if (mouse_b & 1) { // kdyz se zmackne levy knoflik premistime rybu pod mys
       x = mouse_x;
       y = mouse_y;
    }

    DrawFish();
}
```

6.1.2 sdl_5.c

Podobně jako s proměnnou `screen` i proměnné pro myš a klávesnici si musíme v SDL vytvořit a pojmenovat sami. Není potřeba říkat SDL, že chceme inicializovat klávesnici a myš. Inicializují se hned při inicializaci SDL automaticky.

```
Uint8  *key; // buffer klavesnice
int     mouse_x; // mys x
int     mouse_y; // mys y

// inicializace SDL a vseho okolo
InitSystem();

// prvni cteni stavu klavesnice - inicializace *key
key = SDL_GetKeyState(NULL);
```

Použití klávesnice je téměř stejné jako v Allegru. Jen konstanty v poli pro jednotlivé znaky jsou jiné a aktualizovat musíme vždy sami.

```
// hlavni smycka
while ( EventLoop() &&  (key[SDLK_q] == 0)) {

   // cteni stavu klavesnice
   key = SDL_GetKeyState(NULL);

   // pohybova cast
   if ( key[SDLK_LEFT] )  pozice.x--;
   if ( key[SDLK_RIGHT] ) pozice.x++;
   if ( key[SDLK_UP] )    pozice.y--;
   if ( key[SDLK_DOWN] )  pozice.y++;
```

Práce s myší je také velice podobná. Funkce `SDL_GetMouseState` jako návratovou hodnotu vrací stav tlačítek myši.

```
    // pohyb pres mys
```

```
      if ( SDL_GetMouseState(&mouse_x, &mouse_y) & SDL_BUTTON_LEFT ) {
         pozice.x = mouse_x;
         pozice.y = mouse_y;
      }

      DrawFish();
   }
```

7 Čas

V poslední kapitolce se zaměříme na čas ve hrách. Ukážeme si dva funkční způsoby, jak vytvořit pohyb ve hře tak, aby hra běžela téměř stejně rychle na různých počítačích.

Ke každému řešení si vytvoříme i malý příklad a na konci je malá srovnávací tabulka.

7.1 Konstantní čas

V prvním a jednodušším řešení si na začátku zvolíme pevnou dobu, jak často chceme měnit stav hry. Pro plynulý pohyb je zapotřebí alespoň 25 obrázků za vteřinu. Tedy 1 obrázek za 40ms. Vytvoříme tedy funkci, kterou budeme hlídat čas okolo 40ms. Funkci pro pohyb hry napíšeme jednoduše, protože víme, že bude volána přibližně každých 40ms.

Protože se občas musí počkat na překreslení obrazovky nebo na plánování OS, bude měření času trochu nepřesné. Může se tedy stát, že hra poběží na různých počítačích malinko jinak rychle, ale z mých zkušeností to vůbec nevadí a nikdo si toho nevšimne.

Tento systém práce s časem byl použit například v Kuličovi nebo Racerovi, což jsou velice dobré hry.

7.2 Příklad 6

V posledním příkladu Allegra vytvoříme pohybující se rybu s konstantním časováním.

7.2.1 allegro_6.c

Nejprve potřebujeme vytvořit proměnnou pro odpočítávání času. Je potřeba ji označit `volatile`, abychom zabránili kompilátoru v některých optimalizacích[19].

Funkci, která se bude pravidelně volat, je dobré udělat co možná nejkratší. Protože s funkcí bude pracovat Allegro, je potřeba označit její konec podobně, jako se označuje konec funkce main.

```
// promena pro mereni casu
volatile int time_var = 0;

void timer_callback()
{
   // snizeni promene s casem o 1
   if (time_var > 0) time_var--;
}
END_OF_FUNCTION(timer_callback);
```

Při inicializaci programu inicializujeme funkce pro čas v Allegru.

Pomocí `LOCK_VARIABLE` a `LOCK_FUNCTION` řekneme Allegru, že funkci časovače a proměnnou pro odpočítávání je potřeba chránit před přístupem z více vláken najednou.

Nakonec nastavíme volání naší funkce na každých 10ms.

[19] `while(time_var>0) rest(5);` lze optimalizovat na `if(time_var>0) while(1) rest(5);`, což by vedlo k zacyklení programu.

```
// inicializace casu v Allegru
install_timer();

// pripraveni funkci pro timer
LOCK_VARIABLE(time_var);
LOCK_FUNCTION(timer_callback);

// nastaveni volani funkce timer_procedure kazdych 10ms
install_int(timer_callback, 10);
```

Herní smyčka by pak mohla být podobná této.

Na začátku čekáme, dokud nedoběhne `time_var` na 0. Nebudeme čekat aktivně, ale funkcí `rest` budeme odpočívat. Čekání nastavíme na 40ms ještě předtím, než začneme něco dál počítat nebo kreslit.

```
while ( !key[KEY_Q] ) { // koncime na stisk q

   // cekame, dokud timer nesnizi time_var na 0
   while ( time_var > 0 ) rest(5);

   // nechame timer snizovat time_var ze 4 (40ms)
   time_var = 4;

   MoveGameWorld();

   DrawGameWorld();
}
```

7.3 Proměnný čas

Druhá metoda je výrazně přesnější z hlediska přesnosti času, ale o něco pracnější při psaní. Princip je takový, že funkci pro pohnutí světem hry řekneme, jak dlouho jsme ji už nevolali. A ta se postará o pohnutí světem o daný čas.

S tím také přichází první úskalí této metody. Pokud výpočet pohnutí hry o 20ms trvá 40ms, hra přehltí CPU a bude hodně práce hru ukončit a přitom počítač nerestartovat. Na dnešních počítačích to může znít jako vtip, ale stalo se to i mně osobně.

Funkci pro pohyb hry lze vylepšit a napsat ji jako diskrétní simulaci. Při použití této metody bych to doporučoval udělat. Různé objekty ve hře lze psát s pevným časem na změnu a navíc může mít každý objekt tento pevný čas jiný.

Diskrétní simulace funguje tak, že si udržuje frontu událostí tříděnou podle času. U každé události je čas, za jak dlouho se má provádět. Když se řekne simulaci, že uběhlo 25ms, podívá se do fronty a provede všechny události, které se mají stát v následujících 25ms.

Událost může být například žádost autíčka o pohyb. Autíčkem se tedy pohne a pokud se bude chtít autíčko v budoucnu opět pohnut, zařadí se znovu do fronty událostí a samo si řekne, za jak dlouho se má jeho funkce pro pohyb znovu zavolat.

Tento způsob práce s časem i s diskrétní simulací byl použit v Kuličovi 2.

7.4 Příklad 7

V posledním příkladu k SDL vytvoříme obdobu příkladu 5 z Allegra. Vytvoříme pohybující se rybu, ale tentokrát s proměnným časem.

7.4.1 sdl_6.c

Budeme potřebovat tři proměnné. Jednu pro čas poslední aktualizace, jednu pro aktuální čas a jednu pro rozdíl časů.

Čas aktualizace a aktuální čas nastavíme před začátkem herní smyčky. V herní smyčce zkontrolujeme aktuální čas. Pokud je jiný, zjistíme jejich rozdíl. Nezapomeneme program ošetřit proti příliš velkému rozdílu. Ten může být způsoben například přetížením počítače nebo uspáním programu.

```
int time1, time2; // mereni casu
int dtime;        // rozdil casu

time1 = time2 = SDL_GetTicks();

// herni smycka
while ( EventLoop() &&  (key[SDLK_q] == 0)) {

   // zjistime novy cas
   time1 = SDL_GetTicks();

   // pokud se zmenil, provedeme prekresleni
   if ( time1 - time2 > 0 ) {

      // zjistime rozdil casu
      if ( (dtime = time1-time2) > 100) dtime = 100;

      // ulozime cas posledni aktualizace
      time2 = time1;

      MoveGameWorld(dtime);

      DrawGameWorld();
   }
}
```

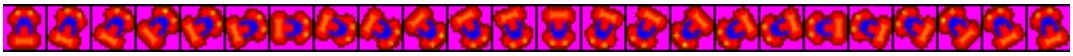
7.5 Srovnávací tabulka

Konstantní čas	**Proměnný čas**
přibližná rychlost	přesná rychlost
na pomalých počítačích pomalé	na pomalých počítačích trhané
užívá jen tolik času CPU, kolik potřebuje	užívá všechen dostupný čas CPU
na rychlejším počítači bez změn	na rychlejším počítači více FPS
jednodušší funkce na pohyb	komplikovanější funkce na pohyb

8 Několik rad na závěr

Na závěr si ještě řekneme několik rad, které mohou pomoci zrychlit výpočet hry.

- Matematické funkce `sin`, `cos`, atd. bývají hodně pomalé. Navíc počítají s přesností, kterou hra zdaleka nepotřebuje. Bývá proto dobré si hodnoty těchto funkcí na začátku hry předpočítat. Allegro už předpočítané tabulky pro goniometrické funkce má.
- Je zbytečné používat proměnné typu `double`. Mají příliš velkou přesnost a počítání s nimi je pomalé. Pro počítání s desetinnými čísly je dobré si vytvořit vlastní typ s pevnou desetinnou čárkou. V Allegru se takový typ jmenuje `fixed`.
- Funkce pro otáčení bitmapy jsou o hodně pomalejší, než funkce pro obyčejné kopírování. Proto některé programy místo jednoho obrázku autíčka načtou obrázků víc, ale každý je jinak otočený. Sice to zrychlí hru, ale stojí to paměť.



Obrázek 5: Roztočené autíčko ze hry SpeedBusters

- Je zbytečné, snažit se kreslit na obrazovku úplně vše. I to, co tam vůbec není. Je dobré umět rychle zjistit, že nějaký objekt hry je úplně mimo obrazovku a vůbec se ho nepokoušet kreslit, protože to je jen ztráta času. Je ještě lépe, pokud se to podaří zjistit o více objektech najednou. Tato věc se hodně řeší zejména ve 3D grafice, kde dobré ořezávaní scény může zrychlit její renderování opravdu výrazně.
- Když bude hra umět speciální grafické efekty, bývá dobré nechat hráči možnost je vypnout. Hra pak bude hratelná i na slabším počítači.
- Pro urychlení vývoje hry a zachování více vlasů na hlavě je šikovné spouštět hru v okně a ne přes celou obrazovku. Zbavit se zamrzlého okna je snadné, ale zbavit se zamrzlé obrazovky bez restartu je mnohdy neřešitelný úkol[20]. Navíc program v okně lze ladit debuggerem, ale program přes celou obrazovku ladit nelze[21].

Děkuji, že jste dočetli až do konce, a přeji hodně radosti při vlastní tvorbě.

[20] V Linuxu velice spolehlivě zabírá `kill -9` z konzole.
[21] Pokus o ladění většinou způsobí zamrznutí obrazovky.

9 Dodatek A - animace

V prvním dodatku si ukážeme, jak udělat jednoduchou animaci. Celý trik animace spočívá v tom, že se pohyb naseká na několik obrázků a ty se v rychlém sledu na obrazovce mění. Jde o podobnou techniku jako při pohybu. SDL ani Allegro přímo nemají žádný typ animované bitmapy nebo něco podobného. Naštěstí je ale vytvoření animace velice snadné[22].

Místo jednoho obrázku se zobrazovaným zvířátkem budeme mít více obrázku, které při naskládání za sebe vytvoří iluzi plynulého pohybu. Je možné jednotlivá políčka animace dát do jedné větší bitmapy nebo nasekat do samostatných souborů. Nevím o žádné velké výhodě či nevýhodě jedné z těchto možností. Používal jsem obě.

Při skládání do jedné bitmapy je možné kolem jednotlivých okének stejně jako je na obrázku s roztočeným autíčkem. Obvykle to bývá výhodné, protože se snáze zaměří přesně výsek a i grafik se snáze do mřížky trefí. Nevýhodou skládání do mřížky je možné zpomalení, protože obrázky v mřížce nebývají tak hezky zarovnané na násobky dvou.[23] Já jsem při skládání obrázků do jednoho souboru vždy používal mřížku.

9.1 Příklad 7

V našem příkladu si ukážeme obrázky naskládané do jedné bitmapy, aby v adresáři s příklady nebylo příliš mnoho souborů. Z důvodu mé lenosti a ohlasu čtenářů se omezím jen na příklad v SDL. Velice bych doporučil při čtení následující ukázky implementace prohlédnout si soubor `ptak.bmp`.

9.1.1 sdl_7.c

Budeme potřebovat obrázek s ptákem a rect pro urční pozice, kam se má kreslit. Pro animaci bude nutný rect, kterým určíme výsek v bitmapě, který představuje aktuální okénko animace a počítadlo animace.

```
SDL_Surface *ptak; // obrazek ptaka
SDL_Rect   pozice; // pozice ptaka
SDL_Rect   anim_rec;   // obdelnik v animaci
int        anim;   // aktualni cislo animacniho okenka
```

Inicializace proměnných se souřadnicemi.

```
// priprava rectu
anim_rec.x = 0;
anim_rec.y = 0;
anim_rec.w = pozice.w = ptak->w;
anim_rec.h = pozice.h = 68;

//  pozice na obrazovce
pozice.x = 50;
pozice.y = 250;

anim = 0;
```

Nehezkou konstrukcí si zajistíme minimální dobu mezi posunem animace minimálně 200ms. Z této konstrukce si rozhodně příklad neberte - je zde jen pro jednoduchost. Obecně platí, že animace je závislá na čase a je ho potřeba při animaci trochu hlídat.

[22] Pro programátora. pro grafika to může být patlačka.

[23] Může být, že obrázky velikostí mocniny dvou či násobky dvou se kopírují rychleji. Mohou se počítači snáze zaměřovat...

```
time1 = SDL_GetTicks();
time2 = SDL_GetTicks();


// nejaka smycka zprav
while ( EventLoop() ) {

// zjistime minimalni spozdeni 200 ms
while ( time1 - time2 < 200 )  time1 = SDL_GetTicks();
time2 = time1;
```

Posuneme čislo aktuálního políčka animace a při dosažení posledního políčka (šetého) začneme znovu od začátku.

Okénka animace máme naskládané pod sebe. Proto při přepočítávání měníme pouze y-ové souřadnice výřezu. Šířka, výška i x-ová souřadnice jsou stále stejné.

Je to snadné, že?

```
// vymazeme obrazovku do svetle modre
SDL_FillRect(screen, NULL,  SDL_MapRGB( screen->format, 80, 180, 250));


// posuneme cykl animace a pripadne zacyklime animaci
if ( ++anim > 6 ) anim = 0;

// prepocitame aktualni obelnik podle animace
anim_rec.y = 69 * anim;


SDL_BlitSurface(ptak, &anim_rec, screen, &pozice);

// aktualizujeme obrazovku
SDL_Flip(screen);
}
```

10 Dodatek B - omezení pohybu bitmapou

V dalším dodatku se podíváme na omezení pohybu podle bitmapy. O co jde? Když máme nějakou 2D hru, ne vždy se nám hodí, aby byl herní plán nakostičkován a chceme aby byl hezky vykreslený.

V tom případě se nám může hodit omezení pohybu bitmapu. Celý trik spočívá v tom, že máme k bitmapě (třeba podkladu) ještě jednu stejně velkou. Na ní ale není žádný pěkný obrázek. Jsou na ní jednou barvou spojené plochy, které mají stejný "význam".

Například cestu nakreslíme bíle, mimo cesty kam se ještě můžeme nakreslit třeba zeleně a kam už se nemůže nakreslíme černě.

Když budeme chtít zjistit, na čem panáček stojí, jednoduše se podíváme do druhé bitpamy na pixel, kde stojí. Když bude bílý, stojí panáček na cestičce, když bude pixel zelený, panáček stojí mimo cestičku. Nemělo by se stát, aby panáček stál na černém pixelu. Tomu budeme bránit tak, že než někam panáčkem pohneme, podíváme se, zda by pixel pod ním náhodou nebyl černý. Tedy na černé pixely panáčka ani nepustíme[24]. Tato technika byla použita ve hrách Kulič a Racer.

To je celé. Snadné, že? Pokud budeme chtít, můžeme si to ještě trošku vylepšit. Například když budeme mít auto. Auto stojí vetšinou na čtyřech kolech. Tedy abychom správně určili povrch, po kterém auto jede, musíme se podívat na 4 pixely - pro každé kolo jeden. S případnou kolizí takového auta je to ještě zajímavější. I když jsou kola ještě na dobrém podkladě, auto už může být dávno v kolizi, protože kapota kola trochu přesahuje. Na kolizi je pak lepší použít několik pixelů, které jsou "na kraji" auta, než kola.

Malý trick z Kuliče. Tam se obrysy neprůchodných předmětů nekreslili přesně podle mapy, ale udělali se širší. Přesněji udělali se širší o polovinu poloměru panáčka. Pak stačilo testovat jen prostřední bod panáčka místo vícebodového systému ala předchozí příklad.

10.1 Příklad 8

V tomto příkladě si ukážeme velice jednoduché omezení panáčka. Bude nás zajímat jen (ne)průchodnost. Abych přecijen nenadržoval jen SDL, bude tento příklad pro změnu jen v Allegru.

10.1.1 allegro_8.c

Z celého programu je pro nás opravdu zajímavá jen funkce `MoveGameWorld()`

Ve funkci si vytvoříme dočasné proměné pro uchování nové pozice panáčka.

```
// souradnice Jaka
int x = 0, y = 0;


void MoveGameWorld()
{
        // pro vypocet novych souradnic
        int new_x = x, new_y = y;

        // aktualizace klavesnice
        poll_keyboard();

        // posun
        if ( key[KEY_UP] ) new_y--;
        if ( key[KEY_DOWN] ) new_y++;
        if ( key[KEY_LEFT] ) new_x--;
        if ( key[KEY_RIGHT] ) new_x++;
```

[24] Nebo se o to alespoň budeme pokoušet :-)

Tuto novou pozici nejprve otestujeme, zda je na ní možné vstoupit a pak teprve panáčka posuneme.

```
        // test, zda v masce ostrova je na souradnici new_xy neni cerva
        //- pak by tam neslo jit.
        if ( getpixel(ostrov_maska, new_x, new_y) != 0 ) {
                x = new_x;
                y = new_y;
        }


        // omezeni Jaka na obrazovku, aby nahodou utekl.
        if ( x < 0 ) x += SCREEN_X;
        if ( y < 0 ) y += SCREEN_Y;
        if ( y > SCREEN_Y ) y -= SCREEN_Y;
        if ( x > SCREEN_X ) x -= SCREEN_X;

}
```

Za povšimnutí možná stojí ještě funkce `DrawGameWorld()` Při kreslení panáčka musíme jeho bitmapku posunout, aby námi testovaný bod odpovídal jeho nohám.

```
void DrawGameWorld()
{
        //  nakresleni pozadi
        draw_sprite(back, ostrov, 0, 0);

        // nakreslime jaka - posuneme ho, aby x,y urcovalo spodek jeho bot
        draw_sprite(back, jake, x-jake->w/2, y-jake->h+2);

        // aktualizace obrazovku
        draw_sprite(screen, back, 0, 0);
}
```

Obsah

Reference

[1] *Shawn Hargreaves* Allegro manuál, `http://www.talula.demon.co.uk/allegro/`

[2] *Sam Lantinga* SDL manuál, `http://www.libsdl.org/`

[3] *John R. Hall* Programming linux games `http://www.lokigames.com/`

[4] *Bernard Lidický* zdrojové kódy her `http://hippo.nipax.cz/`