SDL: Hry nejen pro Linux

napsáno pro server root.cz

http://www.root.cz/serialy/sdl-hry-nejen-pro-linux/

Michal Turek

Obsah

SDL #1 - Úvod

V této sérii článků se vám představí knihovna SDL, která slouží pro vývoj her a multimediálních programů, její největší předností je možnost zkompilovat zdrojový kód pro všechny běžně používané operační systémy. Ukázkové programy budou napsány v jazyku C/C++ a zkompilovatelné minimálně pod operačními systémy GNU/Linux a MS Windows.

SDL #2 - Instalace SDL

V druhé části série si ukážeme, jak nainstalovat SDL a dále budou uvedeny "step-by-step" návody na vytvoření SDL projektů v gcc, MS Visual C++ a Dev-C++.

SDL #3 - Inicializace SDL programu

V první části článku se podíváme na konvenci názvů SDL funkcí a speciální datové typy, které SDL přináší. V druhé části bude popsána inicializace a deinicializace SDL.

SDL #4 - Vytvoření okna

V minulém dílu jsme si dopodrobna vysvětlili inicializaci SDL, ale ještě něco málo zbylo - nastavení vlastností a vytvoření okna. Jak brzy zjistíme, v porovnání s např. Win32 API je tato činnost v SDL mnohem jednodušší.

SDL #5 - Zobrazování grafiky

Dnes se podíváme na grafické funkce poskytované knihovnou SDL. Vzhledem k rozsáhlosti tohoto tématu zde budou uvedeny pouze nejzákladnější věci, podrobnostem se budeme věnovat až v následujících dílech.

SDL #6 - Operace se surfacem

V tomto dílu budeme dále rozvíjet naše znalosti o SDL grafice. Předvedeme si například, jak vyplnit surface barvou, jak docílit toho, aby určitá barva byla transparentní (průhledná), jak nastavit průhlednost i takového surface, který neobsahuje alfa kanál, a další užitečné věci.

SDL #7 - Přímý přístup k pixelům, kurzory

Tentokrát se ponoříme trochu více do hloubky, popíšeme si SDL grafické struktury a tyto znalosti následně využijeme k přímému přístupu k pixelům obrázku. V závěru budeme také měnit kurzor myši.

SDL #8 - OpenGL

Díky přímé podpoře OpenGL umožňuje SDL renderovat i 3D grafické objekty, které se staly nepsaným standardem naprosté většiny dnešních her. Tentokrát se tedy budeme věnovat podpoře OpenGL v SDL.

SDL #9 - Výstup textu pomocí SDL_ttf

V dnešním dílu bude popsána knihovna SDL_ttf, která slouží pro výpisy textů do scény. Se zobrazením textů a především s českými znaky bývá někdy potíž, nicméně použití SDL_ttf je velice jednoduché a naprosto bezproblémové.

SDL #10 - Komunikace se správcem oken, úvod do událostí

Seriál se přehoupl do druhé desítky, příště už na počítání přestanou stačit prsty ;-). Ale ještě než se tak stane, probereme si komunikaci aplikace se správcem oken, což v sobě zahrnuje změnu titulku okna, minimalizaci, přepínání do/z fullscreenu a několik dalších věcí. Ke konci bude také přidán lehký úvod do zpracování událostí.

SDL #11 - Fronta událostí

Na konci minulého dílu jsme nakousli základní práci s událostmi, dnes budeme pokračovat. Tento článek je primárně věnován práci s frontou událostí, ale jelikož ještě nevíme nic o unionu SDL_Event, bude částečně probrán i on.

SDL #12 - Klávesnice

Pravděpodobně nejpoužívanějšími vstupními zařízeními počítače jsou klávesnice a myš, v našem seriálu začneme právě klávesnicí. Podíváme se na ni jak z událostního pohledu, tak "přímým" přístupem a uděláme první krok k interaktivním hrám.

SDL #13 - Myš

Na řadě je další vstupní zařízení, tentokrát se jedná o myš. Opět se budeme věnovat, jak událostem, tak přímému přístupu.

SDL #14 - Joystick

Joysticky, kniply, páky a jiné ovladače bývají nedílnou součástí většiny her, hlavně simulátorů. Tento díl bude věnován právě jim.

SDL #15 - Ostatní události

V dnešním dílu o knihovně SDL dokončíme popis událostního systému. Budeme se mimo jiné věnovat změnám velikosti okna, jeho aktivacím a deaktivacím, posílání uživatelských zpráv a dalším věcem, které ještě zbývá probrat.

SDL #16 - Časovače a práce s časem

V dnešním díle se podíváme na systémové časovače a funkce pro práci s časem. Na konci budou také v rychlosti zmíněny rychlostní optimalizace včetně výpočtu FPS.

SDL #17 - Zvuky a hudba

V dnešním dílu o knihovně SDL začneme nový tematický celek, budou jím zvuky a hudba, které přinesou konec všem tichým aplikacím. Na první pohled by se mohlo zdát, že si musí naprostou většinu funkčnosti napsat programátor sám, nicméně je možné používat již hotový mixer v podobě rozšiřující knihovny SDL_mixer, který odstraní většinu námahy.

SDL #18 - Konverze zvuků, knihovna SDL_sound

V tomto díle konverzemi zvuků dokončíme popis funkcí, které SDL poskytuje pro audio. Druhá část článku bude věnována rozšiřující knihovně SDL_sound, která slouží pro dekódování zvuků z .MP3, .MID, .OGG a dalších běžně rozšířených typů souborů.

SDL #19 - Přehrávání zvuků pomocí SDL_mixer

Vše, co se týká SDL audio funkcí, už máme probráno, takže se zkusíme podívat na rozšiřující knihovnu SDL_mixer. Knihovna SDL_mixer poskytuje snadno použitelné funkce pro mixování zvuků a hudby. Je vhodná obzvlášť pro ty, kterým připadá standardní SDL audio API příliš nízkoúrovňové a strohé.

SDL #20 - Hudba a efekty

Ve 20. díle dokončíme popis knihovny SDL_mixer. Budeme se bavit především o hudbě a speciálních efektech, jako je nastavení rozdílné hlasitosti levého a pravého kanálu nebo simulace ztišení vlivem vzdálenosti zdroje zvuku od posluchače.

SDL #21 - CD-ROM

Další oblastí knihovny SDL, kterou si popíšeme, bude API pro práci s CD-ROM. Po přečtení tohoto článku byste měli být schopni si vytvořit jednoduchý CD přehrávač, jenž zahrnuje přehrávání a pauzy, listování a pohyb ve skladbách a také vysouvání mechaniky pro vložení nového disku.

SDL #22 - Vícevláknové programování

V dnešním díle o knihovně SDL se budeme věnovat podpoře tzv. vícevláknového programování. Podíváme se na vytváření nových vláken a samozřejmě také jejich synchronizaci, která nikdy nesmí chybět.

SDL #23 - SDL_RWops, SDL_Overlay, na co se zapomnělo

V dnešním, závěrečném, díle o knihovně SDL se pokusím shrnout všechny věci, na které jsem během psaní seriálu pozapomněl popř. kterým jsem se z důvodu mé neznalosti nevěnoval pozornost. Mimo jiné se budeme věnovat SDL_RWops, YUV video overlay, nahrávání sdílených knihoven za běhu aplikace a proměnným prostředí.

Úvod

V této sérii článků se vám představí knihovna SDL, která slouží pro vývoj her a multimediálních programů, její největší předností je možnost zkompilovat zdrojový kód pro všechny běžně používané operační systémy. Ukázkové programy budou napsány v jazyku C/C++ a zkompilovatelné minimálně pod operačními systémy GNU/Linux a MS Windows.

Základní informace o SDL

Počátky knihovny Simple DirectMedia Layer (SDL) ukazují ke společnosti Loki Entertainment Software, která se zabývá portováním her do operačního systému GNU/Linux, a jejímu hlavnímu programátoru Samu Lantingovi. Byla navržena jako obecné nízkoúrovňové API (aplikační programové rozhraní) pro tvorbu her a obecně multimediálních aplikací. Z velké části zastřešuje funkce operačních systémů, a tím umožňuje téměř stoprocentní přenositelnost zdrojového kódu. Současná nejnovější stabilní verze je 1.2.8.

SDL obsahuje funkce pro vytvoření okna (včetně fullscreenu) a správu událostí. Dvourozměrná grafika je zahrnuta přímo, 3D grafika je realizována pomocí OpenGL, které má přímou podporu. SDL dále umožňuje práci s audiem, CD-ROM a časovači, pokročilejší programátory jistě potěší podpora vícevláknového programování.

Jak už plyne z názvu (Simple...), je tato knihovna relativně malá. V jádru obsahuje pouze základní funkcionalitu, díky čemuž je přehledná a nezahrnuje programátora žádným gigantickým API. Vše "navíc" poskytují různé nadstavby, např. SDL_image pro nahrávání obrázků (samotné SDL umí nahrát pouze formát BMP), SDL_sound a SDL_mixer pro zvuky, SDL_ttf pro truetype fonty, SDL_net pro síťování a další. V nejhorším případě musí programátor vše potřebné dotvořit sám, nicméně v naprosté většině případů už to někdo řešil před ním, stačí hledat.

Operační systémy a programovací jazyky

V současnosti je SDL portováno do operačních systémů GNU/Linux, MS Windows, BeOS, MacOS Classic, MacOS X, FreeBSD, OpenBSD, BSD/OS, Solaris, IRIX, a QNX. Dále je ho možno najít na Windows CE, AmigaOS, Dreamcast, Atari, NetBSD, AIX, OSF/Tru64 a SymbianOS, ale tyto systémy zatím nejsou oficiálně podporovány.

SDL je napsáno v jazyce C a samozřejmě funguje i v C++. Může být však používáno i v dalších jazycích. Jsou jimi Ada, C#, Eiffel, Erlang, Euphoria, Guile, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python a Ruby. Všechny příklady v tomto seriálu budou napsány v jazyce C nebo C++. Pokud vás zajímají jiné, odkazy na implementace naleznete na domovské stránce SDL.

Licence

SDL je k dispozici pod licencí GNU Lesser General Public License (GNU LGPL) verze 2 nebo novější. Podrobnosti ohledně licencování naleznete na licenční stránce SDL nebo přímo v textu licence.

Všechny ukázkové programy k těmto článkům budou šířeny, pokud výslovně nebude uvedeno jinak, pod licencí GNU General Public License (GNU GPL) verze 2 nebo novější.

Výhody a nevýhody

Hlavní výhody už byly popsány výše, jsou jimi především přenositelnost, jednoduchost, rychlost, flexibilita...

Co se týče nevýhod, existuje asi jen jediná. Dokumentace je sice celkem kvalitní, ale začíná být trochu zastaralá (z roku 2001) - neobsahuje popis některých nově přidaných vlastností. Dá se to však kompenzovat pročtením hlavičkových souborů, které jsou hodně a dobře komentované, dostupností zdrojových kódů a spoustou ukázkových programů. U online dokumentace je také spousta příspěvků přidaných uživateli (ve formě diskuze/fóra u každé stránky), které také často pomohou.

Kde lze SDL získat?

Adresa http://www.libsdl.org/ je prvním místem, které by měl programátor, hledající cokoliv ohledně SDL, navštívit. Lze zde nalézt naprosto vše, včetně dokumentace, tutoriálů, FAQ, souborů pro download, stovek aplikací a knihoven využívajících SDL (většinou včetně zdrojových kódů) a spousty dalších věcí.

Samotné SDL bývá také standardně u naprosté většiny Linuxových distribucí, u jiných operačních systémů bude nutné stahovat. Pokud plánujete vývoj nebo kompilaci programů, jsou nutné devel balíčky, které obsahují hlavičkové soubory, dynamické knihovny, zdrojové kódy, dokumentaci a několik ukázkových programů. Runtime knihovny jsou pouze pro spouštění již zkompilovaných programů.

Instalace SDL

V druhé části série si ukážeme, jak nainstalovat SDL a dále budou uvedeny "step-by-step" návody na vytvoření SDL projektů v gcc, MS Visual C++ a Dev-C++.

Instalace SDL

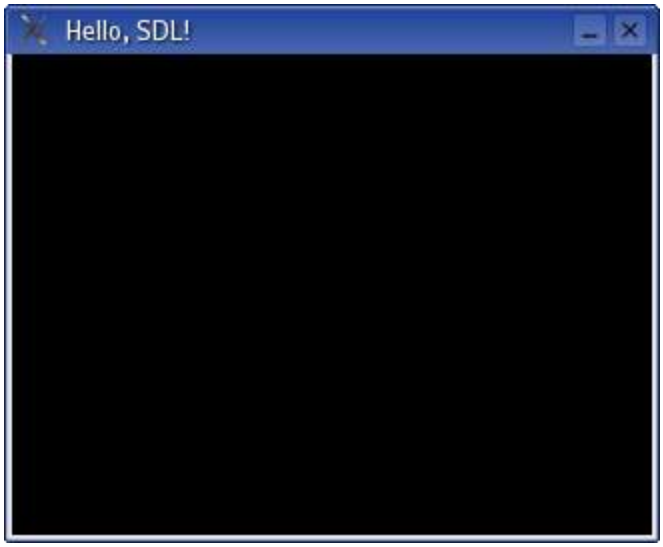
Jak už bylo zmíněno na konci minulého dílu, v Linuxových distribucích bývá SDL standardně přítomno, ale pravděpodobně bude nutné doinstalovat balíčky pro vývoj (devel). U jiných operačních systémů, při požadavku nejnovější verze, či ruční kompilaci lze stahovat z download stránky webu SDL.

V Linuxu se instalace ze zdrojových kódů provádí klasicky pomocí ./configure; make; make install, ve Windows je nejjednodušší cestou vzít předkompilovanou dynamickou knihovnu SDL.dll a zkopírovat ji buď do adresáře C:\Windows\System32\, nebo ke každému vytvářenému projektu zvlášť. Ať už používáte jakýkoli operační systém, nikdy byste neměli zapomenout přiložit k vašemu projektu také informační soubor README-SDL.txt.

Ukázkový program

Vzhledem k tomu, že se při vytváření nového programu začíná vždy založením projektu, budeme tak postupovat i my. Na popis zdrojového kódu se však vzhledem k místu nedostane, vše bude probráno až v následujících dílech.

Velice jednoduchý ukázkový program vytvoří prázdné okno a poté bude čekat na stisk klávesy ESC, tím se ukončí. Nic extra efektního, ale alespoň budeme mít kontrolu, že jsme SDL dokázali zprovoznit.



gcc

Pokud je SDL nainstalováno, měl by jít zdrojový kód zkompilovat například následovně

```
$ g++ -o sdl02 sdl_02.cpp `sdl-config --cflags --libs`
```

Výše uvedený příkaz sdl-config se nainstaloval automaticky se SDL a slouží především k určení cest k hlavičkovým souborům a knihovnám. Před vlastním spuštěním gcc bude obsah části ve zpětných apostrofech proveden shellem a nahrazen do výsledné formy (na mém systému)

```
$ g++ -o sdl02 sdl_02.cpp -I/usr/include/SDL -D_REENTRANT
-L/usr/lib -lSDL -lpthread
```

Mimochodem, všechny volby sdl-config lze získat spuštěním bez parametrů

```
$ sdl-config
Usage: sdl-config [--prefix[=DIR]] [--exec-prefix[=DIR]]
[--version] [--cflags] [--libs] [--static-libs]
```

Visual C++ (6.0)

Aby IDE vědělo, kde má hledat hlavičkové a knihovní (LIB) soubory, je nejprve nutné přidat v menu Tools -> Options -> Directories absolutní cesty k podadresářům include a lib z rozbaleného archivu SDL-devel-1.2.8-VC6.zip.

Dále se vytvoří nový Win32 Application projekt popř. Win32 Console Application projekt, pokud je požadavkem i výstup do konzole. V menu Project -> Settings -> C/C++ -> Code Generation se v listboxu Use run-time library navolí Debug Multithreaded DLL (pro Debug verzi programu) nebo Multithreaded DLL (pro Release verzi programu). Toto se musí vykonat u každého nově vytvářeného projektu, jinak jeho kód nepůjde zkompilovat.

Zbývá přilinkovat knihovny SDL.lib a SDLmain.lib, to lze udělat buď přes nabídky ve vlastnostech projektu, nebo připsáním následujících dvou řádků ke kódu.

```
#pragma comment (lib, "SDL.lib")
#pragma comment (lib, "SDLmain.lib")
```

Bloodshed Dev-C++ (4.9.9.1)

Podobně jako u Visual C++ je nutné v menu Nástroje -> Nastavení kompilátoru -> Adresáře nastavit cesty k hlavičkových a knihovním souborům. Na tomto místě je nutné poznamenat, že devel archiv pro Visual C++ je v Dev-C++ nepoužitelný, pro něj slouží SDL-devel-1.2.8-mingw32.tar.gz. Dynamická knihovna SDL.dll je už ale samozřejmě společná.

Po vytvoření konzolového projektu se v menu Projekt -> Vlastnosti projektu -> Parametry přidají knihovny -lmingw32, -lSDLmain a -lSDL (v tomto pořadí). V případě, že bude na konec seznamu přidáno i -mwindows, nebude se zároveň s aplikací zobrazovat konzolové okno.

Jiné operační systémy a kompilátory

K jiným operačním systémům ani vývojovým prostředím nemám bohužel v současné době přístup. Pokud v nich máte se zprovozněním SDL problémy, mohl by vám pomoci SDL FAQ nebo klasicky Google. Také můžete zkusit diskuzi níže, třeba se najde někdo chytrý...

Makefile pro tyto články

Aby se nemuseli permoníci, co nosí pakety po síti, příliš namáhat, budou kompletní projekty pro všechna testovaná vývojová prostředí pouze u tohoto článku. V příštích dílech bude přikládán pouze jednoduchý ručně psaný Makefile, který může při více souborech se zdrojovými kódy hodně věcí zjednodušit. Celý program se pak zkompiluje pouze zapsáním jediného make do příkazové řádky. Vývojová prostředí obsahují funkci přidání souborů do projektu, takže u nich by se neměly vyskytnou žádné větší problémy.

Inicializace SDL programu

V první části článku se podíváme na konvenci názvů SDL funkcí a speciální datové typy, které SDL přináší. V druhé části bude popsána inicializace a deinicializace SDL.

Konvence názvů SDL funkcí

Knihovna SDL má jen jednoduchou konvenci pro pojmenování svých funkcí, v podstatě se jedná pouze o předponu SDL_. Jako příklad lze uvést jakoukoli funkci, např. inicializační SDL_Init() se musí zavolat na začátku naprosto každého programu, který využívá služeb knihovny SDL.

Za předponou SDL_ se dále může nacházet WM_ nebo GL_, které označuje funkci poskytující operace vztahující se ke správci oken (Window Manager) popř. týkající se knihovny OpenGL. Jako příklad lze uvést příkaz SDL_WM_ToggleFullScreen(), jenž přepíná aplikaci mezi režimem okno/fullscreen, a SDL_GL_SwapBuffers() sloužící pro výměnu předního a zadního bufferu po vykreslení OpenGL scény.

Z předchozích dvou příkladů si lze všimnout, že těla jmen funkcí začínají velkým písmenem a pokud se skládají z více slov, jsou počáteční písmena jednotlivých slov velká.

SDL datové typy

Pro dosažení co největší přenositelnosti kódu definuje SDL své vlastní datové typy, které se při deklaraci proměnných doporučuje upřednostňovat. S jejich použitím se nestane, že u jiného kompilátoru nebo systému, než na kterém probíhá hlavní vývoj, bude mít některý datový typ programovacího jazyka jiný rozsah. Klasickým příkladem je šestnácti a třiceti dvou bitový int. SDL definuje následující datové typy:

Jazyk C	**SDL**
int	SDL_bool (SDL_FALSE, SDL_TRUE)
unsigned char	Uint8
unsigned short	Uint16
unsigned int	Uint32
unsigned long long	Uint64
signed char	Sint8
signed short	Sint16
signed int	Sint32
signed long long	Sint64

Je nutné podotknout, že 64-bitový int nemusí být podporován všemi platformami.

Hlavičkové soubory

Při vývoji stačí většinou vkládat pouze hlavičkový soubor SDL.h, jiné se používají spíše ve speciálních nebo výjimečných případech. Osobně jsem se dále setkal jen se SDL_opengl.h, který řeší umístění knihovny OpenGL na různých platformách. Např. v MacOS je k ní jiná cesta než ve Windows a Linuxu.

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

V zájmu přenositelnosti by také měla být zachována uvedená velikost písmen - tedy SDL velkými a malé h, aby ho byl case sensitive operační systém schopen najít. I když se to nezdá, jedná se o docela častý problém Windows programátorů, kteří, když se onehdy rozhodnou portovat svůj jinak naprosto správný program, stráví tři hodiny nadáváním na ten každý-si-doplňte-své-slovo Linux ;-).

Někdy se lze také setkat s vkládáním hlavičkového souboru jako SDL/SDL.h, ale tento způsob spíše vytváří problémy (/ a \ lomítko), než něčemu pomáhá. Cestu k hlavičkovým souborům lze nastavit ve vývojovém prostředí.

Vstup do programu

Že první funkcí, kterou volá operační systém při spouštění programu, je main(), ví jistě každý programátor. I přesto, že je vytvářena MS Windows aplikace, měla by být tato funkce upřednostněna před WinMain(). Před samotným spuštěním main () SDL provádí ještě určité inicializace.

Pokud je z nějakého důvodu WinMain() nutná, podívejte se do souboru src/main/win32/SDL_main.c ve zdrojových kódech SDL, abyste věděli jaký druh dodatečné inicializace ještě potřebujete, aby SDL pracovalo tak, jak má.

Některé "exotické" kompilátory také mohou mít problémy s formátem zápisu main(), a proto by měla být vždy deklarována takto

```
int main(int argc, char *argv[])
```

Inicializace SDL

SDL se inicializuje voláním funkce

```
int SDL_Init(Uint32 flags);
```

která při úspěchu vrátí hodnotu 0 a při neúspěchu -1. Parametr flags specifikuje, co všechno se má inicializovat. Lze předat symbolické konstanty z následující tabulky nebo jejich binárně OR-ovanou kombinaci.

Symbolická konstanta	**Inicializuje se...**
SDL_INIT_VIDEO	grafika
SDL_INIT_AUDIO	zvuky
SDL_INIT_TIMER	časovače
SDL_INIT_CDROM	CD-ROM
SDL_INIT_JOYSTICK	joystick
SDL_INIT_EVERYTHING	vše
SDL_INIT_NOPARACHUTE	nereagovat na chybové signály (SIGSEGV ap.)

Linux a BeOS podporují také parametr SDL_INIT_EVENTTHREAD, který, pokud bude předán do SDL_Init(), způsobí, že smyčka hlídající události bude běžet asynchronně ve vlastním vláknu.

Pozn.: Pokud budete mít problémy s laděním SDL aplikace ve Visual C++ debuggeru, zkuste nastavit flag SDL_INIT_NOPARACHUTE.

Inicializace dalších subsystémů

Kdykoli po hlavní inicializaci lze pomocí následující funkce inicializovat i další subsystémy. Chování obou rutin je analogické.

```
int SDL_InitSubSystem(Uint32 flags);
```

Kontrola inicializace subsystémů

Zjištění, které subsystémy byly inicializovány a které ne se provede pomocí

```
Uint32 SDL_WasInit(Uint32 flags);
```

Za parametr flags se předají subsystémy, které se mají otestovat a vrácena je bitová maska subsystémů, které jsou inicializované.

V následujícím příkladu se pokusí aplikace o inicializaci grafického a zvukového subsystému. Pokud se grafiku nepodaří inicializovat, program skončí. V případě nedostupnosti zvuků bude program pokračovat dále bez nich.

```
// Globální proměnná
bool use_audio = true;

// V main()
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) == -1)
{
        fprintf(stderr, "Unable to initialize SDL: %s\n",
                        SDL_GetError());

        Uint32 flags = SDL_WasInit(SDL_INIT_VIDEO | SDL_INIT_AUDIO);

        // Grafika musí být vždy
        if(!(flags & SDL_INIT_VIDEO))
        {
                SDL_Quit();
                return 1;
        }

        // Zvuky používat pouze, pokud jsou dostupné
        use_audio = (flags & SDL_INIT_AUDIO) ? true : false;
}
```

Určení typu chyby

V příkladu výše byla po neúspěšné inicializaci vypisována chybová zpráva, na jejíž konec byl připojen řetězec s upřesněním získaným od SDL. Funkce SDL_GetError() vrátí NULLem ukončený řetězec, obsahující informace o poslední vnitřní chybě SDL.

```
char *SDL_GetError(void);
```

Dále existují ještě dvě funkce, které jsou však určeny spíše pro vývojáře knihovny SDL než pro její uživatele. Pomocí první se nastavuje řetězec s chybou a druhou se maže.

```
void SDL_SetError(const char *fmt, ...);
void SDL_ClearError(void);
```

Pozn.: U výše uvedeného výpisu textu pomocí fprintf() resp. printf() se v Linuxu zobrazí text do konzole, pod MS Windows v nekonzolové aplikaci se ve stejném adresáři, kde je umístěn spuštěný EXE soubor, automaticky vytvoří soubory stderr.txt a stdout.txt.

Deinicializace

Před ukončením programu by měla být vždy zavolána funkce SDL_Quit(), která se postará o veškerý úklid.

```
void SDL_Quit(void);
```

V některých cizích zdrojových kódech se lze též setkat s příkazem atexit(SDL_Quit);, který je zapsán hned za SDL_Init() a který při ukončení programu zavolá SDL_Quit() automaticky. Nicméně každá trochu delší aplikace obsahuje alespoň náznak nějaké ukončovací logiky, lepší je umístit SDL_Quit() tam.

Podobně jako mělo SDL_Init() protějšek v SDL_InitSubSystem() i SDL_Quit() má svůj SDL_QuitSubSystem(). Pokud ale není subsystém ukončován někde uprostřed aplikace, stačí na konci zavolat pouze SDL_Quit() a je uvolněno všechno.

```
void SDL_QuitSubSystem(Uint32 flags);
```

Vytvoření okna

V minulém dílu jsme si dopodrobna vysvětlili inicializaci SDL, ale ještě něco málo zbylo - nastavení vlastností a vytvoření okna. Jak brzy zjistíme, v porovnání s např. Win32 API je tato činnost v SDL mnohem jednodušší.

Vytvoření okna

V nejjednodušším a většinou zcela dostačujícím případě se jedná pouze o volání jediné jednoduché funkce. Ne nadarmo se říká, že SDL nechává programátora koncentrovat se na vývoj vlastní hry, místo toho, aby se staral o složité detaily operačního systému...

```
SDL_Surface *SDL_SetVideoMode(int width, int height,
                int bpp, Uint32 flags);
```

Návratovou hodnotou funkce je při chybě NULL, v ostatních případech ukazatel na SDL_Surface. Tato struktura v SDL představuje základní a v podstatě jediný prostředek pro ukládání pixelů obrázku a informací o něm. V tomto případě se jedná o okno (frame buffer) aplikace. Všechny funkce, které SDL poskytuje pro 2D grafiku, jsou realizovány nad touto strukturou.

Parametry width a height určují šířku klientské oblasti okna, bpp specifikuje barevnou hloubku v bitech. Pokud se vloží hodnota 0, bude použita barevná hloubka aktuálně nastavená v systému. Parametr flags specifikuje vlastnosti okna a lze předat některou z následujících symbolických konstant nebo jejich kombinaci (binární OR).

- SDL_SWSURFACE, SDL_HWSURFACE
  Vytvoří surface v systémové nebo video paměti. Pokud nebudou pixely surface často modifikovány, je lepší je uložit přímo do video paměti, protože se pak bude využívat hardwarové akcelerace.
- SDL_ASYNCBLIT
  Povolí asynchronní aktualizaci surface. Tímto se sice na jednoprocesorových systémech blitting (kreslení jednoho obrázku do druhého) většinou zpomalí, ale na SMP systémech dojde ke zrychlení.
- SDL_ANYFORMAT
  Obyčejně, pokud není požadovaná barevná hloubka video surface dostupná, SDL přistoupí k emulaci. Předáním SDL_ANYFORMAT se tomuto předejte - SDL použije video surface s jakoukoli z dostupných barevných hloubek, ale bez emulace.
- SDL_HWPALETTE
  Poskytne SDL exkluzivní přístup k paletě. Bez tohoto flagu nemusí vždy jít získat barva požadovaná přes SDL_SetColors() nebo SDL_SetPalette().
- SDL_DOUBLEBUF
  Povolí hardwarový double buffering. Veškeré kreslení se pak bude provádět na skrytém/nezobrazeném bufferu. Po jeho ukončení se buffery zamění voláním SDL_Flip(). Tento parametr je validní pouze spolu se SDL_HWSURFACE.
- SDL_FULLSCREEN
  Aplikace nepoběží v okně, ale v celoobrazovkovém režimu. Pokud není z jakéhokoli důvodu změna hardwarového rozlišení možná, bude použito následující vyšší rozlišení a scéna se vycentruje na černém pozadí.
- SDL_OPENGL
  Vytvoří okno s podporou OpenGL. Před vlastním voláním SDL_SetVideoMode() by měly být již nastaveny atributy přes SDL_GL_SetAttribute(), pozdější změna již není možná. Použití OpenGL bude věnován některý z budoucích dílů.
- SDL_OPENGLBLIT
  Má stejný význam jako předchozí parametr, ale s tím rozdílem, že bude zároveň možné provádět SDL blitting. Scéna (2D) může mít alfa kanál a pro aktualizaci musí být použito SDL_UpdateRects().
- SDL_RESIZABLE

Bude možná změna velikosti okna. Při roztahování se bude generovat událost SDL_VIDEORESIZE a může být opět zavoláno SDL_SetVideoMode() s novými rozměry.

- SDL_NOFRAME
  Pokud je to možné, vytvoří okno bez titulku a rámu. V celoobrazovkovém režimu se nastavuje automaticky.

Pozn.: Pokud je funkcionalita poskytovaná daným parametrem z nějakého důvodu důležitá, lze otestovat flagy z vráceného surface.

Získání video surface

Ukazatel na aktuální zobrazený surface lze získat pomocí následující funkce. V případě, že SDL provádí nějaké konverze formátu, bude vrácen veřejně viditelný surface, ne opravdový.

```
SDL_Surface *SDL_GetVideoSurface(void);
```

Zjištění dostupnosti video formátu

Před samotným vytvořením okna může být dobré nejdříve zjistit, zda jsou požadované parametry vůbec dostupné.

```
int SDL_VideoModeOK(int width, int height,
                int bpp, Uint32 flags);
```

Funkce vrací hodnotu barevné hloubky pro nejbližší dostupný mód v závislosti na předané šířce, výšce a vlastnostech, parametry jsou tedy stejné jako u funkce SDL_VideoMode(). Jediným rozdílem je, že nulová barevná hloubka (aktuálně nastavená v systému) není pro tuto funkci validní! Pokud není mód podporován pod žádnou barevnou hloubkou, je vrácena nula.

Seznam dostupných rozměrů okna

Pomocí funkce SDL_ListModes() je programátor schopen nagrabovat všechna dostupná rozlišení pro daný formát pixelů a vlastnosti.

```
SDL_Rect **SDL_ListModes(SDL_PixelFormat *format, Uint32 flags);
```

Návratovou hodnotou je ukazatel na pole obdélníků, které budou navíc seřazené od největších rozměrů po nejmenší. V případě, že bude vrácen NULL nejsou dostupná žádná rozlišení a (SDL_Rect **)-1 oznamuje, že jakékoli rozlišení je v pořádku.

Pokud se za parametr format předá NULL, bude seznam vztáhnut vzhledem k SDL_GetVideoInfo()->vfmt (viz dále). Při zjišťování jsou spíše důležité flagy než formát pixelů. Pokud by bylo například předáno SDL_HWSURFACE, hledalo by se pouze v hardwarově podporovaných módech.

Struktura obdélníku je definována následovně. Atributy x a y specifikují pozici levého horního rohu, w a h rozměry. Jednotky jsou v pixelech.

```
typedef struct
{
        Sint16 x, y;
        Uint16 w, h;
} SDL_Rect;
```

Protože bychom zbytečně zabředli do v tuto chvíli ne zrovna životně důležitých detailů, bude struktura SDL_PixelFormat popsána až někdy v budoucnu.

Zjištění informací o grafickém hardware

Funkce SDL_GetVideoInfo() vrátí read only ukazatel na strukturu SDL_VideoInfo, která obsahuje informace o grafickém hardware. Pokud bude volána před SDL_SetVideoMode(), bude atribut vfmt obsahovat formát pixelů "nejvhodnějšího" video módu. V případě volání po SDL_SetVideoMode() bude obsahovat aktuální formát.

```
SDL_VideoInfo *SDL_GetVideoInfo(void);
```

Struktura SDL_VideoInfo se používá pouze při volání této funkce a je deklarována takto:

```
typedef struct
{
        Uint32 hw_available :1; // Lze vytvořit hardwarové surface?
        Uint32 wm_available :1; // Lze komunikovat se správcem oken?
        Uint32 blit_hw      :1; // Akcelerovaný blitting HW --> HW
        Uint32 blit_hw_CC   :1; // Akcelerovaný blitting s Colorkey
        Uint32 blit_hw_A    :1; // Akcelerovaný blitting s Alpha
        Uint32 blit_sw      :1; // Akcelerovaný blitting SW --> HW
        Uint32 blit_sw_CC   :1; // Akcelerovaný blitting s Colorkey
        Uint32 blit_sw_A    :1; // Akcelerovaný blitting s Alpha
        Uint32 blit_fill    :1; // Akcelerované vyplňování barvou?
        Uint32 video_mem;       // Celkové množství video paměti v kB
        SDL_PixelFormat *vfmt;  // Formát grafického surface
} SDL_VideoInfo;
```

Pozn.: Pokud bude pod X11 paměť nulová a žádné hardwarové akcelerace dostupné, změňte grafický ovladač z X11 např. na DGA (pouze fullscreen a X11). V konzoli definujte systémovou proměnnou SDL_VIDEODRIVER ($ export SDL_VIDEODRIVER=dga) a spusťte program. Podrobnosti lze nalézt v SDL FAQ, sekce Development.

Zjištění video ovladače

Do řetězce specifikovaného parametrem namebuf bude uloženo maximálně maxlen znaků (včetně NULL) se jménem inicializovaného video driveru. Jedná se o jednoduché slovo jako x11, dga, directx nebo windib.

```
char *SDL_VideoDriverName(char *namebuf, int maxlen);
```

Pokud ještě nebyla grafika inicializována (SDL_Init()), bude vráceno NULL.

Gamma funkce obrazovky

SDL, pokud to hardware počítače umožňuje, umí změnit "gamma funkci", která kontroluje jas a kontrast barev zobrazených na obrazovce. Vztažná hodnota je 1.0, tzn. nebudou se provádět žádné změny. Menší než jedna představuje ztmavení, větší než jedna zesvětlení. Meze jsou přibližně 0.1 a 10.0.

```
int SDL_SetGamma(float redgamma, float greengamma, float bluegamma);
int SDL_SetGammaRamp(Uint16 *redtable, Uint16 *greentable, Uint16 *bluetable);
int SDL_GetGammaRamp(Uint16 *redtable, Uint16 *greentable, Uint16 *bluetable);
```

Pomocí druhé uvedené funkce lze pro každý kanál barev předat kompletní tabulku. Jedná se o pole 256 Uint16 čísel, které reprezentuje mapování mezi vstupem a výstupem. Vstup je indexem do pole a výstup udává šestnáctibitovou hodnotu gammy na daném indexu, jejíž velikost je změněna na výstupní přesnot. Pokud se za kanál předá NULL, zůstane nezměněn. Vrácení -1 znamená, že funkce nejsou podporovány.

Ukázkové programy

Vytvoření okna "jednoduchým" způsobem

Kód ukazuje pravděpodobně nejjednodušší (a také nejpoužívanější) vytvoření okna. Tento postup není zrovna flexibilní, ale bývá většinou zcela dostatečný.

Vytvoření okna s ověřováním vlastností

Druhý příklad je už o něco složitější. Pokud nejsou některé parametry vytvářeného okna validní, program se je pokouší upravovat tak dlouho, dokud nejsou použitelné nebo nejdou dále upravit. Zároveň s testy vypisuje na konzoli informace, jaké modifikace právě provádí. Můžete zkusit zadávat různé flagy, záporné velikosti okna a podobně.

Vypsání informací o grafickém hardware

Program nevytvoří žádné okno, ale po spuštění pouze vypíše různé informace o grafickém hardware a pak se ukončí.

Zobrazování grafiky

Dnes se podíváme na grafické funkce poskytované knihovnou SDL. Vzhledem k rozsáhlosti tohoto tématu zde budou uvedeny pouze nejzákladnější věci, podrobnostem se budeme věnovat až v následujících dílech.

Návratové hodnoty funkcí

Většina SDL funkcí zabývajících se grafikou dodržuje pravidlo, že nulová návratová hodnota značí úspěch a mínus jednička neúspěch. Pokud tedy nebude uvedeno jinak, platí u funkcí vracejících int tyto hodnoty. Volání funkcí, které vracejí ukazatele, by měly být ošetřeny klasicky na NULL.

Nahrávání obrázků z disku

SDL umí nahrávat pouze obrázky ve formátu BMP, ale díky knihovně SDL_image, která se už stala defakto jeho součástí, může programátor používat i PCX, GIF, JPG, PNG, TGA, TIFF a další méně známé formáty. Stejně jako celé SDL je i SDL_image šířena pod licencí GNU LGPL a lze ji najít na adrese http://www.libsdl.org/projects/SDL_image/.

Po přilinkování knihovny a vložení hlavičkového souboru SDL_image.h je možné volat funkci IMG_Load(), která vrací surface nahrávaného obrázku. Formát je detekován automaticky podle přípony, hlaviček apod.

```
SDL_Surface *SDL_LoadBMP(const char *file);
SDL_Surface *IMG_Load(const char *file);
```

Ukládání obrázků

SDL kromě nahrávání surface z disku umožňuje i ukládání. Jedná se opět pouze o formát BMP, SDL_image ukládání bohužel neumožňuje.

```
int SDL_SaveBMP(SDL_Surface *surface, const char *file);
```

Uvolnění surface

Ke smazání surface lze použít následující funkci, která se o vše postará.

```
void SDL_FreeSurface(SDL_Surface *surface);
```

Konverze formátu surface

Není to podmínkou, ale pokud bude mít surface obrázku stejný formát, jako má okno, jeho zobrazení bude stát procesor mnohem méně výkonu. Nejlepší ze všeho je ihned po vytvoření surface zavolat funkci SDL_DisplayFormat() (resp. SDL_DisplayFormatAlpha() pro surface s alfa kanálem), která se postará o konverzi. Parametrem je převáděný surface a vrácen je nový surface s požadovaným formátem nebo NULL při neúspěchu.

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
SDL_Surface *SDL_DisplayFormatAlpha(SDL_Surface *surface);
```

Výše uvedené funkce využívají služeb SDL_ConvertSurface(), která je jejich obecnější variantou.

```
SDL_Surface *SDL_ConvertSurface(SDL_Surface *src,
                SDL_PixelFormat *fmt, Uint32 flags);
```

První parametr opět představuje zdrojový surface, druhým je požadovaný pixel formát (většinou surface_vzoru->format) a třetím jsou flagy, které už byly probrány dříve (SDL_SWSURFACE, SDL_HWSURFACE apod.).

Vykreslování

Základem veškerého blittingu je v SDL funkce SDL_BlitSurface(), která vezme pixely ze zdrojového surface src a zkopíruje je do cílového surface dst, jímž je většinou, ale ne vždy, framebuffer okna. Funkce samozřejmě umožňuje specifikovat pozici a rozměry oblasti, díky čemuž nemusí být kopírován celý surface.

Pozn.: Termín blitting obecně označuje kopírování pixelů z jedné paměti do druhé. Nemusí se však jednat pouze o strohé přepsání dat, ale mohou se provádět různé další operace - např. přeskakovat pixely určité barvy, míchat zdrojový pixel s cílovým v závislosti na alfa kanálu (blending) a podobně. SDL_BlitSurface() vykonává všechny tyto operace.

```
int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect,
                SDL_Surface *dst, SDL_Rect *dstrect);
```

Pokud bude za některý z obdélníků předán NULL, bude se pracovat s celým objektem, u cílové oblasti se používá pouze pozice, žádné roztahování nebo zmenšování tedy není možné. Po návratu z funkce bude cílový obdélník obsahovat rozměry oblasti, se kterými se při blittingu pracovalo ve skutečnosti, tato hodnota se bude hodit při aktualizaci okna (viz níže). Výsledek blittingu do značné míry závisí na vlastnostech surfaců, především alfa kanálu a transparentní barvě.

Návratovou hodnotou je klasicky 0 (úspěch) a -1 (neúspěch). Pokud bude u hardwarového surface vráceno -2, byla jeho video paměť ztracena. V takovém případě by měl být surface znovu nahrán/vytvořen. Stává se to během přepínání z celoobrazovkového režimu do okna, pokud SDL využívá služeb DirectX 5.0. Pro podrobnosti odkazuji na SDL manuál.

Pozn.: Možná to bude jen přežitek starších verzí, protože SDL v současnosti přepínání aplikace mezi fullscreenem a oknem pod operačním systémem MS Windows vůbec neumožňuje.

Aktualizace obsahu okna

Aby se zobrazila právě vykreslená scéna, je potřeba vykonat ještě jednu operaci - aktualizovat oblast okna, na kterou se kreslilo. V parametrech funkce se předává neplatný obdélník, jehož žádná část by neměla přesahovat okraje okna - neprovádí se žádné testy a tedy ani ořezávání. Pokud budou předány samé nuly, aktualizuje se celé okno. Tyto funkce by nikdy neměly být volány na zamknutý surface.

```
void SDL_UpdateRect(SDL_Surface *screen,
                Sint32 x, Sint32 y, Sint32 w, Sint32 h);
void SDL_UpdateRects(SDL_Surface *screen,
                int numrects, SDL_Rect *rects);
```

Co se týká druhé uvedené funkce, má stejný význam jako první, kromě toho, že lze specifikovat více obdélníků najednou. Tyto obdélníky však nejsou testovány na vzájemné přesahy, jejich aktualizace probíhají nezávisle.

V případě, že okno používá double buffering (SDL_DOUBLEBUF předaný do SDL_SetVideoMode()), buffery se prohodí voláním SDL_Flip(). Hardware pak počká na vertikální zatmění stínítka monitoru a až poté provede požadovanou operaci.

```
int SDL_Flip(SDL_Surface *screen);
```

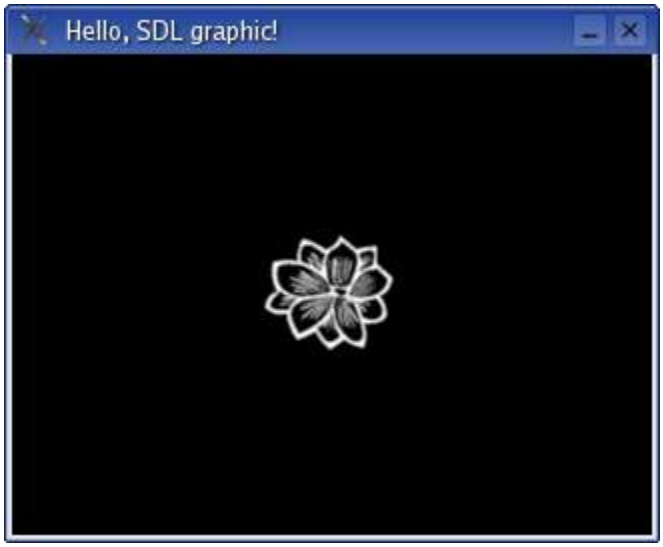
Pokud hardware double buffering nepodporuje nebo není zapnutý, je SDL_Flip() ekvivalentní volání SDL_UpdateRect (screen, 0, 0, 0, 0), čili překreslí se celé okno.

Ukázkové programy

Jelikož by se mnoho kódu z příkladů pokaždé opakovalo, budou obecně použitelné funkce umisťovány do souborů functions.h a functions.cpp. Prozatím bude obsahovat pouze pomocnou funkci na nahrávání obrázků z disku a funkci pro výpočet počtu snímků za sekundu. Animace a pohyby v programu díky FPS poběží stejně rychle na každém počítači.

Hello, SDL graphic!

Asi nejjednodušší program, jaký lze vytvořit na demonstraci použití SDL grafiky. Vykresluje se v něm jednoduchý obrázek, který byl nahrán z disku za použití knihovny SDL_image. Na své centrované pozici zůstává i při roztahování okna.



Objekt odrážející se od okrajů

Druhý příklad je o něco složitější než ten první. Místo statického obrázku je vykreslován dynamický objekt, který se pohybuje po přímce oknem a odráží se od okrajů. Díky tomu, že obsahuje i alfa kanál, jím může prosvítat pozadí.



Ve vnitřního fungování programů existuje jeden rozdíl. U prvního má okno pouze jeden buffer a scéna se po vykreslení aktualizuje pomocí funkce SDL_UpdateRect(). Druhý příklad využívá double buffering (je-li podporován) a scéna se musí aktualizovat voláním SDL_Flip(). SDL_UpdateRect() by v tomto případě nemělo žádný efekt.

Operace se surfacem

V tomto dílu budeme dále rozvíjet naše znalosti o SDL grafice. Předvedeme si například, jak vyplnit surface barvou, jak docílit toho, aby určitá barva byla transparentní (průhledná), jak nastavit průhlednost i takového surface, který neobsahuje alfa kanál, a další užitečné věci.

Vytvoření prázdného surface

Prázdný SDL_Surface se dá v programu vytvořit pomocí funkce SDL_CreateRGBSurface(). První čtyři parametry jistě není třeba popisovat, jsou jimi flagy, šířka, výška a barevná hloubka. Bity masek definují pořadí barevných složek v pixelu, označují tedy, jestli bude obrázek uložen jako RGB, BGR popř. v jiném formátu. U osmi a čtyř bitové barevné hloubky bude alokována prázdná paleta.

```
SDL_Surface *SDL_CreateRGBSurface(Uint32 flags,
                int width, int height, int depth,
                Uint32 Rmask, Uint32 Gmask,
                Uint32 Bmask, Uint32 Amask);

SDL_Surface *SDL_CreateRGBSurfaceFrom(void *pixels,
                int width, int height, int depth, int pitch,
                Uint32 Rmask, Uint32 Gmask,
                Uint32 Bmask, Uint32 Amask);
```

Při vytváření surface je vhodné specifikovat masky v závislosti na pořadí bytů, které se používá na dané platformě (little/big endian). SDL definuje symbolickou konstantu SDL_BYTEORDER, jež se rovná buď hodnotě SDL_LIL_ENDIAN, nebo SDL_BIG_ENDIAN.

Například při vytváření textury pro OpenGL je vhodné podle následujícího příkladu vytvořit pomocný surface, voláním SDL_Blit() do něj pixely zkopírovat, tím se transformují do správného formátu, a až poté vytvořit texturu.

```
SDL_Surface *surface = SDL_CreateRGBSurface(
                SDL_SWSURFACE,
                128, 128, 32,
#if SDL_BYTEORDER == SDL_LIL_ENDIAN
                0x000000FF,
                0x0000FF00,
                0x00FF0000,
                0xFF000000
#else
                0xFF000000,
                0x00FF0000,
                0x0000FF00,
                0x000000FF
#endif
                );
```

Ořezávací obdélník surface

Příkazem SDL_SetClipRect() lze surface "virtuálně ořezat", pro funkce se pak bude chovat, jako by měl definovánu tuto novou velikost. Maximální rozměry mohou být následně obnoveny předáním NULL. Uvedená vlastnost se bere v úvahu, když se do surface kreslí, ne když je kreslen!

Pokud funkce vrátí SDL_FALSE, obdélník neprotínal surface a při vykreslování se tedy nezobrazí nic. Zasahuje-li alespoň část obdélníku do surface, je vráceno SDL_TRUE a při kreslení se bude brát v úvahu oblast průniku.

```
SDL_bool SDL_SetClipRect(SDL_Surface *surface, SDL_Rect *rect);
void SDL_GetClipRect(SDL_Surface *surface, SDL_Rect *rect);
```

Pomocí druhé uvedené funkce lze získat aktuální ořezávací roviny obdélníku.

Specifikace barvy

Kvůli mnoha různým formátům surface - zvláště paletovým - může být výběr barvy komplikovanější, než by se na první pohled mohlo zdát. Naštěstí SDL poskytuje příkazy, které se umí postarat o všechny detaily.

```
Uint32 SDL_MapRGB(SDL_PixelFormat *fmt,
                Uint8 r, Uint8 g, Uint8 b);
Uint32 SDL_MapRGBA(SDL_PixelFormat *fmt,
                Uint8 r, Uint8 g, Uint8 b, Uint8 a);
```

První parametr funkce je formátem pixelů, který daný surface používá (většinou surface->format), a ostatní představují jednotlivé RGB(A) složky. Barva je vrácena jako 32-bitové číslo, jenž je buď přímo požadovanou barvou, nebo, v případě paletového pixel formátu, barvou která se nachází v paletě a nejvíce se blíží požadované.

Opačný směr, tedy získání RGB(A) složek barvy z pixelu, zprostředkovávají funkce

```
void SDL_GetRGB(Uint32 pixel, SDL_PixelFormat *fmt,
                Uint8 *r, Uint8 *g, Uint8 *b);
void SDL_GetRGBA(Uint32 pixel, SDL_PixelFormat *fmt,
                Uint8 *r, Uint8 *g, Uint8 *b, Uint8 *a);
```

Vyplnění surface barvou

Funkce SDL_FillRect() se většinou používá ke změně barvy pozadí okna, ale lze ji použít na libovolný surface.

```
int SDL_FillRect(SDL_Surface *dst, SDL_Rect *dstrect, Uint32 color);
```

První parametr představuje surface, na který bude operace aplikována a druhý omezuje velikost obarvované plochy. Pokud by byl tento obdélník nastaven na NULL, předpokládá se vyplnění celého surface. Color určuje barvu.

Obsahuje-li surface ořezávací obdélník, bude vyplněn pouze jeho průnik s dstrect a dstrect bude nastaven na rozměry vyplněné oblasti. Následující příklad nastaví pozadí okna na červenou barvu.

```
// Červené pozadí okna
SDL_FillRect(g_screen, NULL, SDL_MapRGB(g_screen->format, 255, 0, 0));
```

Nastavení klíčové (průhledné) barvy

Transparentní barva surface se dá nastavit pomocí funkce SDL_SetColorKey(). Při blittingu nebudou pixely této barvy vykresleny.

```
int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);
```

Za parametr flag se při této operaci musí předat symbolická konstanta SDL_SRCCOLORKEY, předáním nuly se transparentní barva zruší. Následujícím příkazem se v surface zprůhlední růžové pixely.

```
SDL_SetColorKey(surface, SDL_SRCCOLORKEY,
                SDL_MapRGB(surface->format, 255, 0, 255));
```

Pokud je flag binárně ORován se SDL_RLEACCEL, bude surface vykreslován s použitím RLE akcelerace. To je výhodné u spojitých oblastí průhledných pixelů (na řádcích). Surface je pro použití RLE akcelerace zakódován při prvním předání do funkce SDL_BlitSurface() nebo SDL_DisplayFormat().

Alfa hodnota surface

Pomocí funkce SDL_SetAlpha() lze nastavit globální úroveň průhlednosti, která bude aplikována na každý pixel surface.

```
int SDL_SetAlpha(SDL_Surface *surface, Uint32 flag, Uint8 alpha);
```

Parametr flag musí být nastaven na SDL_SRCALPHA a může být, stejně jako u předešlé funkce, ORován se SDL_RLEACCEL. Poslední parametr specifikuje úroveň alfy. Nula (SDL_ALPHA_TRANSPARENT) má význam úplné průhlednosti a 255 (SDL_ALPHA_OPAQUE) značí neprůhlednost. Speciální hodnotou je 128, která je určitým způsobem optimalizována, takže blitting bude rychlejší než u jiných hodnot. Při použití této techniky nesmí mít surface alfa kanál, použila by se alfa jednotlivých pixelů.

Nastavení palety

Barvy v paletě osmibitového a čtyřbitového surface se dají nastavit pomocí funkce SDL_SetColors().

```
int SDL_SetColors(SDL_Surface *surface, SDL_Color *colors,
                  int firstcolor, int ncolors);
```

Funkci se předává ukazatel na daný surface, pole barev, první barvu a celkový počet barev. Pokud byly úspěšně nastaveny všechny barvy, je vrácena jednička. Pokud některé byly nastaveny a některé ne, je vrácena nula. V takovém případě by měl programátor zjistit ze surface nově vzniklou paletu. Nejedná-li se o paletový surface, nic se neprovede a je vrácena také nula.

Je-li předaný surface asociován s oknem a byl-li v SDL_SetVideoMode() definován flag SDL_HWPALETTE, vrátí tato funkce vždy jedničku a nastavení palety je vždy garantováno.

V případě, že se jedná o framebuffer s hardwarovým surface, obsahuje vždy dvě palety, logickou (používají ji funkce pro blitting) a fyzickou (používá ji hardware k mapování na obrazovku). Aby je bylo možné specifikovat odděleně, musí mít framebuffer nastaven již zmíněný flag SDL_HWPALETTE. K oddělené specifikaci palet pak slouží funkce

```
int SDL_SetPalette(SDL_Surface *surface, int flags,
                  SDL_Color *colors, int firstcolor,
                  int ncolors);
```

Parametr flags může být nastaven buď na hodnotu SDL_LOGPAL (logická paleta), nebo na SDL_PHYSPAL (fyzická paleta), většinou se modifikuje pouze jedna z nich, čímž se dociluje různých efektů. Volání SDL_SetPalette() s parametrem flags nastaveným na SDL_LOGPAL | SDL_PHYSPAL je ekvivalentem SDL_SetColors().

V SDL manuálu se u popisu těchto funkcí nachází příklad na nastavení palety úrovně šedi.

Ukázkové programy

Vlastnosti surface

Podstata tohoto programu tkví především ve vykreslovací funkci. Surface okna je zmenšen pomocí SDL_SetClipRect() tak, aby u okrajů vznikla deseti pixelová mezera. Poté je vykresleno červené pozadí a do každého rohu stejný obrázek, ale s jinými vlastnostmi.

V levém horním rohu se nachází originál tak, jak byl nahrán z disku, u obrázku vpravo je bílá barva pixelů nastavena na transparentní. Vlevo dole byla nastavena 50% průhlednost a vpravo dole se nachází kombinace obou. Je důležité poznamenat, že obrázek je ve formátu RGB, bez alfa kanálu.



Přímý přístup k pixelům, kurzory

Tentokrát se ponoříme trochu více do hloubky, popíšeme si SDL grafické struktury a tyto znalosti následně využijeme k přímému přístupu k pixelům obrázku. V závěru budeme také měnit kurzor myši.

Struktura SDL_Surface

Každý už jistě ví, že základem veškeré grafiky, kterou poskytuje knihovna SDL, je struktura SDL_Surface. Poprvé jsme se s ní setkali už u funkce SDL_SetVideoMode(), kde představovala framebuffer okna, a následně u všech kreslících funkcí. Obecně může být jakýmkoli úložištěm pixelů. Pro vlastní programování není znalost jejího vnitřního formátu většinou nijak zásadní, nicméně alespoň všeobecná představa se může hodit.

```
typedef struct SDL_Surface
{
        Uint32 flags;
        SDL_PixelFormat *format;
        int w, h;
        Uint16 pitch;
        void *pixels;

        SDL_Rect clip_rect;
        int refcount;

        // + další privátní složky (viz SDL_video.h)
} SDL_Surface;
```

Položka flag může u obecného surface nabývat pouze kombinací hodnot SDL_SWSURFACE, SDL_HWSURFACE a SDL_ASYNCBLIT. Flagy fullscreenu, změny velikosti a podobné jsou dostupné pouze u surface okna.

Hardwarový surface a tedy i hardwarovou akceleraci bývá vhodné používat při blittingu, který se tím výrazně urychlí, naopak při častých modifikacích pixelů (oheň v ukázkovém programu k tomuto článku je typickým příkladem) není jeho použití zrovna nejvhodnější, protože by pixely neustále kolovaly ke grafické kartě a zpět. V podobných případech je vhodnější uložit surface do systémové paměti.

Druhá položka struktury představuje formát pixelů (více níže), w a h specifikují rozměry obrázku v pixelech a pitch je délka jednoho řádku v bytech, ten může být zarovnán na určitou velikost. Pointer pixels ukazuje na grafická data obrázku (levý horní roh), jedná se buď o pixely, nebo v případě barevné hloubky osm bitů a menší o indexy do palety.

Clip_rect je ořezávací obdélník, díky kterému je možné obrázek pro některé funkce "imaginárně zmenšit", setkali jsme se s ním už minule u funkce SDL_SetClipRect(). Konečně refcount je počet referencí, který se používá při uvolňování obrázku z paměti. Kromě těchto parametrů obsahuje SDL_Surface ještě další privátní složky.

Pozn.: Žádný z těchto parametrů, vyjma ruční modifikace pixelů, by neměl být zadáván explicitně. Pro tyto činnosti slouží standardní funkce, které byly probrány v minulých článcích.

Struktura SDL_PixelFormat

Tato struktura popisuje formát pixelů uložených v surface, její podrobná znalost je nutná jen při požadavku přímého přístupu k pixelům. V ostatních případech by mělo stačit pouze vědět, že existuje a že ji lze najít v surface->format.

```
typedef struct
{
        SDL_Palette *palette;
        Uint8  BitsPerPixel;
```

```
        Uint8  BytesPerPixel;
        Uint32 Rmask, Gmask, Bmask, Amask;
        Uint8  Rloss, Gloss, Bloss, Aloss;
        Uint8  Rshift, Gshift, Bshift, Ashift;
        Uint32 colorkey;
        Uint8  alpha;
} SDL_PixelFormat;
```

Palette buď ukazuje na paletu, nebo je, u barevné hloubky větší než 8 bitů, nastaveno na NULL. Barevná hloubka je specifikována hned ve dvou položkách. V první z nich je uložena v bitech a u druhé jsou jednotkami byty.

Bity RGBA masky jsou na pozici dané složky v jedničce, RGBA loss určuje ztrátu přesnosti barevné složky - $2^{[RGBA]loss}$. RGBA shift označuje počet bitů zprava v hodnotě pixelu k dané komponentě. Colorkey určuje transparentní barvu a alpha "globální hodnotu alfa kanálu" surface.

Struktury SDL_Palette a SDL_Color

Struktura SDL_Palette obsahuje ukazatele na barvy palety a SDL_Color je tvořena jednotlivými RGB složkami barvy.

```
typedef struct
{
        int ncolors;
        SDL_Color *colors;
} SDL_Palette;

typedef struct
{
        Uint8 r;
        Uint8 g;
        Uint8 b;
        Uint8 unused;
} SDL_Color;
```

Adresace pixelů a získání barevných komponent

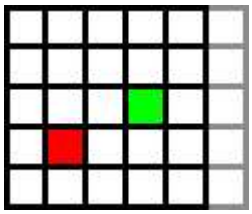
Pixely jsou v surface uloženy do jednorozměrného pole, a tudíž může vyvstat otázka, jak je adresovat při použití dvourozměrných x, y koordinátů. Požadovaná adresa pixelu se získá vynásobením šířky řádku y-ovou pozicí a přičtením x-ové pozice k výsledku. Je nutné vzít v úvahu ještě barevnou hloubku, ale jinak se nejedná o nic složitého.

Příklad bude možná názornější. Na obrázku níže je vidět mřížka, ve které každý čtvereček symbolizuje jeden pixel. Šedý okraj vpravo představuje nevyužitou část paměti (parametr pitch). Adresa zvýrazněných pixelů (indexů do palety) se bude rovnat (měřeno v bytech):

```
Uint8 *adr;
int bypp = s->format->BytesPerPixel;

// ADRESA = POČÁTEK + ŘÁDEK*ŠÍŘKA ŘÁDKU
//              + SLOUPEC*ŠÍŘKA PIXELU;

// Zelený
adr = (Uint8 *)s->pixels + 2*s->pitch + 3*bypp;
// Červený
adr = (Uint8 *)s->pixels + 3*s->pitch + 1*bypp;
```



Je-li pixel načtený, je většinou potřeba získat hodnoty jednotlivých RGB(A) složek. Žádné pevně dané pořadí (RGB, BGR apod.) není v SDL obecným pravidlem. Jak tedy na to? Pixel se binárně ANDuje s maskou barvy, čímž se vynulují hodnoty všech ostatních komponent, poté se aplikují dva binární posuny, nejprve o shift doprava a následně o loss doleva.

Po průchodu následujícím kódem bude proměnná red obsahovat červenou složku barvy v pixelu. Získání modré, zelené nebo alfy je analogické.

```
Uint8 red;
Uint32 tmp, pixel;
// fmt je ukazatel na formát pixelů

tmp = pixel & fmt->Rmask;       // Maskování
tmp = tmp >> fmt->Rshift;       // Posun na pravý okraj
tmp = tmp << fmt->Rloss;        // Expanze na 8 bitů
red = (Uint8)tmp;               // "Ořeže" nuly vlevo
```

Druhou možností by bylo použít standardní funkci SDL_GetRGB(), která byla popsána v minulém dílu.

Mimochodem, vždy je možné si zavést konvenci, že všechny surface v programu budou například ve formátu RGB(A) a tím tyto komplikace obejít. Na druhou stranu, program bude méně univerzální a při skládání dvou kódů vyvíjených nezávisle na sobě mohou vzniknout zbytečné komplikace.

Zamknutí surface

V případě, že chce programátor přistupovat přes ukazatel surface->pixels přímo k jednotlivým pixelům, měl by nejdříve surface uzamknout. Jedinou výjimkou jsou takové surface, u kterých makro SDL_MUSTLOCK() vrátí nulu, pak je přístup k pixelům možný kdykoli.

Za "práci s pixely" se považuje ruční přístup k datům přes ukazatel ve struktuře. Naopak u kreslících funkcí, které jsou poskytovány SDL (SDL_BlitSurface() apod.), by surface nikdy být zamknut neměl!

```
int SDL_LockSurface(SDL_Surface *surface);
void SDL_UnlockSurface(SDL_Surface *surface);
```

Po ukončení úprav pixelů by mělo vždy následovat odemknutí a jelikož jsou zámky vícenásobné, mělo by ke každému zamknutí existovat odpovídající odemknutí. To znamená, že pokud je surface zamknut dvakrát, měl by být také dvakrát odemknut.

Mezi těmito funkcemi by se také nemělo vyskytnout žádné systémové nebo knihovní volání. V obecném případě by zamykání a odemykání mohlo vypadat např. takto:

```
if(SDL_MUSTLOCK(screen))
{
        if(SDL_LockSurface(screen) < 0)
        {
                return;
```

```
        }
}

// Práce s pixely

if(SDL_MUSTLOCK(screen))
{
        SDL_UnlockSurface(screen);
}
```

Kurzor myši

Na závěr výkladu o SDL grafice bude probráno téma změny kurzoru myši. K jeho vytvoření slouží funkce SDL_CreateCursor(), která vrací ukazatel na nově vytvořenou strukturu SDL_Cursor.

```
SDL_Cursor *SDL_CreateCursor(Uint8 *data, Uint8 *mask,
                int w, int h, int hot_x, int hot_y);
```

První dva parametry jsou bitovými mapami a určují, jak bude výsledný kurzor vypadat (viz dále). Další dva označují šířku a výšku, obě hodnoty musí být násobkem čísla osm a poslední dva parametry specifikují aktivní bod kurzoru.

Kurzor vytvořený pomocí SDL může být pouze černobílý, takže by mělo stačit pouze jedno bitové pole. Nesmí se však zapomenout ještě na průhlednou a případně invertovanou barvu, což dává celkem čtyři možné kombinace, jejichž význam vysvětluje následující tabulka.

Data	Maska	Výsledný pixel kurzoru
0	1	Bílý
1	1	Černý
0	0	Průhledný
1	0	Je-li dostupný, tak invertovaný, jinak černý

Po skončení práce s kurzorem by měla být vždy zavolána funkce SDL_FreeCursor(), která se postará o jeho uvolnění z paměti.

```
void SDL_FreeCursor(SDL_Cursor *cursor);
```

Kurzor lze nastavit za aktivní voláním funkce SDL_SetCursor(). Naopak aktuálně aktivní kurzor lze získat funkcí SDL_GetCursor().

```
void SDL_SetCursor(SDL_Cursor *cursor);
SDL_Cursor *SDL_GetCursor(void);
```

Poslední operací, která se dá provést s kurzorem myši, je jeho zobrazení popř. skrytí. Po startu aplikace je implicitně zobrazen.

```
int SDL_ShowCursor(int toggle);
```

Symbolická konstanta SDL_DISABLE kurzor skryje, naopak SDL_ENABLE ho zobrazí. Pomocí SDL_QUERY bude vrácen aktuální stav.

V následujícím příkladu se vytvoří kurzor ve tvaru bílého čtverce o velikosti 8x8 pixelů, za aktivní bod je definován levý horní roh.

```
// Globální proměnná
SDL_Cursor *g_cursor;

// Inicializace
Uint8 data[] = { 0x00, 0x00, 0x00, 0x00,
                 0x00, 0x00, 0x00, 0x00 };
Uint8 mask[] = { 0xFF, 0xFF, 0xFF, 0xFF,
                 0xFF, 0xFF, 0xFF, 0xFF };

g_cursor = SDL_CreateCursor(data, mask, 8, 8, 0, 0);
SDL_SetCursor(g_cursor);

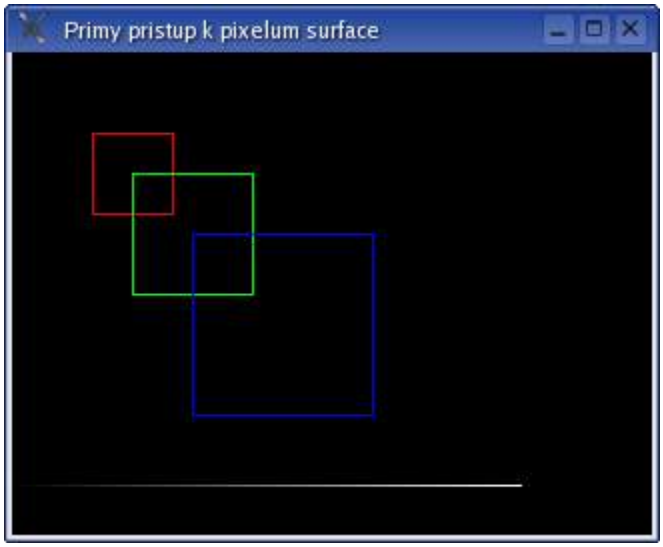
// Deinicializace (většinou konec aplikace)
SDL_FreeCursor(g_cursor);
```

Pozn.: SDL sice umožňuje vytvářet pouze černobílé kurzory, ale to neznamená, že nelze používat i barevné. Vždy je možné pomocí SDL_ShowCursor(SDL_DISABLE) standardní kurzor skrýt a místo něho při každém vykreslení zobrazit libovolný obrázek nebo dokonce spritovou animaci (animovaný kurzor).

Ukázkové programy

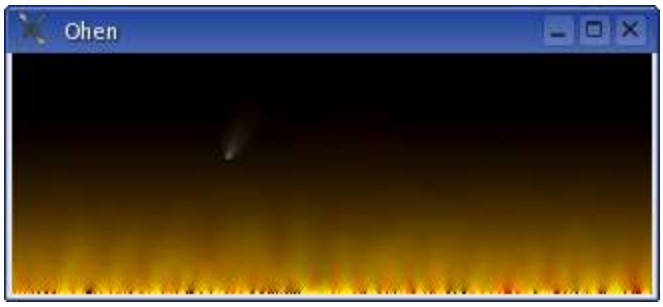
Přímý přístup k pixelům surface

Při ruční modifikaci pixelů bývá největším problémem adresovat místo v paměti, na které se má zapisovat. O tuto činnost se stará funkce DrawPixel(), která byla převzata ze SDL intro a trochu upravena. Demonstrační program touto technikou vykreslí tři čtverce a linku palety šedi.



Oheň

Druhý příklad simuluje hořící oheň. Na nejnižším řádku se generují náhodné pixely z palety barev ohně, které se s rostoucí výškou postupně rozmazávají. V programu je dále definován kurzor myši ve tvaru "zaměřovače" (černé kolečko s bílým středem; na screenshotu není vidět), kterým je možné do ohně přidávat bílé pixely.


Ohen

OpenGL

Díky přímé podpoře OpenGL umožňuje SDL renderovat i 3D grafické objekty, které se staly nepsaným standardem naprosté většiny dnešních her. Tentokrát se tedy budeme věnovat podpoře OpenGL v SDL.

Okno s podporou OpenGL

Ve čtvrtém dílu bylo ukázáno, že jediným rozdílem mezi vytvořením "klasického" okna a okna s podporou OpenGL je symbolická konstanta SDL_OPENGL (respektive SDL_OPENGLBLIT), která se při inicializaci předá spolu s ostatními flagy funkci SDL_SetVideoMode(). Tím bychom mohli celý článek skoro ukončit, ale zbývá probrat ještě několik věcí...

Soubory pro OpenGL

SDL nabízí programátorovi hlavičkový soubor SDL_opengl.h, který za něj vyřeší různé umístění OpenGL souborů gl.h a glu.h v některých systémech. Zároveň umožňuje používat rozšíření (extensiony), ale nevkládá je klasicky prostřednictvím glext.h, ale jeho obsah zahrnuje přímo v sobě.

Nemělo by být zapomenuto na přilinkování OpenGL knihoven (libGL.so, libGLU.so v Linuxu popř. opengl32.lib a glu32.lib ve Visual C++ pod MS Windows), jinak program nepůjde s odkazy na neexistující funkce vytvořit.

Atributy OpenGL kontextu

Před samotným voláním SDL_SetVideoMode() by již měly být specifikovány atributy definující vlastnosti OpenGL kontextu, po vytvoření okna už nepůjdou změnit.

```
int SDL_GL_SetAttribute(SDL_GLattr attr, int value);
```

Prvním parametrem se určuje nastavovaný atribut a druhý parametr představuje jeho hodnotu. Za atributy lze použít některou z následujících konstant.

- SDL_GL_RED_SIZE, SDL_GL_GREEN_SIZE, SDL_GL_BLUE_SIZE, SDL_GL_ALPHA_SIZE
  Velikosti jednotlivých barevných komponent ve framebufferu
- SDL_GL_BUFFER_SIZE
  Velikost framebufferu v bitech
- SDL_GL_DOUBLEBUFFER
  Nula vypíná OpenGL double buffering, jednička zapíná. Tento parametr nemá nic společného se SDL_DOUBLEBUF předávaného do SDL_SetVideoMode().
- SDL_GL_DEPTH_SIZE
  Velikost bufferu hloubky
- SDL_GL_STENCIL_SIZE
  Velikost stencil bufferu
- SDL_GL_ACCUM_RED_SIZE, SDL_GL_ACCUM_GREEN_SIZE, SDL_GL_ACCUM_BLUE_SIZE, SDL_GL_ACCUM_ALPHA_SIZE
  Velikosti jednotlivých komponent v akumulačním bufferu
- SDL_GL_STEREO
  Stereoskopický OpenGL kontext; parametr není dostupný na všech systémech
- SDL_GL_MULTISAMPLEBUFFERS, SDL_GL_MULTISAMPLESAMPLES
  Zapíná fullscreenový antialiasing (fsaa) a specifikuje počet vzorků; do SDL přidán ve verzi 1.2.6 a je dostupný pouze, pokud grafická karta podporuje rozšíření GL_ARB_multisample. Tento parametr zlepšuje grafické vzezření aplikace - vyhlazuje ostré hrany barevných přechodů.

Pozn.: Poslednímu parametru, fsaa, se nebudu dále věnovat, protože moje grafická karta zmíněný extension nepodporuje.

Projevuje se to tak, že se SDL_SetVideoMode() při jeho definování ukončí s chybou a následný SDL_GetError() vrátí řetězec "Couldn't find matching GLX visual".

Pravděpodobně bude nutné vytvořit "obyčejné" OpenGL okno a zeptat se gluCheckExtension(), zda je fsaa podporován. Pokud ano, zavřít okno a vytvořit ho znovu, tentokrát s podporou fsaa, pokud ne, pokračovat beze změny dále. Druhou možností je načítat konfiguraci ze souboru a nechat jeho zapnutí na uživateli.

Typický příklad nastavení OpenGL atributů

```
// Umístit PŘED volání SDL_SetVideoMode()
SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);    // Doublebuffering ano
SDL_GL_SetAttribute(SDL_GL_BUFFER_SIZE, 24);    // 24 bitový framebuffer
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);     // 24 bitový depth buffer

SDL_GL_SetAttribute(SDL_GL_STENCIL_SIZE, 0);    // Žádný stencil buffer
SDL_GL_SetAttribute(SDL_GL_ACCUM_RED_SIZE, 0);  // Žádný akumulační buffer
SDL_GL_SetAttribute(SDL_GL_ACCUM_GREEN_SIZE, 0);
SDL_GL_SetAttribute(SDL_GL_ACCUM_BLUE_SIZE, 0);
SDL_GL_SetAttribute(SDL_GL_ACCUM_ALPHA_SIZE, 0);

// Pouze pokud grafická karta podporuje GL_ARB_multisample
// SDL_GL_SetAttribute(SDL_GL_MULTISAMPLEBUFFERS, 1);// FSAA ano
// SDL_GL_SetAttribute(SDL_GL_MULTISAMPLESAMPLES, 2);// 2 vzorky
```

Zjištění atributů

Někdy může být dobré po vytvoření okna zjistit, zda byl, nebo nebyl atribut nastaven. Slouží k tomu funkce SDL_GL_GetAttribute().

```
int SDL_GL_GetAttribute(SDLGLattr attr, int *value);
```

Stejně jako SDL_GL_SetAttribute() i tato funkce vrací při úspěchu 0 a při neúspěchu -1, ale měla by být volána až po SDL_SetVideoMode(). Hodnota zjišťovaného atributu bude uložena na adresu value.

Příklad na zjištění velikosti hloubkového bufferu:

```
// Umístit ZA volání SDL_SetVideoMode()
int tmp;
SDL_GL_GetAttribute(SDL_GL_DEPTH_SIZE, &tmp);

printf("Velikost hloubkového bufferu je %d bitů\n.", tmp);
```

Prohození vykreslovacích bufferů

K prohození předního a zadního bufferu po renderingu scény slouží v SDL funkce SDL_GL_SwapBuffers(), bez jejího volání by se nikdy nic nezobrazilo.

```
void SDL_GL_SwapBuffers(void);
```

Pokud byla při vytváření okna definována možnost použití i klasické SDL grafiky (SDL_OPENGLBLIT), je nutné volat navíc i SDL_UpdateRects().

Získání adresy OpenGL funkce

Ukazatel na jakoukoli OpenGL rutinu (většinou se jedná o rozšíření) lze získat pomocí funkce

```
void *SDL_GL_GetProcAddress(const char* proc);
```

Parametrem je řetězec se jménem funkce a návratovou hodnotou daný ukazatel. Pokud nebude funkce nalezena je vráceno NULL.

Specifikace OpenGL knihovny

SDL se v běžném případě linkuje s OpenGL knihovnou, která se nachází v systému, ale pokud programátor chce, může být SDL zkompilováno, aby nahrávalo OpenGL ovladač v runtimu (standardně vypnuto).

```
int SDL_GL_LoadLibrary(const char *path);
```

Tato funkce musí být opět volána ještě před SDL_SetVideoMode(), parametr specifikuje diskovou cesta k OpenGL knihovně. Pokud se ji podaří nahrát, je vrácena nula, jinak -1. Následně musí být pomocí SDL_GL_GetProcAddress() získány ukazatele na všechny OpenGL funkce, včetně glEnable(), glBegin() atd., takže použití této techniky může leckomu připadat velmi těžkopádné.

OpenGL textury a SDL_Surface

Jednou z velkých výhod spojení OpenGL s knihovnou SDL je možnost nahrávat obrázky pro textury za použití knihovny SDL_Image. Bohužel však existují dvě překážky, které znemožňují přímočaré použití.

První z nich je množství nejrůznějších vnitřních formátů SDL_Surface, které samotnému SDL sice nevadí, ale při použití kdekoli jinde se na ně musí pamatovat a vždy hlídat správný formát. Když se pomine paletový režim, pak stále zůstává prakticky libovolné umístění barevných složek (RGB, BGR apod.). Pravděpodobně nejspolehlivějším překonáním tohoto problému je vytvořit nový surface s pro OpenGL použitelným formátem a přes SDL_Blit() do něj zkopírovat původní surface.

Druhý problém spočívá v tom, že textura vytvořená ze SDL_Surface je v OpenGL vzhůru nohama, knihovny totiž používají vzájemně nekompatibilní souřadnicový systém - v SDL je bod 0, 0 nahoře, u OpenGL textur standardně dole.

Řešení je hned několik. Všude v programu lze zadávat v koordinát jako 1-v. Tím se sice problém spolehlivě vyřeší, ale musí se dávat pozor, aby toto pravidlo nebylo porušeno. Textury z více různých zdrojů se stanou vražednou kombinací...

Další možnost spočívá ve změně souřadnicového systému textur, stačí vložit následující kód do inicializace. Nicméně u textur z více zdrojů mohou opět vzniknout problémy a psát tyto čtyři řádky zvlášť při každém použití, nemusí být zrovna pohodlné.

```
glMatrixMode(GL_TEXTURE);
        glLoadIdentity();
        glScalef(1.0f, -1.0f, 1.0f);
glMatrixMode(GL_MODELVIEW);
```

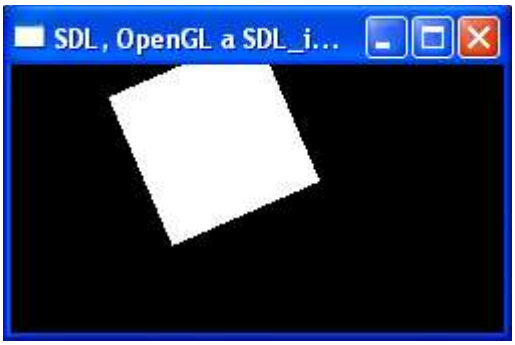
Posledním a asi nejvhodnějším způsobem je před vlastním vytvořením textury přímo v surface natvrdo prohodit řádky. Tento postup je ukázán ve druhém ukázkovém programu z této lekce.

Poznámka ohledně změny velikosti okna

Při vytváření OpenGL aplikací pod knihovnou SDL jsem objevil jistou nekompatibilitu mezi systémy Linux a Windows. Když uživatel změní velikost okna, aplikace by měla zareagovat a přizpůsobit se. Ve Windows stačí aktualizovat OpenGL viewport a perspektivu, nicméně v Linuxu musí být zavolána i funkce SDL_SetVideoMode(). Bez ní bude program vypadat jako na následujícím obrázku - okno se sice roztáhne, ale oblast, do které se kreslí, zůstane nezměněna.



Problémem je, že volání SDL_SetVideoMode() způsobí ve Windows ztrátu OpenGL kontextu, čili resetují se všechna nastavení (barva pozadí, blending, mlha...), zmizí textury, display listy atd.



Tento problém řeším podmíněným překladem. Když kompiluji program pro Linux, definuji symbolickou konstantu, která způsobí přidání SDL_SetVideoMode() do kódu, když ve Windows, řádek s #define zakomentuji. Možná to není zrovna nejlepší cesta, ale bez problémů funguje. Pokud někdo znáte lepší řešení, svěřte se prosím do diskuze...

```
#define CALL_SETVIDEOMODE_WHEN_RESIZING

        // Ošetření události změny velikosti okna
        case SDL_VIDEORESIZE:
#ifdef CALL_SETVIDEOMODE_WHEN_RESIZING
                g_screen = SDL_SetVideoMode(event.resize.w,
                        event.resize.h, WIN_BPP, WIN_FLAGS);

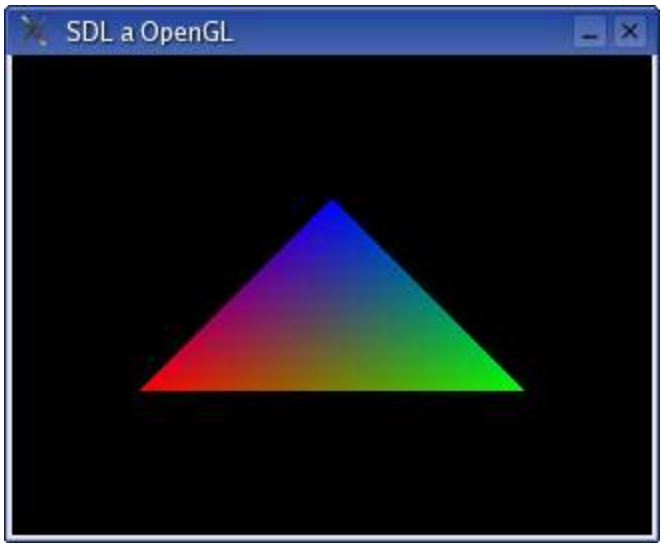
                if(g_screen == NULL)
                {
                        fprintf(stderr,
                                "Unable to resize window: %s\n",
                                SDL_GetError());
                        return false;
                }
#endif
                ResizeGL(event.resize.w, event.resize.h);
                break;
```

Možná by to šlo celé automatizovat pomocí symbolických konstant, které se během překladu definují nezávisle na programátorovi a které většinou obsahují jméno kompilátoru, verzi, operační systém atd., ale proč si komplikovat život.

Ukázkové programy

RGB Trojúhelník

Příklad ukazuje nastavení OpenGL atributů a vytvoření okna s podporou OpenGL. Aby nezůstalo jen u černého pozadí, je vykreslován trojúhelník s lineárním mísením barev.



Rotující logo SDL

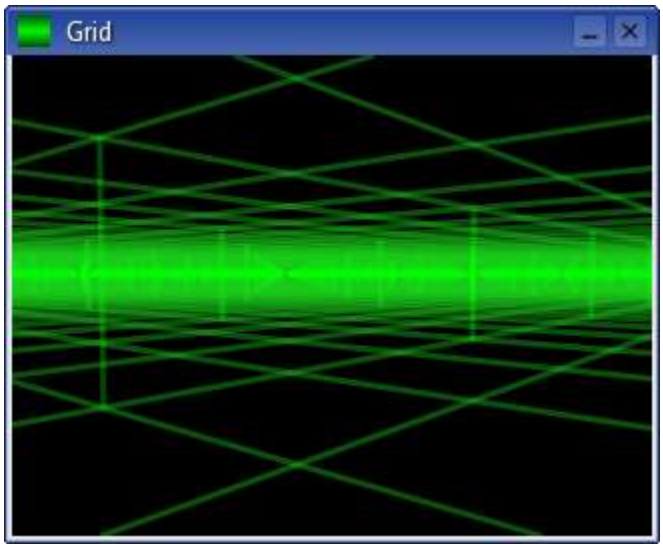
Druhý příklad vykresluje jednoduchou animaci rotujícího loga knihovny SDL. Obrázek pro texturu je uložen na disku ve formátu PNG a do programu je nahráván pomocí knihovny SDL_image.



Pohyb v mřížce

Jedná se o jednoduché demo ovládané myší, ve kterém se hráč pohybuje mřížkou. Díky periodickému opakování

elementárních buněk v prostoru nelze nikdy dojít na okraj. Kód je založen na jedné malé knihovně, kterou se v poslední době snažím dát dohromady, ale zatím ještě nebyla nezveřejněna.



Výstup textu pomocí SDL_ttf

V dnešním dílu bude popsána knihovna SDL_ttf, která slouží pro výpisy textů do scény. Se zobrazením textů a především s českými znaky bývá někdy potíž, nicméně použití SDL_ttf je velice jednoduché a naprosto bezproblémové.

Stručně o SDL_ttf

SDL_ttf není samostatná knihovna, ale jedná se spíše o jakýsi obal/rozhraní knihovny FreeType, který vznikl kvůli maximálnímu zjednodušení výpisu textů v SDL aplikacích. Jeho použití spočívá v inicializaci, nahrání fontu z disku (formáty .FON, .TTF) a samotný výpis textu, který probíhá tak, že je řetězec nejdříve vykreslen do SDL_Surface, který se poté přilepí na obrazovku.

Licence

Knihovna SDL_ttf je stejně jako samotné SDL šířena pod GNU LGPL. Předtím, než ji začnete používat, měli byste se seznámit se softwarovými patenty týkajícími se TrueType fontů a sami se rozhodnout, zda použít SDL_ttf, nebo zvolit jinou alternativu.

Jak to chápu já (nejsem právník!!!), tak čtení, konverze nebo generování TrueType fontů pod tyto patenty nespadá, navíc FreeType 2.0 žádné (známé) patentované techniky nepoužívá. Podrobnosti lze najít u knihovny FreeType.

Druhé upozornění se týká vlastních fontů, na mnoho z nich mají jejich tvůrci copyright. Nicméně toto by neměl být až tak velký problém, po internetu se potulují spousty fontů, které jsou volně šiřitelné - google "free fonts".

Instalace, zprovoznění

Nejdříve je nutné stáhnout a nainstalovat knihovnu FreeType (2.0 nebo novější) a až poté se může přistoupit k samotnému SDL_ttf. Obě bývají v standardních balíčcích Linuxových distribucí, možná však bude nutné nainstalovat ještě jejich "devel" verze. SDL_ttf by mělo fungovat na všech systémech, ve kterých funguje SDL. Mimochodem dokumentaci lze najít zde.

Co se týče vlastního programování, je nutné k parametrům gcc přidat volbu -lSDL_ttf, která způsobí přilinkování knihovny. Do zdrojových kódů se dále musí inkludovat soubor SDL_ttf.h, ale to už je samozřejmost.

Inicializace, deinicializace

Jak brzy zjistíme, obecné TTF funkce si vzaly za vzor SDL, rozdíly spočívají v podstatě pouze ve jméně, které začíná na předponu TTF_, návratové hodnoty jsou také stejné.

```
int  TTF_Init(void);
void TTF_Quit(void);
int  TTF_WasInit(void);

char *TTF_GetError(void);
void TTF_SetError(const char *fmt, ...);
```

Nahrávání fontů

Hlavní funkcí pro loading fontu do aplikace je TTF_OpenFont(), která vrací ukazatel na TTF_Font. Tato struktura se bude předávat ostatním TTF funkcím, nikdy by se k ní z důvodu budoucí kompatibility nemělo přistupovat přímo. V parametrech funkce se specifikuje disková cesta k souboru a ptsize definuje velikost fontu v měřítku 72 DPI.

```
TTF_Font *TTF_OpenFont(const char *file, int ptsize);
TTF_Font *TTF_OpenFontIndex(const char *file,
                int ptsize, long index)
```

Druhá uvedená funkce má v podstatě stejný význam, jako první, ale v posledním parametru lze navíc určit, který font ze souboru, pokud jich obsahuje více, se má použít. První bývá vždy na indexu 0 a pokud bude předáno číslo vyšší, než jich ve skutečnosti obsahuje, použije se poslední z nich.

Pro nahrání fontu existuje ještě jedna funkce, která však nepracuje se soubory na disku, ale s daty v paměti.

```
TTF_Font *TTF_OpenFontIndexRW(SDL_RWops *src,
                int freesrc, int ptsize, long index);
```

Po skončení práce s fontem by nemělo být nikdy zapomenuto na jeho uvolnění...

```
void TTF_CloseFont(TTF_Font *font)
```

Renderování textu

V úvodním odstavci už bylo zmíněno, že zobrazení textu probíhá dvoustupňově, nejdříve se vytvoří surface s vykresleným řetězcem, se kterým se může dále pracovat - například zobrazit ho na obrazovku pomocí SDL_Blit(). Pro vytvoření tohoto surface slouží celkem devět funkcí, které se dají rozdělit do tří skupin a to buď podle způsobu vykreslování, nebo podle formátu předaného řetězce.

Pokud je řetězec v kódování LATIN1 (ISO 8859-1, 7-bitový anglický text), renderuje se funkcemi se symbolickým jménem TTF_RenderText_*(), je-li v Unicode utf8, používá se TTF_RenderUTF8_*() a funkce TTF_RenderUNICODE_*() slouží pro vykreslení řetězce ve formátu Unicode utf16.

Druhý typ dělení spočívá v technice a kvalitě vykreslení. Základem jsou funkce TTF_Render*_Solid(), které nepoužívají žádný typ vyhlazování. Funkce se jmény TTF_Render*_Shaded() vyhlazování sice používají, ale neumí vytvořit průhledné pozadí. U třetího typu, TTF_Render*_Blended(), je text vyhlazený a pozadí průhledné.

Na uvedeném obrázku jsou názorně vidět definované rozdíly. Řetězce byly vykresleny bílou barvou na modré pozadí okna, u druhého z nich je navíc definováno černé pozadí.



Jak už bylo řečeno, všech devět funkcí si je velmi podobných. Všechny vracejí ukazatel na vytvořený surface s textem. Za první parametr se předává ukazatel na strukturu fontu, v druhém se specifikuje řetězec a třetí slouží k definování barvy textu. U druhého typu, Shaded, se navíc předává ještě barva pozadí.

```
SDL_Surface *TTF_RenderText_Solid(TTF_Font *font,
                const char *text, SDL_Color fg);
SDL_Surface *TTF_RenderUTF8_Solid(TTF_Font *font,
                const char *text, SDL_Color fg);
SDL_Surface *TTF_RenderUNICODE_Solid(TTF_Font *font,
                const Uint16 *text, SDL_Color fg);

SDL_Surface *TTF_RenderText_Shaded(TTF_Font *font,
                const char *text, SDL_Color fg,
```

```
                SDL_Color bg);
SDL_Surface *TTF_RenderUTF8_Shaded(TTF_Font *font,
                const char *text, SDL_Color fg,
                SDL_Color bg);
SDL_Surface *TTF_RenderUNICODE_Shaded(TTF_Font *font,
                const Uint16 *text, SDL_Color fg,
                SDL_Color bg);

SDL_Surface *TTF_RenderText_Blended(TTF_Font *font,
                const char *text, SDL_Color fg);
SDL_Surface *TTF_RenderUTF8_Blended(TTF_Font *font,
                const char *text, SDL_Color fg);
SDL_Surface *TTF_RenderUNICODE_Blended(TTF_Font *font,
                const Uint16 *text, SDL_Color fg);
```

Funkce typu Solid generují osmi bitový paletový surface, u kterého první (resp. nultý) pixel specifikuje barvu pozadí a druhý barvu textu. U funkcí Shaded je zvláštním pixelem pouze ten první, protože kvůli barevným přechodům na okrajích znaků nemůže být barva textu určena jednoznačně. Blended funkce vytvářejí třiceti dvou bitový surface ve formátu ARGB. Text se tedy renderuje ve vysoké kvalitě s alfa blendingem, na druhou stranu je tato metoda o něco pomalejší.

SDL_ttf dále definuje funkce pro vykreslení jednoho (Unicode) znaku.

```
SDL_Surface *TTF_RenderGlyph_Solid(TTF_Font *font,
                Uint16 ch, SDL_Color fg);
SDL_Surface *TTF_RenderGlyph_Shaded(TTF_Font *font,
                Uint16 ch, SDL_Color fg, SDL_Color bg);
SDL_Surface *TTF_RenderGlyph_Blended(TTF_Font *font,
                Uint16 ch, SDL_Color fg);
```

Příklad na výpis textu

Kompletní příklad vykreslení textu včetně inicializace, nahrání fontu a deinicializace by mohl vypadat následovně.

```
// Globální proměnná
TTF_Font *g_font;


// Inicializace (za SDL_Init())
if(TTF_Init() == -1)
{
        printf("Unable to initialize SDL_ttf: %s\n",
                        TTF_GetError());
        return false;
}

g_font = TTF_OpenFont("font.ttf", 12);
if(!g_font)
{
        printf("Unable to open font: %s\n",
                        TTF_GetError());
        return false;
}


// Vykreslování
```

```
SDL_Color col = { 255, 255, 255, 0 };
SDL_Rect rect = { 20, 20, 0, 0 };
SDL_Surface *text;

text = TTF_RenderText_Solid(g_font, "Text", fg_col);
if(text != NULL)
{
        SDL_BlitSurface(text, NULL, g_screen, &rect);
        SDL_FreeSurface(text);
}


// Deinicializace
if(g_font != NULL)
{
        TTF_CloseFont(g_font);
        g_font = NULL;
}
TTF_Quit();
```

Další užitečné funkce

Pomocí následujících dvou funkcí lze specifikovat/dotázat se, zda má být font vykreslován normálně, tučně, kurzívou nebo podtržený. Za parametr style lze předat binárně ORovanou kombinaci symbolických konstant TTF_STYLE_NORMAL, TTF_STYLE_BOLD, TTF_STYLE_ITALIC a TTF_STYLE_UNDERLINE.

```
void TTF_SetFontStyle(TTF_Font *font, int style);
int TTF_GetFontStyle(TTF_Font *font);
```

Pixelové rozměry řetězce po vykreslení, které lze použít například při zarovnávání (doleva/na střed/doprava) lze získat funkcemi

```
int TTF_SizeText(TTF_Font *font,
                const char *text,
                int *w, int *h);
int TTF_SizeUTF8(TTF_Font *font,
                const char *text,
                int *w, int *h);
int TTF_SizeUNICODE(TTF_Font *font,
                const Uint16 *text,
                int *w, int *h);
```

Funkce TTF_FontHeight() vrátí maximální výšku předaného fontu v pixelech. Tato hodnota však není moc vhodná pro posun na další řádek, protože by byly moc blízko u sebe. K tomu slouží TTF_FontLineSkip(). Mimochodem SDL_ttf neposkytuje žádné funkce pro víceřádkové výpisy textu, programátor si je musí implementovat sám.

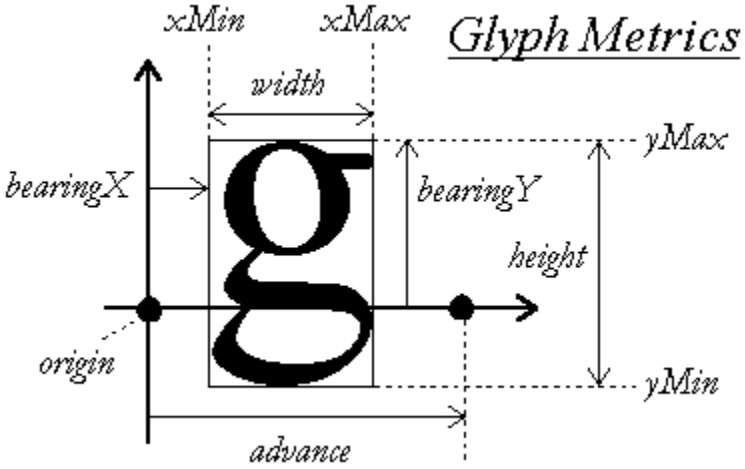
```
int TTF_FontHeight(TTF_Font *font);
int TTF_FontLineSkip(TTF_Font *font);
```

Hodnota vrácená funkcí TTF_FontAscent() označuje vzdálenost od horního okraje fontu k jeho základní lince, která se dá použít při vykreslování znaků relativně k hornímu okraji. TTF_FontDescent() se naopak vztahuje k okraji dolnímu.

```
int TTF_FontAscent(TTF_Font *font);
int TTF_FontDescent(TTF_Font *font);
```

Kompletní informace o rozměrech určitého znaku lze získat pomocí funkce TTF_GlyphMetrics(). Jednotlivé parametry vysvětluje obrázek níže (byl převzat ze SDL_ttf dokumentace a předtím z FreeType dokumentace). Velice rozsáhlý článek o GlyphMetrics lze najít v dokumentaci knihovny FreeType.

```
int TTF_GlyphMetrics(TTF_Font *font, Uint16 ch,
                     int *minx, int *maxx,
                     int *miny, int *maxy,
                     int *advance);
```



Při práci s širokými Unicode znaky může nastat situace, že budou byty ve znaku vzhledem k procesoru prohozené (little/big endian). Pomocí funkce TTF_ByteSwappedUNICODE() lze tento stav změnit. Při předání nenulové hodnoty (UNICODE_BOM_SWAPPED) se budou byty prohazovat, s nulou (UNICODE_BOM_NATIVE) nebudou.

```
void TTF_ByteSwappedUNICODE(int swapped);
```

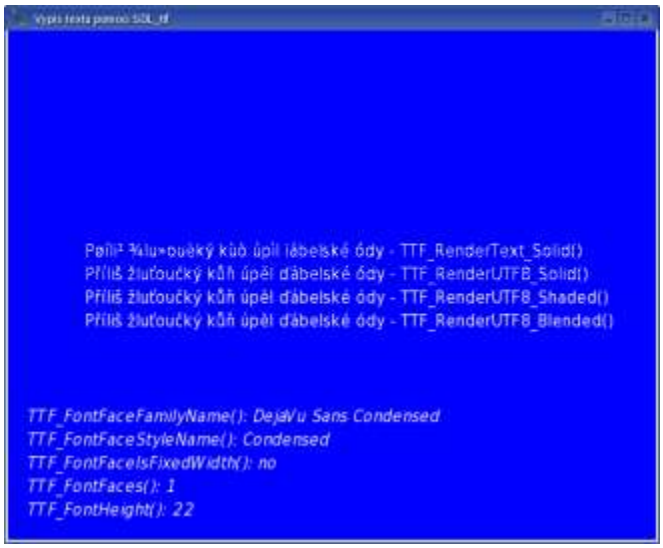
Poslední funkce, které budou probrány, poskytují spíše informativní hodnoty. Jsou jimi jméno rodiny fontu, jeho typ, jestli je font proporcionální nebo ne a počet faců.

```
char *TTF_FontFaceFamilyName(TTF_Font *font);
char *TTF_FontFaceStyleName(TTF_Font *font);
int  TTF_FontFaceIsFixedWidth(TTF_Font *font);
long TTF_FontFaces(TTF_Font *font);
```

Ukázkové programy

Výpis textu pomocí SDL_ttf

Ukázkový program je dnes relativně jednoduchý, na modrém pozadí je zobrazeno několik řádek textu. Každá řádka se kreslí jinou technikou a je ukázán i výpis českých znaků. V levém dolním rohu se zobrazují i informace o použitém fontu.


Příliš žluťoučký kůň úpěl ďábelské ódy - TTF_RenderText_Solid()
Příliš žluťoučký kůň úpěl ďábelské ódy - TTF_RenderUTF8_Solid()
Příliš žluťoučký kůň úpěl ďábelské ódy - TTF_RenderUTF8_Shaded()
Příliš žluťoučký kůň úpěl ďábelské ódy - TTF_RenderUTF8_Blended()
TTF_FontFaceFamilyName(): DejaVu Sans Condensed
TTF_FontFaceStyleName(): Condensed
TTF_FontFaceIsFixedWidth(): no
TTF_FontFaces(): 1
TTF_FontHeight(): 22

Komunikace se správcem oken, úvod do událostí

Seriál se přehoupl do druhé desítky, příště už na počítání přestanou stačit prsty ;-). Ale ještě než se tak stane, probereme si komunikaci aplikace se správcem oken, což v sobě zahrnuje změnu titulku okna, minimalizaci, přepínání do/z fullscreenu a několik dalších věcí. Ke konci bude také přidán lehký úvod do zpracování událostí.

Správce oken

Knihovna SDL poskytuje několik příkazů, které zajišťují komunikaci mezi aplikací a správcem oken (Window Manager). Samozřejmě není možné komunikovat, neexistuje-li druhá strana - většinou se jedná o běh v textovém režimu, když není spuštěný X server. SDL toho po pravdě nepodporuje mnoho, v podstatě pouze změnu názvu a ikony v titulkovém pruhu, programovou minimalizaci okna a přepnutí do/z fullscreenu. Názvy funkce, které zajišťují tyto činnosti, začínají na předponu SDL_WM_.

Titulkový pruh

Začneme jednoduše, řetězec v titulku okna se změní funkcí SDL_WM_SetCaption(), ostatně tato funkce byla použita snad ve všech ukázkových příkladech, takže by se nemělo jednat o nic nového.

```
void SDL_WM_SetCaption(const char *title, const char *icon);
void SDL_WM_GetCaption(char **title, char **icon);
```

První parametr je jasný, specifikuje se jím řetězec v titulku. Existenci druhého jsem však nikdy nepochopil. SDL dokumentace ho popisuje jako "jméno ikony" a ani hlavičkový soubor ani zdrojové kódy více informací bohužel neposkytují. Co si pod ním představit tedy opravdu netuším. Možná se jedná o "textovou ikonu", pro správce oken, které grafické neumožňují, ale toto je pouze má neověřená spekulace. Každopádně, pokud se předá NULL, nic se nezkazí.

Ikona aplikace se nastavuje funkcí SDL_WM_SetIcon(), která by měla být volána před SDL_SetVideoMode(). Co se týká rozměrů, jsou doporučovány klasické 32x32 pixelů velké ikony, neměly by s nimi být žádné problémy.

```
void SDL_WM_SetIcon(SDL_Surface *icon, Uint8 *mask);
```

První parametr představuje surface s ikonou a druhý je bitovou maskou pro průhledné části. Je-li předáno NULL, použije se klíčová barva surface a pokud ani ta není specifikována, bude ikona neprůhledná.

Bity masky nastavené do jedničky specifikují zobrazované a nuly naopak průhledné pixely, řádky jdou od shora dolů a každý z nich se skládá z (šířka / 8) bytů, zaokrouhleno nahoru. Nejvýznamnější bit každého bytu reprezentuje nejlevější pixel.

Typický příklad nastavení ikony okna, která nepoužívá průhlednost může vypadat například takto:

```
// Před SDL_SetVideoMode()
SDL_Surface *icon = SDL_LoadBMP("./icon.bmp");
if(icon != NULL)
{
        SDL_WM_SetIcon(icon, NULL);
        SDL_FreeSurface(icon);
}
```

Minimalizace okna

Okno se dá programem minimalizovat voláním funkce SDL_WM_IconifyWindow(). Vrácená nula značí, že minimalizace buď není podporována, nebo ji správce oken odmítl provést. V případě, že se vše uskutečnilo v pořádku, obdrží aplikace zprávu SDL_APPACTIVE s parametrem označujícím ztrátu fokusu.

```
int SDL_WM_IconifyWindow(void);
```

Přepnutí do/z fullscreenu

Pro přepnutí mezi oknem a fullscreenem stačí jediný řádek kódu. Tedy, abychom byli přesní, stačil by, pokud by nebyla funkce SDL_WM_ToggleFullScreen() podporována pouze v X11, v BeOSu je zatím pouze experimentálně.

```
int SDL_WM_ToggleFullScreen(SDL_Surface *surface);
```

Funkce vrací při úspěchu jedničku, jinak nulu, po jejím zavolání by se obsah okna neměl změnit. Pokud surface okna nevyžaduje zamykání při přístupu k pixelům, bude ukazatel obsahovat stejnou adresu paměti jako před voláním.

Jak už bylo zmíněno, pokud program neběží pod X11, ale například v MS Windows, přepnutí mezi oknem a celoobrazovkovým režimem nelze provést. Nicméně..., jak ukazuje demonstrační příklad níže, není problém okno zrušit a následně ho znovu vytvořit s negovaným parametrem režimu.

```
#define WIN_WIDTH 640
#define WIN_HEIGHT 480
#define WIN_BPP 0

// Globální proměnné
SDL_Surface *g_screen;
Uint32 g_win_flags = SDL_RESIZABLE|SDL_FULLSCREEN;

// Přepíná mezi režimy okno/fullscreen
bool ToggleFullscreen()
{
        if(g_win_flags & SDL_FULLSCREEN)// Z fullscreenu do okna
                g_win_flags &= ~SDL_FULLSCREEN;
        else// Z okna do fullscreenu
                g_win_flags |= SDL_FULLSCREEN;

        // Pokus o přepnutí, podporováno pouze v x11
        if(SDL_WM_ToggleFullScreen(g_screen) == 0)
        {
                fprintf(stderr, "Unable to toggle fullscreen,"
                                "trying to recreate window\n");

                SDL_FreeSurface(g_screen);
                g_screen = SDL_SetVideoMode(WIN_WIDTH, WIN_HEIGHT,
                                WIN_BPP, g_win_flags);

                if(g_screen == NULL)
                {
                        fprintf(stderr, "Unable to recreate window: %s\n",
                                        SDL_GetError());
                        return false;// Ukončí program
                }

#ifdef OPENGL_APLIKACE
                // Reinicializace OpenGL (parametry, textury...),
                // starý kontext už není dostupný
                if(!InitGL())
                {
                        fprintf(stderr, "Unable to reinitialize OpenGL\n");
```

```
                        return false;// Ukončí program
                }

                ResizeGLWindow();// Nastaví perspektivu
#endif

                Draw();// Překreslí scénu
        }

        return true;// OK
}
```

Tato a jí podobné funkce se většinou volají v reakci na stisk nějaké klávesy, a protože se v tomto článku začínáme zabývat událostmi, příklad volání lze nalézt níže...

Způsob grabování vstupů

Následující funkce umožňuje nastavit způsob grabování vstupů klávesnice a myši.

```
SDL_GrabMode SDL_WM_GrabInput(SDL_GrabMode mode);
```

V případě, že je nastaveno SDL_GRAB_OFF (implicitní nastavení) budou se zprávy předávat oknu pouze tehdy, pokud je aktivní (má fokus). Naopak, je-li ve stavu SDL_GRAB_ON, myš nemůže opustit klientskou oblast okna a všechny vstupy klávesnice jsou předávány přímo oknu, čili nejsou interpretovány okenním manažerem. Poslední možný parametr SDL_GRAB_QUERY slouží k dotazu na aktuální stav.

Správa událostí

Komunikace mezi operačním systémem a SDL aplikací je vystavěna na tzv. událostním modelu. Vždy, když v systému nastane nějaká událost, například uživatel stiskne klávesu nebo pohne myší, generuje operační systém objekt dané události, nastaví jeho parametry (stisknutá klávesa, nová pozice myši) a předá ho aplikaci. Někdy se také říká, že operační systém poslal aplikaci zprávu o události. Ta na ni může zareagovat naprosto libovolným způsobem, včetně její ignorace.

Povídání o událostech začneme praktickým příkladem jejich zpracování. V tuto chvíli nemusíte pochopit naprosto všechny detaily, pokud však vstřebáte základní principy, máte z 95 procent vyhráno, dále už se bude jednat jen o nabalování speciálních znalostí. No, a pokud následující příklad nepochopíte, tak to zkuste ještě jednou ;), od této chvíle se bez těchto věcí neobejdete.

Bývá dobrým zvykem vložit veškerou práci s událostmi do specializované funkce. Definujeme, že vrácené false z ProcessEvent() říká hlavní smyčce programu, že je z nějakého důvodu nutné ukončit aplikaci. V tomto případě chce uživatel buď ukončit program, stisknul klávesu Escape, nebo se nezdařilo přepnutí mezi oknem a fullscreenem.

Uvnitř funkce deklarujeme proměnnou typu SDL_Event, kterou budeme v cyklu naplňovat událostmi čekajícími ve frontě. Pokud je fronta prázdná, cyklus, a tedy i celá funkce se ukončí a řízení je předáno hlavní smyčce programu.

```
bool ProcessEvent()
{
        SDL_Event event;// Objekt události

        while(SDL_PollEvent(&event))
        {
```

Rozvětvíme kód podle typu události a pokud se jedná o klávesnici, zanoříme se, v závislosti na typu klávesy, ještě více do hloubky. Ošetřen je pouze Escape ukončující aplikaci a F1, která způsobí přepnutí do/z fullscreenu.

Mimochodem, názvy kláves lze najít v SDL dokumentaci téměř dole pod nadpisem "SDL Keysym definitions" nebo v hlavičkovém souboru SDL_keysym.h.

```
                switch(event.type)
                {
                // Klávesnice
                case SDL_KEYDOWN:
                        switch(event.key.keysym.sym)
                        {
                        case SDLK_ESCAPE:
                                return false;
                                break;

                        case SDLK_F1:
                                if(!ToggleFullscreen())
                                        return false;
                                break;

                        default:
                                break;
                        }
                        break;
```

V každé aplikaci by měl být ošetřen SDL_QUIT, tato událost nastane, když má být program ukončen. Uživatel například klikl na křížek v pravém horním rohu okna, stiskl ALT+F4, klikl pravým tlačítkem myši v hlavním panelu na aplikaci a zvolil Zavřít atd. Zareagujeme, jak se očekává, skončíme.

```
                // Požadavek na ukončení
                case SDL_QUIT:
                        return false;
                        break;
```

Pro zachování jednoduchosti tento ukázkový kód ostatní události ignoruje.

```
                // Ostatní se ignorují
                default:
                        break;
                }
        }

        return true;
}
```

Ukázkové programy

Úvod do událostí

Program tentokrát nic nevykresluje, na začátku je nastavena ikona a titulek okna a poté se hlídají události klávesnice. Pokud je stisknut ESC, aplikace se ukončí, v reakci na F1 se okno přepne do celoobrazovkého režimu nebo zpět, M okno minimalizuje a G změní způsob grabování vstupů (SDL_WM_GrabInput()). Pokud je stisknuta jiná klávesa, vypíše se její číslo a jméno.

Fronta událostí

Na konci minulého dílu jsme nakousli základní práci s událostmi, dnes budeme pokračovat. Tento článek je primárně věnován práci s frontou událostí, ale jelikož ještě nevíme nic o unionu SDL_Event, bude částečně probrán i on.

Základem zpracování událostí je v SDL union SDL_Event a funkce, které načítají tyto objekty z fronty zpráv. Nejdříve bude v rychlosti probrána zmíněná datová struktura a pak se budeme celý zbytek článku věnovat práci s frontou událostí.

Union SDL_Event

Pro ty kteří už zapomněli... Datový typ union je podobný klasické struktuře, rozdíl mezi nimi spočívá v tom, že v určitém okamžiku může v jeho vnitřku existovat vždy jen jedna z deklarovaných položek. Při vytváření se alokuje paměť o velikosti největší z nich.

Union SDL_Event je pravděpodobně, hned po SDL_Surface, druhý nejdůležitější a nejpoužívanější ze všech SDL datových typů. Jak už bylo několikrát zmíněno, poskytuje programátorovi rozhraní pro práci s událostmi.

```
typedef union
{
        Uint8 type;                     // Typ události

        SDL_ActiveEvent         active; // (De)aktivace okna
        SDL_KeyboardEvent       key;    // Klávesnice
        SDL_MouseMotionEvent    motion; // Myš
        SDL_MouseButtonEvent    button;
        SDL_JoyAxisEvent        jaxis;  // Joystick
        SDL_JoyBallEvent        jball;
        SDL_JoyHatEvent         jhat;
        SDL_JoyButtonEvent      jbutton;
        SDL_ResizeEvent         resize; // Změna velikosti okna
        SDL_ExposeEvent         expose; // Požadavek na překreslení
        SDL_QuitEvent           quit;   // Požadavek na ukončení
        SDL_UserEvent           user;   // Uživatelská událost
        SDL_SywWMEvent          syswm;  // Systémově závislá
} SDL_Event;
```

Jak to všechno funguje? Když uživatel například změní velikost okna, SDL vygeneruje objekt SDL_Event, atribut type nastaví na hodnotu SDL_VIDEORESIZE a do jeho parametrů (podobjekt resize) uloží nové rozměry okna. Celý objekt je pak vložen do fronty událostí.

Detekuje-li aplikace příchod zprávy, podle parametru type zjistí, že se jedná o změnu velikosti okna a v resize.w, resize.h najde nové rozměry. V závislosti na nich pak provede odpovídající akci - například překreslí scénu nebo aktualizuje OpenGL perspektivu.

Proměnná type může nabývat hodnot uvedených v následující tabulce v levém sloupci. Vpravo se pak nachází odpovídající struktura, ve které se hledají podrobnosti o události.

Typ události	**Odpovídající struktura**
SDL_ACTIVEEVENT	SDL_ActiveEvent
SDL_KEYDOWN/UP	SDL_KeyboardEvent
SDL_MOUSEMOTION	SDL_MouseMotionEvent
SDL_MOUSEBUTTONDOWN/UP	SDL_MouseButtonEvent

SDL_JOYAXISMOTION	SDL_JoyAxisEvent
SDL_JOYBALLMOTION	SDL_JoyBallEvent
SDL_JOYHATMOTION	SDL_JoyHatEvent
SDL_JOYBUTTONDOWN/UP	SDL_JoyButtonEvent
SDL_QUIT	SDL_QuitEvent
SDL_VIDEORESIZE	SDL_ResizeEvent
SDL_VIDEOEXPOSE	SDL_ExposeEvent
SDL_USEREVENT	SDL_UserEvent
SDL_SYSWMEVENT	SDL_SysWMEvent

Protože popis těchto struktur a všeho, co s nimi souvisí, zabere několik následujících článků, budeme se nejprve věnovat funkcím, které vyzvedávají události z fronty a pak až konkrétnímu popisu jednotlivých zpráv.

Načítání událostí z fronty

Existují dva základní způsoby, jak načíst událost z fronty, v SDL je reprezentují funkce SDL_PollEvent() a SDL_WaitEvent(). Obě vezmou událost, která je zrovna na řadě, zkopírují její data do předaného parametru a odstraní ji z fronty. Co se stane s událostí dál, záleží na programátorovi, který vytváří aplikaci.

```
int SDL_PollEvent(SDL_Event *event);
int SDL_WaitEvent(SDL_Event *event);
```

Rozdíl mezi těmito funkcemi se projeví až tehdy, když je fronta prázdná. Jak už z názvu SDL_WaitEvent() vyplývá, tato funkce čeká libovolně dlouho, dokud nějaká zpráva nedorazí. Narozdíl od toho, SDL_PollEvent() se v případě prázdné fronty ihned ukončí a nulovým návratovým kódem oznámí, že nebylo načteno nic. V ostatních případech vrátí jedničku, která vyjadřuje, že byla nějaká zpráva načtena a má se zpracovat. SDL_WaitEvent() naproti tomu vrátí nulu, pouze pokud nastane nějaká chyba.

Události se typicky zpracovávají v cyklu, který se ukončí, když je fronta prázdná.

```
SDL_Event event;
while(SDL_PollEvent(&event))
{
        // Zpracování události
}

// Všechny události zpracovány
```

Může vyvstat otázka, kterou z funkcí je lepší používat. Dá se říci, že v 99 procentech případů sáhne programátor po SDL_PollEvent() a SDL_WaitEvent() použije pouze ve výjimečných případech. Důvodem je, že program potřebuje neustále provádět určitou činnost, jako jsou animace a herní logika. Systémové časovače však nemusí být pro tento typ úloh zrovna nejlepší volbou, protože jsou většinou výrazně pomalejší než cyklus, který se provádí neustále dokola. U jednoduchých her je v podstatě jedno, co se použije, nicméně u trochu složitějších bývají s rychlostí velké problémy.

Události se načítají ze vstupních zařízení funkcí SDL_PumpEvents(), bez ní by nikdy aplikaci nepřišla žádná zpráva. V SDL_PollEvent() a SDL_WaitEvent() je volána automaticky, při jiném způsobu načítání zpráv musí být použita explicitně.

```
void SDL_PumpEvents(void);
```

SDL dokumentace uvádí, že SDL_PumpEvents() nesmí být použito v jiném vláknu než, ve kterém bylo voláno SDL_SetVideoMode().

Vkládání událostí do fronty

Událostní systém v SDL není pouze jednosměrný, ale může být použit i k dialogu mezi různými částmi aplikace. Funkce SDL_PushEvent() přebírá ukazatel na objekt události, který umístí do fronty (resp. její kopii), a v případě úspěchu vrátí 0, jinak -1.

```
int SDL_PushEvent(SDL_Event *event);
```

Většinou se posílají uživatelské události (SDL_USEREVENT), ale jelikož ještě nebyly vysvětleny, ukážeme si poslání na SDL_QUIT. Tato událost zprostředkovává programu požadavek na ukončení a nepřebírá žádné parametry.

```
void PushQuitEvent()
{
        SDL_Event event;
        event.type = SDL_QUIT;
        SDL_PushEvent(&event);
}
```

Kdykoli by byla v programu zavolána tato funkce, aplikace by byla ukončena (předpokládá se standardní ukončení aplikace po příchodu SDL_QUIT).

Obecná práce s frontou

Všechny činnosti s frontou zpráv, které byly právě probrány, a také některé další, mohou být provedeny pomocí SDL_PeepEvents(). Tato obecná funkce přebírá v prvních dvou parametrech ukazatel na pole událostí a samozřejmě jeho velikost.

```
int SDL_PeepEvents(SDL_Event *events, int numevents,
                SDL_eventaction action, Uint32 mask);
```

Parametr action specifikuje, co se má vykonat. Je-li nastaven na SDL_ADDEVENT, jsou události z pole vloženy do fronty, v případě SDL_PEEKEVENT budou vráceny, ale ne vymazány. K vrácení a následnému vymazání z fronty slouží SDL_GETEVENT.

Poslední parametr definuje masku událostí, se kterými se má pracovat. Jedná se o ANDované flagy makra SDL_EVENTMASK(typ_události). Také se mohou použít přímé názvy masek, lze je najít v hlavičkovém souboru SDL_events.h. Maska pro libovolné události nese označení SDL_ALLEVENTS.

Návratová hodnota představuje počet vložených/načtených událostí, v případě chyby je vráceno -1.

Zákaz generování některých událostí

Pomocí funkce SDL_EventState() lze specifikovat, jestli se mají události daného typu vkládat do fronty, nebo ne. První parametr specifikuje typ události a druhý určuje činnost funkce. Je-li předán flag SDL_IGNORE, události se do fronty vkládat nebudou, v případě SDL_ENABLE se zpracovávají normálně a SDL_QUERY slouží k dotazům. Vrácen je vždy stav po modifikaci.

```
Uint8 SDL_EventState(Uint8 type, int state);
```

Tato funkce se uplatní především při používání technik, které zjišťují stav vstupních zařízení přímo a bylo by tedy zbytečné generovat události. Například funkce SDL_GetKeyState() umožňuje programátorovi dotázat se na stisk klávesy. K tomu však až příště.

Pomocí následujících dvou řádků lze zakázat generování všech událostí klávesnice.

```
SDL_EventState(SDL_KEYUP, SDL_IGNORE);
SDL_EventState(SDL_KEYDOWN, SDL_IGNORE);
```

Filtr událostí

O něco flexibilnější než předchozí funkce je SDL_SetEventFilter(), pomocí které lze do SDL předat ukazatel na rutinu, jenž může filtrovat události nejen podle typu, ale i podle ostatních parametrů.

```
void SDL_SetEventFilter(SDL_EventFilter filter);
SDL_EventFilter SDL_GetEventFilter(void);
```

Typ SDL_EventFilter je definován jako ukazatel na funkci, které je v parametru předán ukazatel na událost. Vrátí-li tato funkce číslo jedna, bude událost vložena do fronty, v případě nuly nebude.

```
typedef int (*SDL_EventFilter)(const SDL_Event *event);
```

Jediná připomínka vzniká u události SDL_QUIT. Filtr je pro ni volán pouze tehdy, pokud správce oken vyžaduje zavřít okno aplikace. V takovém případě vrácená jednička říká, aby bylo okno zavřeno, cokoli jiného ponechá okno otevřené (je-li to možné). Pokud je vyvoláno SDL_QUIT kvůli signálu o přerušení, filtr volán nebude a zpráva je vždy doručena přímo aplikaci.

A ještě dvě krátké poznámky: Filtr není volán na události, které vkládá do fronty sama aplikace (SDL_PushEvent(), SDL_PeepEvents()), a jelikož může být vykonáván v jiném vláknu, měli byste si dávat pozor, co v něm děláte.

V následujícím příkladu má SDL zakázáno generovat události klávesnice, pokud se nejedná o klávesu abecedního znaku nebo čísla. Je to vlastně analogie příkladu u funkce SDL_EventState(), nicméně zde byl výběr navíc omezen podle parametrů. Všimněte si, že filtr má podobnou strukturu jako samotné zpracovávání událostí.

```
#include <ctype.h>// Kvůli isalnum()

int EventFilter(const SDL_Event *event)
{
	switch(event->type)
	{
	case SDL_KEYDOWN:
		// Abecední znak nebo číslo?
		if(isalnum(event->key.keysym.sym))
			return 1;// Vložit
		else
			return 0;// Nevkládat
		break;

	default:
		break;
	}

	return 1;// Vložit
}

// Zapnutí filtru např. v Init()
SDL_SetEventFilter(EventFilter);
```

Ukázkové programy

Fronta událostí

Program demonstruje nejčastější techniky používané při práci s frontou událostí. Jsou jimi načítání a vkládání zpráv z/do fronty a filtrování událostí. Pomocí mezerníku lze zapínat a vypínat filtrování zpráv o uvolnění kláves (při příchodu se vypíše oznámení do konzole), zprávy týkající se myši se vždy ignorují. Stejně jako u příkladu z minula, ani zde se nic nevykresluje.

Klávesnice

Pravděpodobně nejpoužívanějšími vstupními zařízeními počítače jsou klávesnice a myš, v našem seriálu začneme právě klávesnicí. Podíváme se na ni jak z událostního pohledu, tak "přímým" přístupem a uděláme první krok k interaktivním hrám.

Události klávesnice

SDL definuje pro klávesnici dvě události, první je generována, když uživatel stiskne klávesu, a druhá, když ji uvolní. Parametr type objektu SDL_Event je v takovém případě nastaven na hodnotu SDL_KEYDOWN resp. SDL_KEYUP a podrobnosti o události jsou uloženy do proměnné key, což je objekt struktury SDL_KeyboardEvent.

```
typedef struct
{
        Uint8 type;
        Uint8 state;
        SDL_keysym keysym;
} SDL_KeyboardEvent;
```

Jak už bylo řečeno, type obsahuje buď hodnotu SDL_KEYDOWN nebo SDL_KEYUP. Atribut state nese naprosto stejnou informaci, ale používá pro to jména SDL_PRESSED a SDL_RELEASED, jinak žádný rozdíl.

Poslední z uvedených atributů je struktura SDL_keysym poskytující informace o stisknuté klávese. Je definována následovně.

```
typedef struct
{
        Uint8 scancode;
        SDLKey sym;
        SDLMod mod;
        Uint16 unicode;
} SDL_keysym;
```

Scancode představuje scankód, který pochází přímo od hardwaru, ale v praxi se v podstatě nepoužívá.

Naproti tomu proměnná sym, odvozená od SDLKey, je používána velice často, nese v sobě symbolické jméno stisknuté klávesy. Proměnná mod oznamuje přítomnost modifikátorů, jako jsou shift, ctrl, alt atd. Současným zkoumáním sym a mod, lze tedy velice snadno implementovat klávesové zkratky.

Poslední položka obsahuje, pokud jsou překlady zapnuté, hodnotu klávesy/znaku v kódování unicode.

Všechny uvedené skutečnosti budou podrobně rozebrány v následujícím textu...

Symbolická jména kláves

SDLKey je v hlavičkovém souboru SDL_keysym.h deklarováno jako výčtový typ, který definuje symbolická jména jednotlivých kláves. Znaky z první poloviny ASCII tabulky (do 127) jsou namapovány na odpovídající klávesy na klávesnici. Z toho plyne, že konstanta SDLK_a může být při porovnávání parametru sym nahrazena obyčejným znakem 'a' a podobně.

Všechna symbolická jména kláves začínají na předponu 'SDLK_', za kterou následuje vlastní název - SDLK_SPACE (mezerník), SDLK_RETURN (enter), SDLK_UP (šipka nahoru), SDLK_F1 (funkční klávesa F1), atd. Nejrozumnější asi bude, když si tato jména najde každý sám v SDL dokumentaci. Dole v hlavním menu je umístěn odkaz 8-1. SDL Keysym definitions.

Na stejném místě lze nalézt i definice modifikátorů z parametru mod, jejich názvy začínají na 'KMOD_' a při testech se vždy využívá funkce bitového součinu (AND). Jelikož je jich jen několik, uvedeme si je i do textu článku. Mimochodem, stejným způsobem jako SDLMod je definován i SDLKey.

```
typedef enum
{
        KMOD_NONE  = 0x0000,
        KMOD_LSHIFT= 0x0001,
        KMOD_RSHIFT= 0x0002,
        KMOD_LCTRL = 0x0040,
        KMOD_RCTRL = 0x0080,
        KMOD_LALT  = 0x0100,
        KMOD_RALT  = 0x0200,
        KMOD_LMETA = 0x0400,
        KMOD_RMETA = 0x0800,
        KMOD_NUM   = 0x1000,
        KMOD_CAPS  = 0x2000,
        KMOD_MODE  = 0x4000,
} SDLMod;

#define KMOD_CTR   (KMOD_LCTRL|KMOD_RCTRL)
#define KMOD_SHIFT (KMOD_LSHIFT|KMOD_RSHIFT)
#define KMOD_ALT   (KMOD_LALT|KMOD_RALT)
#define KMOD_META  (KMOD_LMETA|KMOD_RMETA)
```

Na následujícím příkladu se implementuje klávesová zkratka (levý)Alt+Enter, jejímž výsledkem bude přepnutí okna do fullscreenu.

```
// Zpracování událostí, stisk klávesy
case SDLK_RETURN:
        if(event.key.keysym.mod & KMOD_LALT)
                if(!ToggleFullscreen())
                        return false;
        break;
```

Pozn.: Funkce ToggleFullscreen() byla naprogramována v minulém dílu tohoto seriálu.

Unicode znaky

Na chvíli se ještě vrátíme zpět k SDL_keysym. Pokud je parametr unicode v této struktuře nenulový, pak obsahuje unicode znak, který odpovídá stisknuté klávese a je-li navíc horních devět bitů nulových, bude ekvivalentní ASCII znaku (16 - 9 = 7 ;-). SDL dokumentace obsahuje příklad demonstrující obsah tohoto odstavce.

```
char ch;
if((keysym.unicode & 0xFF80) == 0)
        ch = keysym.unicode & 0x7F;
else
        printf("Mezinárodní znak.\n");
```

Jelikož jsou překlady znaků do unicode relativně výkonově náročné, jsou v SDL standardně vypnuté. Jednička, předaná do SDL_EnableUNICODE(), podporu zapíná, nula vypíná a mínus jedničky může být využito k dotazům.

```
int SDL_EnableUNICODE(int enable);
```

Opakování událostí při držení klávesy

Windows programátory by po spuštění ukázkových příkladů mohlo teoreticky překvapit, že stisk klávesy, její déletrvající držení a uvolnění, způsobí vygenerování VŽDY DVOU událostí - zprávy o stisku a následně zprávy o uvolnění, nic dalšího.

Ve Win32 API se narozdíl od SDL první zpráva WM_KEYDOWN (analogie SDL_KEYDOWN) pošle aplikaci při stisku a pokud je klávesa držena delší dobu, následují po určitém časovém intervalu zprávy další.

SDL může být požádáno, aby se chovalo stejným způsobem. Slouží k tomu funkce SDL_EnableKeyRepeat(), jejíž parametr delay říká, za jak dlouho se má od stisku poslat první opakovací zpráva (čili druhá SDL_KEYDOWN v pořadí) a parametr interval specifikuje periodu odesílání následujících zpráv. Obě hodnoty jsou v jednotkách milisekund.

```
int SDL_EnableKeyRepeat(int delay, int interval);
```

Předání nuly do delay způsobí vypnutí opakování, což je v SDL implicitní stav. Místo zadání konkrétních hodnot, je možné časy specifikovat také symbolickými konstantami SDL_DEFAULT_REPEAT_DELAY a SDL_DEFAULT_REPEAT_INTERVAL. Funkce při úspěchu vrátí 0, jinak -1.

"Přímý" přístup ke klávesnici

Při programování her vzniká relativně často potřeba dotázat se kdykoli v programu, zda je určitá klávesa stisknutá nebo ne. Události jsou v tomto případě nepoužitelné, protože neinformují o aktuálním stavu, ale jen o jeho změnách. Proto lze v některých zdrojových kódech najít konstrukce podobné těm na následujícím výpise.

```
// Globální pole indikátorů klávesnice
// V Init() nastavit všechny položky na false
bool g_keys[MAX_KEYS];

// Události stisku a uvolnění
case SDL_KEYDOWN:
        // Nastavit indikátor dané klávesy
        g_keys[event.key.keysym.sym] = true;
        break;

case SDL_KEYUP:
        // Vynulovat indikátor dané klávesy
        g_keys[event.key.keysym.sym] = false;
        break;

// Zjištění stisku klávesy
if(g_keys[SDKL_UP])
        JdiNahoru(g_fps);
else
        NicNedelej();
```

Pozn.: Velikost globálního pole g_keys jsme definovali jako MAX_KEYS indikátorů. V některých knihovnách je zvykem jeho rozsah definovat na 256, nicméně letmý pohled do SDL_keysym.h ukáže, že tuto konstantu v SDL použít nelze, definovaných kláves je víc.

Po krátkém hledání můžeme v SDL objevit funkci SDL_GetKeyState(), jež vrací ukazatel na vnitřní pole indikátorů klávesnice (analogie našeho g_keys), které může být indexováno SDLK_* symboly. Není-li parametr funkce nastaven na NULL, SDL do něj vloží velikost tohoto pole.

```
Uint8 *SDL_GetKeyState(int *numkeys);
```

Jedničková hodnota na indexu oznamuje, že je klávesa stisknutá, v případě nuly není. Lze také použít symbolické konstanty SDL_PRESSED a SDL_RELEASED. Před samotnými dotazy na klávesy může být vhodné funkcí SDL_PumpEvents() (viz minulý díl) informace v poli aktualizovat.

Přepis kódu výše do SDL by tedy mohl vypadat následovně.

```
// Zjištění stisku klávesy
SDL_PumpEvents();

Uint8* keys;
keys = SDL_GetKeyState(NULL);

if(keys[SDLK_UP] == SDL_PRESSED)
        JdiNahoru(g_fps);
else
        NicNedelej();
```

Pomocí SDL_GetKeyState() je samozřejmě možné zjistit i přítomnost modifikátorů, většinou se však využívá služeb specializované funkce SDL_GetModState(). V jejím případě není vrácen ukazatel na pole, ale bitová maska.

```
SDLMod SDL_GetModState(void);
void SDL_SetModState(SDLMod modstate);
```

Pomocí druhé uvedené funkce lze pro program klávesu modifikátoru virtuálně stisknout.

Kdy použít události a kdy přímý přístup

Zkusíme, podobně jako u událostí výše, definovat klávesovou zkratku Alt+Enter pro přepnutí okna do fullscreenu a pak si vysvětlíme, proč není tento kód obecně použitelný.

```
// Tento kód není obecně použitelný!!!

if(keys[SDLK_RETURN] == SDL_PRESSED)
        if(SDL_GetModState() & KMOD_LALT)
                if(!ToggleFullscreen())
                        return false;
```

Výpis je ve svém principu naprosto správný, ale po zobrazení kompletního zdrojového kódu kterékoli z ukázkových aplikací zjistíme, že samotný test klávesové zkratky a tedy i přepnutí do fullscreenu je vloženo do hlavního cyklu aplikace, který se provádí neustále dokola.

Řekněme, že se právě teď nacházíme ve fullscreenu a chceme se přepnout do okna. Kód správně detekuje Alt+Enter a změní stav. Problém je, že za několik milisekund (po vykreslení a aktualizaci scény) nastane další průchod cyklem a uživatel stále drží Alt+Enter. Takže se aplikace opět přepne, tentokrát zpět do fullscreenu. To se bude opakovat neustále dokola, dokud budou obě klávesy stisknuté. Po uvolnění navíc není určen výsledek.

Uvedeme si ještě jeden obecně nepoužitelný příklad.

```
// Tento kód není obecně použitelný!!!

// Zpracování událostí klávesnice
case SDLK_UP:
        JdiNahoru(g_fps);
        break;
```

Co se stane teď? Když uživatel stiskne šipku nahoru, postavička ve hře se posune o pár pixelů nahoru, ale pak zůstane stát. Při libovolně dlouhém držení klávesy přijde jen jedna zpráva o stisku.

Z příkladů výše tedy jasně vyplývá, že na různé pohyby postaviček po scéně je vhodné používat přímý přístup ke klávesnici (rychlost pohybů vztahovat k aktuálnímu FPS) a přepínání nejrůznějších flagů ošetřovat událostmi. Obecnou platnost této poučky trochu nabourává funkce SDL_EnableKeyRepeat(), ale pokud se jí budeme držet, neměly by nastat žádné problémy...

Řetězec se jménem klávesy

Někdy může být potřebné zjistit jméno stisknuté klávesy. V SDL je to s pomocí funkce SDL_GetKeyName() velice snadné. Za parametr se předává symbolické jméno klávesy a výstupem je řetězec ukončený NULL.

```
char *SDL_GetKeyName(SDLKey key);
```

Následující kód by ve spuštěném programu zajistil výpisy jmen stisknutých kláves. Pro jednoduchost je uveden jen výpis do konzole, ale kdyby se text zobrazoval graficky do okna (např. s pomocí SDL_ttf), měli bychom k dispozici základní GUI výběru kláves pro ovládání hry.

```
// Zpracování událostí, stisk klávesy
case SDL_KEYDOWN:
        printf("%s\n", SDL_GetKeyName(
                        event.key.keysym.sym));
        break;
```

Výstup by po několika úderech na klávesnici vypadal nějak takto:

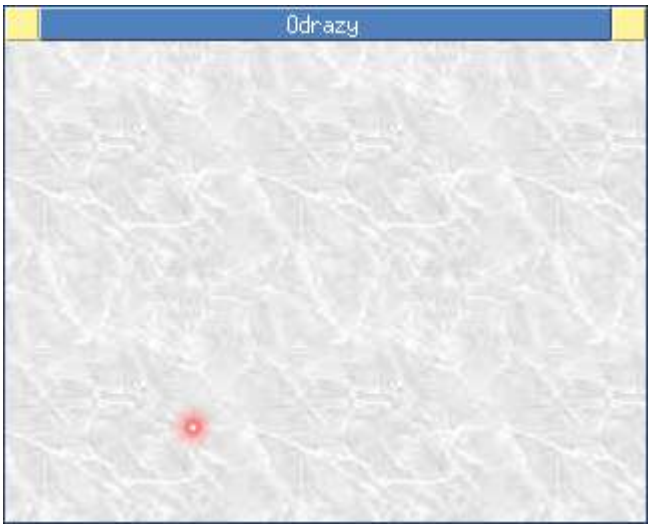
```
space           // Mezerník
return          // Enter
caps lock
left shift
left ctrl
f               // Písmeno f
down            // Šipka dolů
...
```

Ukázkové programy

Odrazy

Ukázkový program vykresluje objekt, se kterým je možno pomocí šipek (přímý přístup) pohybovat. Stisk nemění polohu přímo, ale je jím ovlivněno zrychlení, v každém průchodu je pozice zvětšována o rychlost. Také je aplikována gravitace. V případě, že objekt narazí do stěny (okraj okna), odrazí se a jeho rychlost je o něco zmenšena.

Jako bonus byl v programu implementován pomocí událostí i jeden cheat. Na klávesnici se naťuká posloupnost "cheat" a co se stane, uvidíte po spuštění ;-).


Odrazy

Myš

Na řadě je další vstupní zařízení, tentokrát se jedná o myš. Opět se budeme věnovat, jak událostem, tak přímému přístupu.

Stisk tlačítka myši

Vždy, když uživatel stiskne některé tlačítko myši, vygeneruje SDL dvě události - SDL_MOUSEBUTTONDOWN a SDL_MOUSEBUTTONUP. První z nich je odeslána při stisku a druhá při uvolnění. V obou případech se podrobnosti o události hledají v podobjektu event.button, který byl odvozen ze struktury SDL_MouseButtonEvent.

```
typedef struct
{
        Uint8 type;
        Uint8 button;
        Uint8 state;
        Uint16 x, y;
} SDL_MouseButtonEvent;
```

Atribut type je klasicky nastaven na jméno události a proměnná button ukládá jméno tlačítka, což je jedna ze symbolických konstant SDL_BUTTON_LEFT, SDL_BUTTON_MIDDLE a SDL_BUTTON_RIGHT. Ve verzi 1.2.5 SDL dále přibyla jména SDL_BUTTON_WHEELUP a SDL_BUTTON_WHEELDOWN, jenž oznamují točení rolovacím kolečkem nahoru a dolů.

Stejně jako u klávesnice, i zde může být state nastaveno na SDL_PRESSED nebo SDL_RELEASED, tuto informaci však už máme k dispozici z parametru type. Proměnné x a y poskytují pozici myši v klientské oblasti okna při stisku, bod [0, 0] se nachází v levém horním rohu.

V následujícím příkladu program zachytává stisk levého tlačítka myši a jako reakci vypíše do konzole informaci o pozici v okně.

```
// Ošetření událostí
case SDL_MOUSEBUTTONDOWN:
        switch(event.button.button)
        {
        case SDL_BUTTON_LEFT:
                printf("BUTTON_LEFT - pos(%d,%d)\n",
                                event.button.x,
                                event.button.y);
                fflush(stdout);
                break;

        default:
                break;
        }
        break;
```

Výstup programu po dvou stisknutích levého tlačítka:

```
BUTTON_LEFT - pos(65,103)
BUTTON_LEFT - pos(91,104)
```

Událost pohybu myší

Pohyb myší oznamuje SDL zprávou SDL_MOUSEMOTION, podrobnosti se následně hledají v objektu event.motion

odvozeného od SDL_MouseMotionEvent.

```
typedef struct
{
        Uint8 type;
        Uint8 state;
        Uint16 x, y;
        Sint16 xrel, yrel;
} SDL_MouseMotionEvent;
```

Proměnná state definuje stavy tlačítek při pohybu. Pro zjištění, které je stisknuté a které ne, může být výhodné použít makro SDL_BUTTON(). Parametry x a y specifikují pozici kurzoru myši v okně, xrel a yrel obsahují relativní hodnotu posunu.

Po příchodu události o pohybu myši v příkladu níže, vypíše program absolutní polohu kurzoru v okně, změnu polohy od minula a případně informaci o stisku tlačítek.

```
// Ošetření událostí
case SDL_MOUSEMOTION:
        printf("MOUSEMOTION - pos(%d,%d), relpos(%d,%d)%s%s%s\n",
                event.motion.x, event.motion.y,
                event.motion.xrel, event.motion.yrel,
                (event.motion.state & SDL_BUTTON(SDL_BUTTON_LEFT))
                                ? ", left" : "",
                (event.motion.state & SDL_BUTTON(SDL_BUTTON_MIDDLE))
                                ? ", middle" : "",
                (event.motion.state & SDL_BUTTON(SDL_BUTTON_RIGHT))
                                ? ", right" : "");
        fflush(stdout);
        break;
```

Pokud je program spuštěn, začnou se při pohybování myší generovat výpisy podobné následujícím.

```
MOUSEMOTION - pos(130,91), relpos(4,0)
MOUSEMOTION - pos(134,91), relpos(4,0)
MOUSEMOTION - pos(138,91), relpos(4,0)
MOUSEMOTION - pos(136,91), relpos(-2,0), left
MOUSEMOTION - pos(132,93), relpos(-4,2), left, right
MOUSEMOTION - pos(130,93), relpos(-2,0), left, right
MOUSEMOTION - pos(128,95), relpos(-2,2), left, right
```

"Přímý" přístup k myši

Stejně jako u klávesnice, i u myši je možné používat metody přímého přístupu. Lze se tedy kdekoli v programu dotázat na aktuální polohu kurzoru nebo stisk tlačítek.

```
Uint8 SDL_GetMouseState(int *x, int *y);
```

Tato funkce uloží na adresu ukazatelů v parametrech aktuální polohu myši v okně a vrátí bitové pole tlačítkových flagů. Pro rozlišení, které je stisknuté a které ne, je opět nejjednodušší použít bitový součin s makrem SDL_BUTTON(). Pokud nás zajímají pouze tlačítka, je možné předat do parametrů hodnoty NULL.

Před samotným přístupem k myši bývá vhodné zavolat funkci SDL_PumpEvents(), která aktualizuje informace v SDL.

Podobným způsobem se lze dotazovat i na relativní změny polohy od minulého volání této funkce nebo od zpracování události o pohybu myši.

```
Uint8 SDL_GetRelativeMouseState(int *x, int *y);
```

Do proměnných x a y bude v příkladu níže uložena aktuální poloha myši, kód v sekci if se provede jen tehdy, je-li stisknuto levé tlačítko.

```
// Kdekoli v programu
int x, y;

SDL_PumpEvents();
if(SDL_GetMouseState(&x, &y) & SDL_BUTTON(SDL_BUTTON_LEFT))
        printf("Levé tlačítko na %d, %d.\n", x, y);
```

Ruční změna polohy myši

Novou polohu myši lze specifikovat voláním funkce SDL_WarpMouse(), do parametrů se předávají požadované x a y souřadnice.

```
void SDL_WarpMouse(Uint16 x, Uint16 y);
```

Tato funkce má jeden vedlejší efekt, způsobuje generování události SDL_MOUSEMOTION, což může být někdy, kvůli zacyklení, nežádoucí (v reakci na událost se ošetří změna polohy a myš se přesune na nové místo, tím se generuje další událost, která se opět ošetří, myš se přesune atd.). Asi nejjednodušší řešení spočívá v ignorování této "přebytečné" události.

Při změnách natočení kamery ve 3D akčních hrách končí ošetření každého pohybu myši nastavením její polohy zpět na střed okna. Je to z důvodu, že kdyby opustila okno, mohlo by ztratit fokus (většinou po kliknutí na jiné okno při střelbě) a operační systém by v takovém případě přestal posílat zprávy. Hra by se každou chvíli stávala nehratelnou.

Implementace rotace kamery v závislosti na pohybech myši by mohla vypadat následovně.

```
// Ošetření událostí
case SDL_MOUSEMOTION:
        // SDL_WarpMouse() generuje SDL_MOUSEMOTION,
        // bez testu na střed okna by se aplikace zacyklila
        if(event.motion.x != GetWinWidth() >> 1
                || event.motion.y != GetWinHeight() >> 1)
        {
                m_cam.Rotate(event.motion.xrel,
                        event.motion.yrel, GetFPS());

                // Přesun zpět doprostřed okna
                SDL_WarpMouse(GetWinWidth() >> 1,
                        GetWinHeight() >> 1);
        }
        break;
```

Všimněte si především ignorování událostí, které generuje funkce SDL_WarpMouse(). Mimochodem, tento kód jsme použili v příkladu Pohyb v mřížce z osmého dílu. Jedná se o metodu QGridApp::ProcessEvent(SDL_Event& event).

Další možností by mohl být zákaz pro myš opustit okno aplikace, zbavili bychom se tak neustálého měnění její polohy a následného rozlišování validity událostí. V SDL stačí zavolat funkci SDL_WM_GrabInput() s parametrem SDL_GRAB_ON (popsána v desátém dílu), v některých jiných knihovnách však takové vymoženosti nejsou.

Barevné kurzory

Ještě jedna specialitka na závěr. V sedmém dílu jsme si ukázali, jak požádat SDL, aby změnilo kurzor myši ze standardní šipky na jiný. Tato technika však měla tu nevýhodu, že kurzor mohl být pouze černobílý.

V tuto chvíli však už máme dostatek znalostí, abychom standardní kurzor myši vypnuli a vykreslovali si vlastní, na nějž už nejsou kladena žádná omezení.

```
// Pro vycentrování obrázku na aktivní bod kurzoru
// U šipek levý horní roh, u zaměřovačů střed, apod.
#define POSUN_DOLEVA 0
#define POSUN_NAHORU 0

// Inicializace, skryje kurzor
SDL_ShowCursor(0);

// Vykreslování (kurzor by se měl vždy kreslit jako poslední)
SDL_Rect rect;

SDL_GetMouseState(&rect.x, &rect.y);
rect.x -= POSUN_DOLEVA;
rect.y -= POSUN_NAHORU;

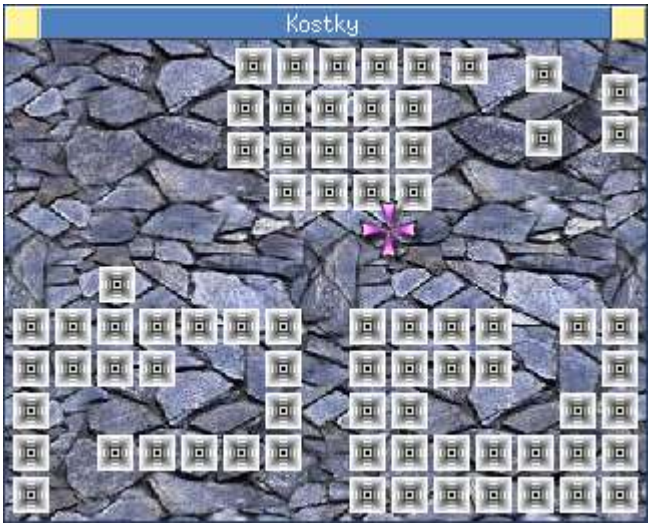
SDL_BlitSurface(g_cur_press, NULL, g_screen, &rect);
```

Tento kód předpokládá, že se scéna periodicky překresluje, nejlépe v "klasické" herní smyčce nebo v reakci na pohyb myši a pokaždé se kreslí úplně všechno.

Ukázkové programy

Kostky

Program zobrazuje v dolní části okna spoustu kostiček, které lze kliknutím myši zachytit a následně s nimi pohybovat. Jsou implementovány i kolize a také vlastní barevný kurzor, jenž se po kliknutí na nějakou kostku změní na jiný.



Joystick

Joysticky, kniply, páky a jiné ovladače bývají nedílnou součástí většiny her, hlavně simulátorů. Tento díl bude věnován právě jim.

Upozornění: Hned na začátku je potřeba říct, že jsem nikdy s žádným joystickem nepracoval a nemám ho ani k dispozici! Protože je však nedílnou součástí SDL, mělo by mu nějaké místo být věnováno. Vše, co zde tedy bude napsáno, vychází výhradně ze SDL dokumentace a bohužel není z mé strany žádným způsobem ověřeno.

Příprava Joysticku pro použití

Základním předpokladem, aby mohl být joystick v aplikaci používán, je předání symbolické konstanty SDL_INIT_JOYSTICK do parametrů funkce SDL_Init(), která inicializuje SDL.

Dalším důležitým krokem přípravy je dotaz, kolik joysticků je připojeno k počítači. SDL k tomu poskytuje funkci SDL_NumJoysticks(), její návratovou hodnotou je samozřejmě daný počet.

```
int SDL_NumJoysticks(void);
```

Víme-li, že je k počítači alespoň jeden joystick připojen, lze přistoupit k jeho otevření, které se vykoná voláním funkce SDL_JoystickOpen().

```
SDL_Joystick *SDL_JoystickOpen(int device_index);
```

Za parametr se předává index joysticku, což je v podstatě jeho pořadí v systému. Hodnoty mohou být pouze v rozmezí 0 až SDL_NumJoysticks()-1. Pomocí tohoto čísla bedeme také joystick identifikovat při zpracování událostí, ale o tom až později.

Návratovou hodnotou funkce je ukazatel na objekt struktury SDL_Joystick, který budeme předávat do všech joystickových funkcí, v případě neúspěchu pak NULL.

Kdekoli v aplikaci může být vznesen dotaz, zda je joystick otevřen nebo ne. Slouží k tomu funkce SDL_JoystickOpened(), která, je-li otevřen, vrátí jedničku, jinak nulu.

```
int SDL_JoystickOpened(int device_index);
```

Po skončení práce by měly být uvolněny všechny zdroje, které aplikace alokovala, to samé platí i pro joysticky.

```
void SDL_JoystickClose(SDL_Joystick *joystick);
```

Následující příklad demonstruje obecnou inicializaci joysticku.

```
// Globální proměnná
SDL_Joystick *g_joy = NULL;

// Inicializace
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK);

// Je vůbec nějaký joystick dostupný?
if(SDL_NumJoysticks() > 0)
{
        // Pokud ano, jeden otevře
        g_joy = SDL_JoystickOpen(0);
```

```
        if(g_joy == NULL)
                fprintf(stderr, "Nepodařilo se otevřít joystick.\n");
}

// Deinicializace
if(g_joy != NULL)
        SDL_JoystickClose(g_joy);
```

Informace o otevřeném joysticku

K získání jména joysticku slouží funkce SDL_JoystickName(), která vrací řetězec ukončený NULL. Jelikož se za parametr předává pouze index zařízení, lze tuto funkci volat ještě před vlastním otevřením joysticku. Není-li žádné jméno dostupné, je vráceno NULL.

```
const char *SDL_JoystickName(int device_index);
```

Index zařízení se z joystickové struktury získá funkcí SDL_JoystickIndex().

```
int SDL_JoystickIndex(SDL_Joystick *joystick);
```

Pomocí následujících čtyř funkcí se provádí dotazy na technické parametry daného joysticku. Funkce vrací po řadě počet os (páka), trackballů, kloboučků (POV Hat) a tlačítek.

```
int SDL_JoystickNumAxes(SDL_Joystick *joystick);
int SDL_JoystickNumBalls(SDL_Joystick *joystick);
int SDL_JoystickNumHats(SDL_Joystick *joystick);
int SDL_JoystickNumButtons(SDL_Joystick *joystick);
```

V následujícím příkladu otevřeme první joystick a vypíšeme o něm do konzole všechny informace, které se dají zjistit.

```
SDL_Joystick *joy;

if(SDL_NumJoysticks() > 0)
{
        // Otevře první joystick
        joy = SDL_JoystickOpen(0);

        if(joy)
        {
                printf("Joystick #0\n");
                printf("Jméno: %s\n",
                        SDL_JoystickName(0));
                printf("Počet os: %d\n",
                        SDL_JoystickNumAxes(joy));
                printf("Počet kloboučků: %d\n",
                        SDL_JoystickNumHats(joy));
                printf("Počet trackballů: %d\n",
                        SDL_JoystickNumBalls(joy));
                printf("Počet tlačítek: %d\n",
                        SDL_JoystickNumButtons(joy));
        }
        else
                printf("Nelze otevřít joystick #0\n");
```

```
        // Zavře joystick
        if(SDL_JoystickOpened(0))
                SDL_JoystickClose(joy);
}
```

Pokud by bylo potřeba získat informace o všech joysticích v systému, není problém vložit tento kód do cyklu.

Události joysticku

Události joysticku jsou implicitně vypnuty, a proto, aby se jejich doručování povolilo, je nutné zavolat funkci SDL_JoystickEventState(). Za její parametr může být předána jedna ze symbolických konstant SDL_QUERY, SDL_ENABLE popř. SDL_IGNORE.

```
int SDL_JoystickEventState(int state);
```

Každá z jednotlivých částí joysticku má přiřazenu vlastní událost. Jedna pro pohyb pákou, jedna pro tlačítka, další pro trackball a ještě jedna pro klobouček - celkem čtyři události. Asi nejlepší bude, když půjdeme popořadě.

Osa (páka) - SDL_JoyAxisEvent

Pokud je parametr event.type události nastaven na hodnotu SDL_JOYAXISMOTION, jedná se o pohyb pákou. Další informace se pak hledají v parametru event.jaxis, což je objekt struktury SDL_JoyAxisEvent.

```
typedef struct
{
        Uint8 type;
        Uint8 which;
        Uint8 axis;
        Sint16 value;
} SDL_JoyAxisEvent;
```

Jak brzy uvidíme, je parametr which přítomen u všech joystickových zpráv, jedná se o index joysticku, na kterém událost nastala. Axis představuje index osy, na většině moderních zařízení je osa x reprezentována nulou a y jedničkou. Value udává aktuální polohu páky, je to číslo v rozmezí od -32768 do 32767.

Tlačítka - SDL_JoyButtonEvent

Další joystickovou událostí, kterou SDL poskytuje, je stisk respektive uvolnění některého z tlačítek. Parametr type v takovém případě obsahuje buď hodnotu SDL_JOYBUTTONDOWN, nebo SDL_JOYBUTTONUP a objekt v SDL_Event má jméno event.jbutton.

```
typedef struct
{
        Uint8 type;
        Uint8 which;
        Uint8 button;
        Uint8 state;
} SDL_JoyButtonEvent;
```

Proměnná button opět obsahuje index tlačítka a state může být nastaveno na SDL_PRESSED nebo SDL_RELEASED. Tato informace už však byla získána z parametru type.

Trackball - SDL_JoyBallEvent

Další část joysticku, která může generovat události, je trackball. Jméno zprávy je nastaveno na SDL_JOYBALLMOTION a informace se hledají v event.jball, proměnné struktury SDL_JoyBallEvent.

```
typedef struct
{
        Uint8 type;
        Uint8 which;
        Uint8 ball;
        Sint16 xrel, yrel;
} SDL_JoyBallEvent;
```

Parametr ball označuje index trackballu a xrel spolu s yrel udává relativní pohyb na osách x a y. Absolutní pozici nelze, kvůli obecné podstatě trackballu, získat.

Klobouček - SDL_JoyHatEvent

Událost kloboučku má jméno SDL_JOYHATMOTION a informace jsou uloženy v event.jhat.

```
typedef struct
{
        Uint8 type;
        Uint8 which;
        Uint8 hat;
        Uint8 value;
} SDL_JoyHatEvent;
```

V parametru hat je analogicky index kloboučku a value obsahuje pozici. Jedná se o binárně ORovanou kombinaci následujících symbolických konstant. Jejich význam je jistě každému jasný.

```
SDL_HAT_CENTERED
SDL_HAT_UP
SDL_HAT_RIGHT
SDL_HAT_DOWN
SDL_HAT_LEFT
```

Dále mohou být použity také předdefinové kombinace.

```
#define SDL_HAT_RIGHTUP         (SDL_HAT_RIGHT|SDL_HAT_UP)
#define SDL_HAT_RIGHTDOWN       (SDL_HAT_RIGHT|SDL_HAT_DOWN)
#define SDL_HAT_LEFTUP          (SDL_HAT_LEFT|SDL_HAT_UP)
#define SDL_HAT_LEFTDOWN        (SDL_HAT_LEFT|SDL_HAT_DOWN)
```

"Přímý" přístup

Další možností, jak přistupovat k joysticku, jsou, stejně jako u myši nebo klávesnice, přímé dotazy na jeho stav. Mimochodem, na mnoha místech SDL dokumentace se objevují poznámky, že je lepší preferovat události.

Pokud nejsou zapnuté joystickové události, je nutné pro získání informací volat funkci SDL_JoystickUpdate(), která aktualizuje stav všech částí všech připojených joysticků. Při událostním systému je volána automaticky.

```
void SDL_JoystickUpdate(void);
```

Osa (páka)

Pro zjištění polohy páky slouží funkce SDL_JoystickGetAxis(). Za první parametr se předává ukazatel na strukturu joysticku - toto je obecné pravidlo všech funkcí pro přímý přístup.

```
Sint16 SDL_JoystickGetAxis(SDL_Joystick *joystick, int axis);
```

Druhý parametr specifikuje index osy a návratovou hodnotou je její pozice. Někdy může být nutné počítat s jistou tolerancí, která padá na účet chvění. Je zajímavé, že některé joysticky používají osy 2 a 3 coby extra tlačítka.

Následující příklad je opět přebrán ze SDL dokumentace, je v něm ukázáno, jak podle polohy páky určit směr pohybu.

```
Sint16 x_move, y_move;
SDL_Joystick *joy1;

// Inicializace joy1...

x_move = SDL_JoystickGetAxis(joy1, 0);
y_move = SDL_JoystickGetAxis(joy1, 1);
```

Tlačítka

Druhým parametrem funkce SDL_JoystickGetButton() je možné specifikovat tlačítko, jehož stav potřebujeme zjistit. Je-li stisknuto, vrátí funkce jedničku a pokud ne, nulu.

```
Uint8 SDL_JoystickGetButton(SDL_Joystick *joystick, int button);
```

Trackball

Následující funkcí lze zjistit relativní pohyb trackbalu, hodnoty se vždy vztahují k minulému volání. V případě úspěchu je vrácena nula, jinak -1.

```
int SDL_JoystickGetBall(SDL_Joystick *joystick,
                int ball, int *dx, int *dy);
```

V příkladu níže se program pokusí zjistit přírůstky relativní pozice trackballu a následně je vypsat na terminál.

```
int delta_x, delta_y;
SDL_Joystick *joy;

SDL_JoystickUpdate();

if(SDL_JoystickGetBall(joy, 0, &delta_x, &delta_y) == -1)
        printf("Chyba při čtení trackballu!\n");
else
        printf("Trackball delta - X:%d, Y:%d\n",
                        delta_x, delta_y);
```

Kloboučeк

Ke kloboučku se přistupuje funkcí SDL_JoystickGetHat(). Její návratovou hodnotou je kombinace symbolických konstant, které už byly popsány u událostí.

```
Uint8 SDL_JoystickGetHat(SDL_Joystick *joystick, int hat);
```

Force Feedback

Force Feedback bohužel není v současné době podporován. SDL dokumentace uvádí, že Sam Lantinga <slouken@libsdl.org> (autor SDL) snažně prosí osoby, které mají s těmito technikami nějaké zkušenosti, o nápady, jak co nejlépe navrhnout API.

Ukázkové programy

Jak jsem psal na začátku, s joysticky nemám naprosto žádné zkušenosti. Navíc žádný nevlastním, a proto, i kdybych něco

stvořil, nedokázal bych ověřit funkčnost a případně program odladit. Z tohoto důvodu nebude tento díl obsahovat žádný ukázkový program :-(

Nicméně jeden můj kamarád, Ladislav Zima, je lídrem nezávisleho herního vývojového týmu Zimtech. Ve hře Becher Rescue ovládání pomocí joysticků implementoval, takže pokud máte chuť, není nic snazšího, než nahlédnout do zdrojových kódů (GNU GPL). Jedná se o soubor main.cpp, od řádku 73.

Tímto bych mu chtěl také veřejně poděkovat za přečtení článku a upozornění na největší chyby, jichž jsem se dopustil.

Události joysticku/gamepadu se v Becher Rescue mapují na zpracování událostí klávesnice. Pokud je pozice páky mimo "mrtvou zónu" (dead-zone - joysticky jako analogová zařízení se nedrží přesně v nule, ale pozice páky se lehce "klepe" okolo ní), jakoby zmáčkne příslušné tlačítko na klávesnici pro pohyb postavy. Ignoruje se tedy vzdálenost páky od středu.

Ještě jedna poznámka: na mnoha joysticích a gamepadech se neposílá událost uvolnění tlačítka, takže s tím počítejte. V Becher Rescue se zmáčknutí tlačítka joysticku převede na zvednutí a opětovné stisknutí příslušné klávesy panáčka.

Ostatní události

V dnešním dílu o knihovně SDL dokončíme popis událostního systému. Budeme se mimo jiné věnovat změnám velikosti okna, jeho aktivacím a deaktivacím, posílání uživatelských zpráv a dalším věcem, které ještě zbývá probrat.

Ukončení aplikace

Požadavek na ukončení posílá operační systém/správce oken, když z nějakého důvodu požaduje po aplikaci, aby se ukončila. Ve většině případů se jedná o pokus uživatele zavřít okno programu.

Dalším důvodem pro vložení SDL_QUIT do fronty může být příchozí signál od operačního systému. SDL_Init() vždy instaluje handlery pro SIGINT (přerušení klávesnicí) a SIGTERM (požadavek na ukončení od operačního systému). Pokud programátor handlery pro tyto signály nevytvoří, jsou použity defaultní, které generují událost SDL_QUIT. U ní nelze žádným způsobem zjistit, z jakého důvodu byla poslána, ale nastavením handleru může být standardní chování přepsáno.

SDL_QUIT je nejjednodušší ze všech událostí. V SDL_Event sice existuje objekt struktury SDL_QuitEvent (event.quit), ten však neobsahuje jinou informaci, než je typ události.

```
typedef struct
{
        Uint8 type
} SDL_QuitEvent;
```

Nejvhodnější reakcí na tuto zprávu je buď rovnou ukončit aplikaci, nebo, ve výjimečných případech, začít s ukončovacím dialogem, jako je zobrazení hlavního menu hry a podobně.

Kdykoli v programu se lze pomocí SDL_QuitRequested() zeptat, jestli náhodou nebyl vznesen požadavek na ukončení, v takovém případě vrátí toto makro nenulovou hodnotu.

Změna velikosti okna

Změnu velikosti okna oznamuje SDL posláním zprávy SDL_VIDEORESIZE, která v event.resize, objektu struktury SDL_ResizeEvent, poskytuje novou šířku a výšku okna.

```
typedef struct
{
        Uint8 type;
        int w, h;
} SDL_ResizeEvent;
```

V reakci na událost by měla být zavolána funkce SDL_SetVideoMode() (viz 4. díl), která aktualizuje velikost klientské oblasti okna, do níž program kreslí.

Pozn.: V MS Windows s OpenGL způsobuje SDL_SetVideoMode() jisté problémy, viz 8. díl věnovaný OpenGL a SDL.

Pokud chceme z nějakého důvodu zakázat uživateli, aby mohl změnit velikost okna, není nic snazšího, než NEpředat funkci SDL_SetVideoMode() parametr SDL_RESIZABLE.

Požadavek na překreslení

Událostí SDL_VIDEOEXPOSE oznamuje SDL programu, že je z nějakého důvodu nutné překreslit obsah okna. Tento stav může nastat, když je okno modifikováno vně aplikace, obvykle správcem oken. Objekt struktury SDL_ExposeEvent, jenž lze najít v event.expose, neobsahuje kromě typu žádné parametry.

```
typedef struct
{
        Uint8 type
} SDL_ExposeEvent;
```

(De)aktivace okna

Přijde-li aplikaci zpráva SDL_ACTIVEEVENT, znamená to, že uživatel okno buď aktivoval nebo deaktivoval (např. minimalizace). Specifické informace se pak hledají v event.active.

```
typedef struct
{
        Uint8 type;
        Uint8 gain;
        Uint8 state;
} SDL_ActiveEvent;
```

Proměnná gain má v případě deaktivace nulovou hodnotu, jednička naopak označuje aktivaci. State může být nastaveno celkem na tři různé konstanty: SDL_APPMOUSEFOCUS, SDL_APPINPUTFOCUS a SDL_APPACTIVE.

První z nich vyjadřuje, že okno ztratilo/získalo fokus myši, což defakto znamená, že myš opustila nebo dosáhla oblasti okna. U druhé je předmětem zájmu klávesnice. Tento parametr obyčejně vyjadřuje, že se jiná aplikace stala aktivní. Konečně poslední možnost oznamuje minimalizaci, respektive obnovení minimalizovaného okna.

Podobným stylem, jako se přistupovalo ke klávesnici nebo myši, se lze dotazovat i na stav okna. Funkce SDL_GetAppState () vrací kombinaci tří, výše zmíněných, symbolických konstant.

```
Uint8 SDL_GetAppState(void);
```

Deaktivační událost se hodí například v případě, kdy kvůli animacím periodicky překreslujeme scénu. Je zbytečné, aby se tato činnost prováděla i tehdy, je-li okno minimalizované, protože uživatel nemá šanci cokoli zahlédnout.

Zachytí-li aplikace v příkladu níže minimalizaci okna (událost SDL_ACTIVEEVENT, state obsahuje SDL_APPACTIVE a gain je nulový), je zavolána funkce SDL_WaitEvent(), která uspí program. Parametr NULL v tomto případě říká, že nechceme, aby byla událost, jež probudí aplikaci, odstraněna z fronty.

```
// Zpracování událostí
case SDL_ACTIVEEVENT:
        if(event.active.state & SDL_APPACTIVE)
        {
                if(event.active.gain == 0)
                {
                        SDL_WaitEvent(NULL);
                }
        }
        break;
```

Tento kód však nemusí fungovat vždy! Je to způsobeno tím, že ve frontě může být za právě zpracovávanou událostí ještě nějaká další, která způsobí okamžité ukončení SDL_WaitEvent() a opětovné spuštění programu.

Řešení může spočívat v přesunutí SDL_WaitEvent() mimo událostní smyčku do podmínky třeba if(wait), kde wait je bool proměnná nastavená na true na stejném místě, na kterém se v tuto chvíli nachází SDL_WaitEvent(). Funkce se tedy spustí až tehdy, máme-li jistotu, že je fronta prázdná.

Mimochodem, další důležitou podmínkou, aby se dala aplikace uspat, je vypnutí všech systémových časovačů - generují zprávy, které by opět vedly k předčasnému probuzení.

Uživatelské události

Pamatujete, jak jsme v 11. díle popisovali funkci SDL_PushEvent()? Řekli jsme si, že uvnitř aplikace se neposílají standardní, ale většinou tzv. uživatelské události. Při jejich používání musí program zajistit nejen jejich zpracování, ale také posílání.

```
typedef struct
{
        Uint8 type;
        int code;
        void *data1;
        void *data2;
} SDL_UserEvent;
```

Parametr type může nabývat hodnot z rozsahu SDL_USEREVENT až SDL_NUMEVENTS-1. Vzhledem k tomu, že má SDL_USEREVENT hodnotu 24 a celkový počet je 32, není počet událostí nijak závratný. Řešením může být druhý parametr, jenž může být použit, stejně jako vše u uživatelských událostí, naprosto libovolným způsobem - čili i na rozlišení "typu" události. Tímto malým podvodem se jejich počet právě rozrostl, v případě 32 bitového procesoru, z původních osmi na několik desítek miliard (přesně $8 * 2^{32}$).

Díky tomu, že jsou další dva parametry datového typu ukazatelů na void, mohou se spolu s událostmi posílat naprosto libovolná data (resp. ukazatele na ně), jejich velikost navíc není omezena.

V následujícím příkladu pošleme funkci zpracovávající události testovací zprávu. Pro jednoduchost bude bez parametrů.

```
// Nejlépe do vyhrazeného hlavičkového souboru
#define USR_EVT_MOJE_UDALOST 0

// Kdekoli v kódu
SDL_Event event;

event.type = SDL_USEREVENT;
event.user.code = USR_EVT_MOJE_UDALOST;
event.user.data1 = NULL;// Bez parametrů
event.user.data2 = NULL;
SDL_PushEvent(&event);
```

Pozn.: Doporučuji vytvořit si nějaký speciální hlavičkový soubor, ve kterém se budou uchovávat jména/kódy všech uživatelských událostí hezky přehledně na jednom místě. U větších projektů, zvlášť když se na vývoji podílí více lidí, se pak nedostanete do situace, kdy si začnete říkat 'Ne, událost číslo 15 už bude určitě zabraná, ale 735689 by ještě mohla být volná ;-)'.

Uživatelská událost se dá ošetřit úplně stejně, jako kterákoli jiná, podrobnosti se tentokrát nacházejí v podobjektu event.user.

```
// Zpracování událostí
case SDL_USEREVENT:
        switch(event.user.code)
        {
        case USR_EVT_MOJE_UDALOST:
                // K parametrům by se přistupovalo takto:
```

```
                // param1 = (pretypovani*)event.user.data1;
                // param2 = (pretypovani*)event.user.data2;
                NecoUdelej();
                break;

        default:
                break;
        }
        break;
```

Systémově závislé události

Poslední typ událostí, které ještě zbývá probrat, jsou události závislé na systému, v němž aplikace běží. Program samozřejmě nikdy nedostává zprávy, které se běžně používají v SDL, ale jen jejich ekvivalenty poskytované daným systémem.

Například pošle-li MS Windows zprávu WM_KEY_DOWN, SDL ji přeloží na SDL_KEYDOWN a umístí do fronty zpráv. U jiného systému se událost stisku klávesy může jmenovat úplně jinak, ale díky SDL program pracuje vždy jen s obecnou zprávou SDL_KEYDOWN.

Pro události, které nemají svůj ekvivalent, poskytuje SDL speciální událost SDL_SYSWMEVENT, která je v sobě všechny zahrnuje. Podrobnosti se hledají v event.syswm. Vkládání těchto událostí do fronty je standardně vypnuté, zapnutí je umožněno klasicky pomocí funkce SDL_EventState().

```
typedef struct
{
        Uint8 type;
        SDL_SysWMmsg *msg;
} SDL_SysWMEvent;
```

Jediným parametrem události je ukazatel na objekt struktury SDL_SysWMmsg. Ta je pomocí komplikovaných příkazů preprocesoru deklarována v hlavičkovém souboru SDL_syswm.h vždy pro daný systém, na němž se program právě kompiluje. Například pro zprávy MS Windows vypadá deklarace struktury následovně (Win32 programátorům jistě povědomá).

```
struct SDL_SysWMmsg
{
        SDL_version version;
        HWND hwnd;
        UINT msg;
        WPARAM wParam;
        LPARAM lParam;
};
```

Systémovým událostem další prostor věnován nebude. Tyto techniky jsou především naprosto nepřenositelné, u kódu napsaného pro MS Windows nelze předpokládat, natož ani doufat, že půjde pod Linuxem a to samé samozřejmě platí i opačným směrem.

SDL běží na desítkách systémů, zkusíte-li implementovat danou funkčnost pro každý zvlášť, většinou skončíte se dvěma nebo maximálně se třemi nejpoužívanějšími, které jsou pro vás dostupné, a ostatní jednoduše podporovány nebudou.

Navíc, pokud se pustíte do využívání událostí závislých na systému, pravděpodobně máte dostatek znalostí, že to zvládnete i bez pomoci tohoto seriálu...

Ukázkové programy

Menu

Ukázkový program demonstruje vytvoření jednoduchého menu, které je zapouzdřené do speciální třídy (resp. do dvou). Rodičovské QMenu se stará o operace, jako je vkládání položek, pohyb v menu, posílání událostí apod., a potomek QSDLMenu o vykreslování. Nechcete-li použít pro vykreslování SDL/SDL_ttf, ale například OpenGL, není nic snazšího než naprogramovat dalšího potomka, základní funkčnost zůstává zachována v QMenu.

Pozn.: To písmenko Q na začátku, označuje třídy z jedné mé knihovničky (zatím soukromá, neustále spousta změn). Tímto programem jsem defakto zaplácl dvě mouchy jednou ranou - ukázkový program ke článku a menu do semestrální práce z C++ ;-)

Co se týče funkcí programu, tak jednotlivé položky menu umožňují výběr obrázku na pozadí, skrytí menu a ukončení aplikace. Mezi položkami se dá pohybovat šipkami a výběr zprostředkovává klávesa enter. Po jejím stisknutí generuje třída uživatelskou událost, jejíž zpracování je už na aplikaci.



Časovače a práce s časem

V dnešním díle se podíváme na systémové časovače a funkce pro práci s časem. Na konci budou také v rychlosti zmíněny rychlostní optimalizace včetně výpočtu FPS.

Systémové časovače

Pokud je nutné spouštět nějakou funkci s určitou frekvencí neustále dokola, může být výhodné využít služeb systémového časovače. Jedná se o mechanismus, kterým je možné požádat SDL, aby vždy po uplynutí určitého času spustilo předem specifikovaný kód. Při používání časovačů je nutné předat do funkce SDL_Init() v inicializaci symbolickou konstantu SDL_INIT_TIMER.

Časovač se aktivuje funkcí SDL_AddTimer(), jejíž první parametr definuje časový interval v milisekundách, po jehož uplynutí se spustí callback funkce, té bude předáván poslední parametr. Návratovou hodnotou je ID právě vytvořeného timeru nebo NULL v případě chyby.

```
SDL_TimerID SDL_AddTimer(Uint32 interval,
                SDL_NewTimerCallback callback,
                void *param);
```

Identifikátor SDL_NewTimerCallback je definován jako ukazatel na funkci se dvěma parametry, která vrací Uint32.

```
typedef Uint32 (*SDL_NewTimerCallback)(Uint32 interval, void *param);
```

Spuštěný časovač se dá zastavit funkcí SDL_RemoveTimer(), předává se mu jeho ID. Minimálně při ukončování programu může být zavolání dobrým nápadem...

```
SDL_bool SDL_RemoveTimer(SDL_TimerID id);
```

Vzhledem k tomu, že zvláště začínající programátoři mívají s ukazateli na funkce mnoho problémů, následuje kompletní ukázka kódu, který je potřeba pro zprovoznění timeru napsat. Definuje se globální proměnná ukládající ID vytvářené timeru, napíše se callback funkce a ukazatel na ni se předá spolu s libovolným parametrem do SDL_AddTimer(). Pokud nevznikne žádný problém, bude se tato funkce opakovaně volat s periodou cca. jedné sekundy (1000 milisekund). Na konci se timer zastaví.

```
// Globální
SDL_TimerID g_timer_id = NULL;

Uint32 Callback(Uint32 interval, void* param)
{
        // Uživatelský kód

        return interval;
}

// Spuštění časovače (např. inicializace)
        g_timer_id = SDL_AddTimer(1000, Callback, NULL);

// Zastavení časovače (např. deinicializace)
        SDL_RemoveTimer(g_timer_id);
```

Jak jste si jistě všimli, callback funkci je předávána hodnota zpoždění před spuštěním. Aby se docílilo periodického volání, musí být jejím výstupem interval pro příště. Ten může být buď stejný jako předešlý (většinou return interval; - viz příklad výše), nebo libovolný jiný.

V případě, že se předchozí interval s právě specifikovaným časem neshodují, SDL aktuální časovač zastaví a spustí nový s novou časovou konstantou. Vše se však děje na pozadí, takže se jako programátoři nemusíme o nic starat, stačí jen vrátit rozdílnou hodnotu.

Jelikož může být callback funkce prováděna v jiném vláknu, než běží zbytek programu, měla by spouštět výhradně thread-safe funkce (problematika vícevláknového programování bude vysvětlena v následujících dílech). Nejpohodlnějším řešením může být vygenerování uživatelské události, na níž pak zareaguje sama aplikace.

Pozn.: Časová přesnost timerů je závislá na platformě, vždy by se mělo počítat s určitou nepřesností. Co vím, tak u Win9x se udávalo cca. 55 ms a u Windows na bázi NT něco málo přes 10 ms. Jedná se však o minimální hodnoty, většinou bývají nepřesnosti vzhledem k zatížení systému mnohem vyšší.

Například ve zmíněných MS Windows jsou timery implementovány posíláním zprávy WM_TIMER. Problémem je, že pokud se už ve frontě tato událost nachází, není do ní nikdy vložena znovu. Tudíž, kdyby aplikace kontrolovala frontu řekněme jednou za sekundu (extrém) a timer byl nastaven na periodu 20 ms, dostávala by aplikace stále jen jednu zprávu za sekundu a ostatní by byly ignorovány.

Zpoždění

Vykonávání programu se dá pozastavit voláním funkce SDL_Delay(), které se předá požadovaný čas v milisekundách. Tato doba bude však z technických důvodů vždy o něco delší.

```
void SDL_Delay(Uint32 ms);
```

Volání SDL_Delay() umožní operačnímu systému přidělit čas CPU i ostatním procesům, resp. program jím říká, že mu po specifikované časové údobí nemusí přidělovat žádný čas procesoru a má ho raději věnovat běhu ostatních procesů/programů, protože by stejně nic nedělal.

Pozn.: Z minulého odstavce jste jistě vytušili, že generovat časové zpoždění pomocí tří vnořených cyklů není zrovna nejšťastnější nápad... ;-)

Volání této funkce s intervalem větším než, řekněme, jedna sekunda také není moc vhodné. Program je kompletně pozastaven a tudíž nereaguje na žádné uživatelské vstupy, nepřekresluje okno a nedělá zkrátka vůbec nic, co dělá v normálním režimu.

První, co napadne uživatele sedícího před monitorem, je, že se ten ***** program zase zaseknul, a v podstatě má pravdu. Takže začne chaoticky klikat na ukončovací křížek, mačkat nejrůznější klávesové zkratky a i když byl program pozastaven záměrně, příchozí SDL_QUIT po obnovení do normálního režimu, popř. systémový kill, ho dozajista ukončí.

Zjištění uplynulého času

Funkce SDL_GetTicks() vrací tzv. referenční čas, u něhož nás nezajímá ani tak hodnota (v tomto případě počet milisekund od inicializace SDL), jako rozdíl hodnot ze dvou volání funkce, který se použije pro výpočet např. posunutí objektu v čase na novou pozici.

```
Uint32 SDL_GetTicks(void);
```

Mimochodem, pozor na přetečení datového typu po cca. 49 dnech, pokud je možné, že program poběží tak dlouho.

SDL_GetTicks() netrpí podobným neduhem, jako funkce s analogickým určením z některých jiných knihoven. Např. často

používaná GetTickCount() z Win32 API vrací "konstantní" hodnotu, která se vždy po uplynutí ~55 milisekund skokově aktualizuje. Mimochodem, aby nevznikl flame, ve Windows je možné použít tzv. Performance counter, který je výrazně přesnější než obyčejný GetTickCount().

Rychlostní optimalizace her

Většina her potřebuje nějakým způsobem zajistit, aby byly všechny pohyby a animace stejně rychlé na všech počítačích, na kterých poběží. Bez zpětné vazby bude jistě rozdíl, když se hra vyvíjená na 300 MHz počítači spustí na 3 GHz systému.

V případě, že program implementuje klasickou herní smyčku, může být rozdíl hodnot ze dvou po sobě jdoucích volání výše zmíněné funkce SDL_GetTicks() použit pro rychlostní optimalizace. Vždy se pracuje výhradně s diferencí současného času a času průchodu stejným místem v minulosti. Ukázka bude asi názornější.

```
// Globální proměnná
Uint32 g_last_time = 0;

// Hlavní smyčka programu
bool done = false;
while(!done)
{
        Uint32 dt = SDL_GetTicks() - g_last_time;

        // Zbytečně malý interval (~100 FPS)
        if(dt < 20)
        {
                // Nechá něco i ostatním procesům
                SDL_Delay(10);
                dt = SDL_GetTicks() - g_last_time;
        }

        g_last_time = SDL_GetTicks();

        ProcessEvent(); // Události
        Update(dt);     // Aktualizace scény
        Draw();         // Překreslení
}
```

Všimněte si, že aktualizační funkci Update() se předává vypočtená hodnota časové diference. Násobení změny pozice diferencí času způsobí, že všechny pohyby budou při spuštění i třeba na desetkrát rychlejším počítači pro uživatele vždy stejné.

```
int g_xpos, g_ypos;

void Update(Uint32 dt)
{
        g_xpos += 0.01 * dt;
        g_ypos += 0.01 * dt;
}
```

Jedná-li se o výkonný počítač, je dt nízké a přírůstek pozice kvůli násobení menší, než na pomalém systému, nicméně bude aplikován častěji. U výrazně pomalého počítače budou přírůstky vysoké, ale kvůli malé frekvenci začne docházet k trhání pohybů. Pak existují jen dvě možnosti: buď se pokusit o optimalizace programu, nebo upgrade počítače.

Výpočet FPS

Existují dva základní způsoby, jak vypočítat počet překreslení scény za sekundu (FPS). První napadne asi každého, v každém průchodu hlavní smyčkou inkrementovat čítač a otestovat, jestli už uplynul čas jedné sekundy od začátku počítání. Pokud ano, obsahuje čítač požadovanou hodnotu FPS.

Druhou možností je použít matematiku a počítat FPS dynamicky. Máme-li k dispozici rozdíl časů mezi dvěma překresleními, stačí se zeptat kolikrát by se vešly do jedné sekundy.

```
float fps = 1000.0f / dt;
```

Vždy byste se měli ujistit, že dt neobsahuje nulu. V minulém příkladu by se nic vážného nestalo, ale tady by došlo k dělení nulou. Aktualizační funkce s využitím FPS bude vypadat následovně, objekt se bude pohybovat v obou souřadnicových osách rychlostí 100 pixelů za sekundu.

```
int g_xpos, g_ypos;

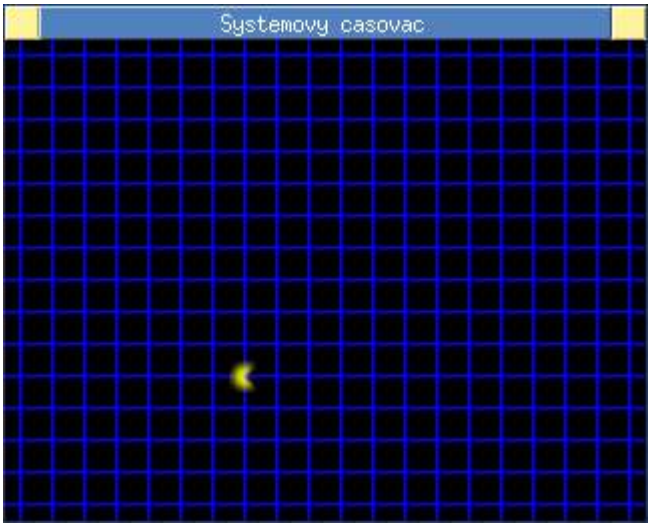
void Update(float fps)
{
        g_xpos += 100.0f / fps;
        g_ypos += 100.0f / fps;
}
```

Je nutné podotknout, že ať už se pohyby objektů regulují pomocí fps nebo dt, výsledek bude vždy stejný. FPS je ale možná o něco přirozenější, také neříkáte, že auto ujede 0.01 metrů za x (mili)sekund, ale že jeho rychlost je 100 km/h.

Ukázkové programy

Systémový časovač

Dnešní ukázkový program tvoří základ pro hru ve stylu Pacmana. Na pozadí je dlaždicově vykreslena mřížka, ve které se pohybuje hráč ovládaný šipkami. Pohyby jsou implementovány pomocí systémového timeru, který lze zrychlit/zpomalit klávesami +/-.



Ukázku na herní smyčku s FPS optimalizacemi lze najít například v ukázkovém příkladu z dvanáctého dílu, ale i mnoha dalších.

Zvuky a hudba

V dnešním díle o knihovně SDL začneme nový tematický celek, budou jím zvuky a hudba, které přinesou konec všem tichým aplikacím.

Zvuky v počítači

Asi to už budete znát, ale pro jistotu uvedu alespoň velice stručný úvod do zpracování audia počítačem. Zvuk je ve své podstatě vlnění - analogová veličina, která se na digitální signál převádí tzv. vzorkováním. Vždy po uplynutí předem stanoveného časového intervalu se odebere "vzorek" zvuku, což je číslo udávající hodnotu signálu v daném okamžiku, a uloží se do paměti.

Při posílání posloupnosti vzorků na zvukovou kartu jsou vytvářeny aproximace původních zvukových vln. Výslednou kvalitu tedy ovlivňuje jak frekvence, se kterou byly vzorky odebrány, tak jejich velikost. Běžně podporovaným audio formátem je 16 bitový vzorek na 22 kHz, což je kompromis mezi výslednou kvalitou a velikostí potřebné paměti pro uložení.

SDL a audio

Knihovna SDL poskytuje nízkoúrovňový přístup pro práci s audiem navržený jako základ pro implementaci softwarového zvukového mixeru. Na první pohled by se mohlo zdát, že si musí naprostou většinu funkčnosti napsat programátor sám, nicméně je možné používat již hotový mixer v podobě rozšiřující knihovny SDL_mixer, který odstraní většinu námahy.

Pokud tedy z nějakého důvodu potřebujete nízkoúrovňový přístup, zvolte si samotné SDL, a pokud preferujete jednoduše volatelné funkce, vždy můžete jít cestou SDL_mixeru. My se budeme věnovat oběma způsobům. Začneme funkcemi, které poskytuje SDL.

Podobně, jako u grafiky, i zvuků lze zvolit ovladač, který bude zprostředkovávat přehrávání. Jejich dostupnost je závislá především na nainstalování v systému a pak také konfiguračními volbami při kompilaci SDL.

Zvolení konkrétního ovladače se dá provést přiřazením jména požadovaného ovladače do systémové proměnné SDL_AUDIODRIVER, o všechny podrobnosti se postará SDL. V Linuxu jsou dostupné například dsp, dma, esd, artsc, ve Win32 dsound a waveout.

V běžící aplikaci se dá jméno aktuálně používaného driveru zjistit voláním funkce SDL_AudioDriverName(). Předává se jí alokovaná paměť, do které se má informace uložit, a proti přetečení také její velikost.

```
char *SDL_AudioDriverName(char *namebuf, int maxlen);
```

Předpokladem pro to, aby šly v SDL aplikaci zvuky vůbec používat, je předání symbolické konstanty SDL_INIT_AUDIO do inicializační funkce SDL_Init(). Pak je samozřejmě nutné nahrát do aplikace nějaké zvuky a případně je zkonvertovat do požadovaného formátu. Teprve potom se může přistoupit k otevření audio zařízení a samotnému přehrávání.

Menší zvláštností oproti jiným knihovnám je, že se musí definovat tzv. callback funkce, kterou bude SDL volat pokaždé, když zvukové kartě dojdou data a bude nutné poslat do streamu nová data.

Struktura SDL_AudioSpec

Tato struktura se používá ke specifikaci formátu audia, používá se především při otevírání zařízení a také při nahrávání zvuků a jejich konverzích na požadovaný formát.

```
typedef struct
{
        int freq;
```

```
        Uint16 format;
        Uint8 channels;
        Uint16 samples;
        Uint8 silence;
        Uint32 size;
        void (*callback)(void *userdata, Uint8 *stream, int len);
        void *userdata;
} SDL_AudioSpec;
```

Atribut freq udává počet vzorků za sekundu (čili frekvenci vzorkování), běžnými hodnotami jsou 11025, 22050 a 44100. Format specifikuje formát audio dat, může nabývat hodnot několika symbolických konstant (viz SDL manuál), které určují, zda je hodnota vzorků osmi nebo šestnácti bitová, znaménková/bezznaménková apod. Channels udává počet oddělených zvukových kanálů, jednička označuje mono a dvojka stereo.

Proměnná samples ukládá velikost bufferu v počtu vzorků. Toto číslo by mělo být mocninou dvou a může být audio ovladačem upraveno na pro hardware vhodnější hodnotu. Většinou se volí z rozmezí od 512 do 8192 v závislosti na rychlosti procesoru. Nižší hodnoty mají kratší reakční čas, ale mohou způsobit podtečení v případě, že zaneprázdněná aplikace nestíhá plnit buffer. Stereo vzorek se skládá z obou kanálů v pořadí levý-pravý a jejich počet je vztažen k času podle vzorce ms = (počet vzorků * 1000) / frekvence.

Silence a size jsou definovány automaticky. V prvním případě se jedná o hodnotu uloženou v bufferu, která reprezentuje ticho a v druhém jeho velikost v bytech.

Callback představuje ukazatel na funkci, kterou SDL volá, když je audio zařízení připraveno přijmout nová data. Předává se jí ukazatel na buffer/stream, jehož délka se rovná len. Userdata je libovolný ukazatel na dodatečná data.

```
void callback(void *userdata, Uint8 *stream, int len);
```

Vzhledem k tomu, že plnění bufferu většinou běží v odděleném vláknu, měl by být přístup k datovým strukturám chráněn pomocí dvojice SDL_LockAudio() a SDL_UnlockAudio(). Je zaručeno, že po locknutí až do unlocku nebude callback volána a v žádném případě by naopak neměly být použity v callback funkci.

```
void SDL_LockAudio(void);
void SDL_UnlockAudio(void);
```

Otevření audio zařízení

Audio zařízení se otevírá pomocí SDL_OpenAudio(), kterému se v prvním parametru předává požadovaný formát. Funkce se pokusí tuto konfiguraci najít a výsledek hledání uloží do obtained, které se tak stává pracovní konfigurací. Ta může být následně použita například pro konverzi všech zvuků do hardwarového formátu. V případě úspěchu je vrácena nula, jinak -1.

```
int SDL_OpenAudio(SDL_AudioSpec *desired,
                  SDL_AudioSpec *obtained);
```

Bylo-li obtained volajícím nastaveno na NULL, budou se, v případě nedostupnosti požadovaného formátu, provádět automatické realtimové konverze do formátu podporovaného hardwarem.

Aby se mohly bezpečně inicializovat data pro callback funkci plnící buffer, je po otevření audio implicitině pozastaveno. Spuštění zvukového výstupu se docílí zavoláním SDL_PauseAudio(0).

```
void SDL_PauseAudio(int pause_on);
```

Na aktuální stav se lze dotázat funkcí SDL_GetAudioStatus(), která vrací výčtový typ obsahující SDL_AUDIO_STOPPED,

SDL_AUDIO_PAUSED nebo SDL_AUDIO_PLAYING.

```
SDL_audiostatus SDL_GetAudioStatus(void);
```

Po skončení práce s audiem by se nemělo zapomenout na jeho deinicializaci.

```
void SDL_CloseAudio(void);
```

Otevření audio zařízení by mohlo vypadat například takto.

```
// Prototyp callback funkce
void AudioCallback(void *userdata, Uint8 *stream, int len);

// Inicializace
SDL_AudioSpec desired, obtained;

desired.freq = 22050;           // FM Rádio kvalita
desired.format = AUDIO_S16LSB;  // 16-bit signed audio
desired.channels = 1;           // Mono
desired.samples = 8192;         // Velikost bufferu
desired.callback = AudioCallback;// Ukazatel na callback
desired.userdata = NULL;        // Žádná uživatelská data

// Otevře audio zařízení
if(SDL_OpenAudio(&desired, &obtained) == -1)
{
        fprintf(stderr, "Unable to open audio: %s\n",
                        SDL_GetError());
        return false;
}

// Příprava callbacku na hraní
// Například loading všech zvuků, jejich konverze apod.
// Obtained obsahuje aktuální konfiguraci

// Spustí zvukový výstup
SDL_PauseAudio(0);

// Konec aplikace
SDL_CloseAudio();
```

Nahrávání zvuků

Samotné SDL umí nahrávat pouze .WAV formát souborů, který vzhledem ke své velikosti není pro praktické použití zrovna vhodný. V praxi se proto pro loading zvuků používají rozšiřující knihovny. Například SDL_sound nebo SDL_mixer si umí poradit i s .MID, .MP3, .OGG a dalšími běžně používanými formáty.

```
SDL_AudioSpec *SDL_LoadWAV(const char *file,
                SDL_AudioSpec *spec,
                Uint8 **audio_buf,
                Uint32 *audio_len);
```

První parametr funkce SDL_LoadWAV() představuje diskovou cestu k souboru se zvukem, na adresu spec bude uložen formát nahraných audio dat. Samotná data se uloží do automaticky alokované paměti, jejíž adresa bude spolu s délkou předána zpět v posledních dvou parametrech.

Funkce vrátí NULL, pokud nelze soubor otevřít, používá nepodporovaný formát nebo je poškozen. Typ chyby se dá zjistit následným SDL_GetError().

Po skončení práce se zvukem je vždy nutné pomocí SDL_FreeWAV() uvolnit alokovaná data.

```
void SDL_FreeWAV(Uint8 *audio_buf);
```

Příklad nahrání zvuku ze souboru...

```
// Globální proměnné
Uint8 *g_sound_data;    // Ukazatel na data
Uint32 g_sound_len;     // Délka dat

// Např. inicializace
SDL_AudioSpec spec;     // Formát dat

if(SDL_LoadWAV("test.wav", &spec, &g_sound_data,
                &g_sound_len) == NULL)
{
        fprintf(stderr, "Unable to load test.wav: %s\n",
                        SDL_GetError());
        return false;
}

// Úklid
SDL_FreeWAV(g_sound_data);
```

Mixování zvuků

SDL poskytuje funkci SDL_MixAudio(), která umí smixovat dva zvuky se stejným formátem. První dva parametry představují cílový a zdrojový buffer, len je délka v bytech a volume označuje hlasitost. Ta může nabývat hodnot od nuly do SDL_MIX_MAXVOLUME (definováno jako 128) a je vhodné ji nastavit na maximum.

```
void SDL_MixAudio(Uint8 *dst, Uint8 *src, Uint32 len, int volume);
```

Plnění audio bufferu

Jak už bylo zmíněno na začátku, SDL je při práci se zvuky velmi nízkoúrovňové, a proto je programátor nucen napsat si i vlastní plnění audio bufferu. Ukazatel na tuto funkci se předává do SDL při otevírání audio zařízení a volání je v podstatě automatické.

Kód převezmeme z ukázkového programu, který se dodává společně se SDL (adresář test). Jeho výsledkem bude zvuk přehrávaný v nekonečné smyčce.

```
Uint8   *g_sound_data;  // Ukazatel na data zvuku
Uint32  g_sound_len;    // Délka dat
int     g_sound_pos;    // Pozice při přehrávání

void AudioCallback(void *userdata, Uint8 *stream, int len)
{
        // Ukazatel na část, kde se má začít přehrávat
        Uint8 *wave_ptr = g_sound_data + g_sound_pos;

        // Délka zvuku do konce
        int wave_left = g_sound_len - g_sound_pos;
```

```
        // Zbývající délka je menší než požadovaná
        // Cyklus, protože celý zvuk muže být kratší
        while(wave_left <= len)
        {
                // Pošle data na zvukovou kartu
                SDL_MixAudio(stream, wave_ptr, wave_left,
                                SDL_MIX_MAXVOLUME);

                // Posune se o právě zapsaná data
                stream += wave_left;
                len -= wave_left;

                // Od začátku zvuku
                wave_ptr = g_sound_data;
                wave_left = g_sound_len;
                g_sound_pos = 0;
        }

        // Zbývající část zvuku je delší než požadovaná
        SDL_MixAudio(stream, wave_ptr, len,
                        SDL_MIX_MAXVOLUME);
        g_sound_pos += len;
}
```

Ukázkové programy

SDL a audio

Větší část dnešního programu se objevila už ve článku, jedná se tedy o aplikaci přehrávající v nekonečné smyčce zvuk nahraný z .wav souboru. Mezerníkem je možné zvukový výstup dočasně pozastavit a pomocí +/- se dá měnit hlasitost.

Konverze zvuků, knihovna SDL_sound

V tomto díle konverzemi zvuků dokončíme popis funkcí, které SDL poskytuje pro audio. Druhá část článku bude věnována rozšiřující knihovně SDL_sound, která slouží pro dekódování zvuků z .MP3, .MID, .OGG a dalších běžně rozšířených typů souborů.

Konverze zvuků

Transformace zvuků z jednoho formátu na jiný je v SDL dvoustupňový proces. Nejprve je nutné vytvořit objekt struktury SDL_AudioCVT, nastavit ho na správné parametry a nakonec ho předat jako parametr do konverzní funkce.

```
typedef struct
{
        int needed;
        Uint16 src_format;
        Uint16 dest_format;
        double rate_incr;
        Uint8 *buf;             // Buffer s daty
        int len;                // Délka originálu
        int len_cvt;
        int len_mult;           // Výpočet délky pro alokaci
        double len_ratio;       // Výpočet výsledné délky
        void (*filters[10])(struct SDL_AudioCVT *cvt, Uint16 format);
        int filter_index;
} SDL_AudioCVT;
```

Většina atributů struktury může být považována za privátní, budeme se proto zabývat jen těmi, které jsou důležité pro používání.

Buf je ukazatelem na zvuk a to jak zdrojový, tak cílový. Původní data tedy budou konverzí přepsána novými. Druhým důsledkem je, že se data mohou při konverzi zvětšit, a tudíž je nutné alokovat dostatek paměti. Číselně by měla být velká len*len_mult bytů, kde len představuje velikost původních dat a len_mult obsahuje násobitel kolikrát se mohou maximálně zvětšit, typickým příkladem je konverze 8-bitového zvuku na 16-bitový.

Len_ratio má podobný význam jako len_mult. Výsledkem násobení len*len_ratio bude po úspěšné konverzi opravdová délka nových dat v bytech.

Předtím než může být objekt SDL_AudioCVT použit pro konverzi, musí být inicializován informacemi o zdrojovém a cílovém formátu. K tomu slouží funkce SDL_BuildAudioCVT(), jejíž parametry jsou stejné jako u struktury SDL_AudioSpec probrané v minulém díle. Informace o zdrojovém zvuku jsou dostupné z načítání a u cíle se v naprosté většině případů volí formát hardwaru ze SDL_OpenAudio().

```
int SDL_BuildAudioCVT(SDL_AudioCVT *cvt,
                Uint16 src_format, Uint8 src_channels, int src_rate,
                Uint16 dst_format, Uint8 dst_channels, int dst_rate);
```

Funkce v případě úspěchu vrátí 1 a v případě neúspěchu -1. Byla-li úspěšná, může se do parametru len konverzní struktury přiřadit délka originálních dat, alokovat paměť pro buffer o velikosti len*len_mult bytů a zkopírovat do něj data zvuku. Pro samotnou konverzi se pak zavolá funkce SDL_ConvertAudio(), jejímž jediným parametrem je konverzní struktura. Úspěch označuje vrácená 0 a neúspěch -1.

```
int SDL_ConvertAudio(SDL_AudioCVT *cvt);
```

Pokud vše proběhne bez problémů, budou výsledná data uložena v atributu buf struktury a jejich délka bude len*len_ratio bytů.

V prvním ukázkovém programu naleznete obecně použitelnou funkci LoadSound(), která nahraje zvuk ze souboru filename, zkonvertuje ho pomocí právě popsané techniky na libovolný formát a výsledek uloží na adresu svého posledního parametru. Kód není vložen přímo do článku kvůli relativně velké délce.

Knihovna SDL_sound

SDL_sound je knihovna určená pro nahrávání zvuků mnoha populárních formátů a jejich dekódování. Aktuálně podporovanými podle dokumentace jsou

- .WAV (Microsoft WAVfile RIFF data, interně)
- .VOC (Creative Labs' Voice formát, interně)
- .MP3 (MPEG-1 Layer 3, prostřednictvím SMPEG a mpglib)
- .MID (MIDI hudba konvertovaná na Waveform data, interně)
- .MOD (MOD formát, prostřednictvím MikMod a ModPlug)
- .OGG (Ogg formát, prostřednictvím Ogg Vorbis knihoven)
- .SPX (Speex formát, prostřednictvím libspeex)
- .SHN (Shorten formát, interně)
- .RAW (Raw zvuková data v jakémkoli formátu, interně)
- .AU (Sun's Audio formát, interně)
- .AIFF (Audio Interchange formát, interně)
- .FLAC (Lossless audio komprese, prostřednictvím libFLAC)

Knihovna je šířena pod licencí GNU LGPL, nicméně externí dekodéry mohou mít licenci jinou. Asi nejlepší bude, když si přečtete soubor COPYING z kořenového adresáře archivu knihovny a následně jednotlivé licence všech v aplikaci používaných formátů.

Při použití je nutné přidat k parametrům linkeru řetězec -lSDL_sound, který způsobí přilinkování knihovny k programu. Hlavičky všech funkcí jsou umístěny v souboru SDL_sound.h a jejich jména začínají na jednotnou předponu 'Sound_'. Pokud nebude v textu uvedeno jinak, bude daná funkce vracet při chybě nulu, jinak nenulovou hodnotu.

Následující funkce jsou podobné svým SDL analogiím. Mělo by stačit uvést, že Sound_Init() by mělo být voláno jako první ze všech Sound_*() funkcí. Naproti tomu Sound_Quit() uvolní všechny systémové prostředky alokované knihovnou SDL_sound a mělo by být umístěno v kódu vždy před SDL_Quit().

```
int Sound_Init(void);                   // Inicializace
int Sound_Quit(void);                   // Deinicializace
const char *Sound_GetError(void);       // Vrátí chybový řetězec
void Sound_ClearError(void);            // Vynuluje ho
```

Zjištění, které dekodéry jsou aktuálně dostupné, lze provést funkcí Sound_AvailableDecoders(), která vrací ukazatel na pole struktur s informacemi o dekodérech. Poslední položka je zarážkou a má hodnotu NULL.

```
const Sound_DecoderInfo **Sound_AvailableDecoders(void);

typedef struct
{
        const char **extensions;        // Přípona souboru
        const char *description;        // Popis dekodéru
        const char *author;             // Autor
```

```
        const char *url;                // URL dekodéru
} Sound_DecoderInfo;
```

Loading zvuků

Zvukový soubor v libovolném podporovaném formátu se do aplikace dá nahrát jednou ze dvou níže uvedených funkcí. První z nich slouží pro nahrávání ze SDL_RWops (SDL abstrakce nad vstupními daty), parametr ext je pouze nápovědou při hledání vhodného dekodéru. Druhá funkce slouží pro načítání z diskového souboru.

```
Sound_Sample *Sound_NewSample(SDL_RWops *rw,
                const char *ext,
                Sound_AudioInfo *desired,
                Uint32 bufferSize);

Sound_Sample *Sound_NewSampleFromFile(const char *fname,
                Sound_AudioInfo *desired,
                Uint32 bufferSize);
```

Parametrem desired lze určit do jakého formátu má být zvuk při dekódování zkonvertován. V případě, že nejsou konverze potřeba, může být nastaven na NULL. Všechny tři atributy mají ve struktuře SDL_AudioSpec své analogie, takže k inicializace stačí pouze tři jednoduchá přiřazení.

```
typedef struct
{
        Uint16 format;  // Formát zvuku
        Uint8 channels; // 1 - mono, 2 - stereo
        Uint32 rate;    // Frekvence (vzorky za sekundu)
} Sound_AudioInfo;
```

Poslední parametr, bufferSize, určuje počáteční velikost čtecího bufferu v bytech. Čím je větší, tím více dekódování může být provedeno v jednom bloku, na druhou stranu bude trvat o něco déle a bude zabráno více zdrojů. Pro různé formáty mohou být vhodné jiné hodnoty, každopádně velikost musí být vždy násobkem velikosti vzorku. Pokud používáte například 16-bitové stereo, kde zabírá každý vzorek 2*2 bytů, musí být velikost násobkem 4.

Obě funkce vrací ukazatel na objekt struktury Sound_Sample, která je pro SDL_sound důležitá asi stejně, jako SDL_Surface pro SDL. Tento objekt ukládá všechny informace o zvukových datech a stavu jejich dekódování. Atributy by měly být považovány za READ-ONLY, pro jejich změny se využívají výhradně API funkce.

```
typedef struct
{
        void *opaque;                   // Interní použití
        const Sound_DecoderInfo *decoder;// Používaný dekodér
        Sound_AudioInfo desired;        // Formát pro konverze
        Sound_AudioInfo actual;         // Aktuální formát vzorku
        void *buffer;                   // Buffer dekódovaných dat
        Uint32 buffer_size;             // Velikost bufferu v bytech
        Sound_SampleFlags flags;        // Flagy vzorku
} Sound_Sample;
```

Po skončení práce by se měly všechny používané zdroje uvolnit. Slouží k tomu funkce Sound_FreeSample(), stačí jí předat ukazatel na nahraný zvuk.

```
void Sound_FreeSample(Sound_Sample *sample);
```

Dekódování dat

Voláním funkce Sound_Decode() se dekódují ze vzorku v pořadí následující data. Jejich velikost bude většinou sample->buffer_size bytů a budou uložena do sample->buffer. Návratová hodnota dává informaci kolik bytů bylo skutečně nahráno.

```
Uint32 Sound_Decode(Sound_Sample *sample);
```

Pokud nelze nahrát všech sample->buffer_size bytů, informace o důvodu se dají najít v sample->flags. Většinou se jedná o konec streamu (SOUND_SAMPLEFLAG_EOF) nebo o nějakou chybu (SOUND_SAMPLEFLAG_ERROR).

```
if(sample->flags & SOUND_SAMPLEFLAG_ERROR)
        NecoUdelej();
```

Pro dekódování všech zvukových dat slouží funkce Sound_DecodeAll(), která do sample->buffer dynamicky alokuje potřebnou paměť a uloží do ní výsledná data, sample->buffer_size bude obsahovat jejich velikost. Opět by se měly testovat flagy ze sample->flags.

```
Uint32 Sound_DecodeAll(Sound_Sample *sample);
```

Při dekódování celého zvuku najednou si raději dávejte pozor na velikost alokované paměti, sami jistě přijdete na to, jak by to dopadlo u půlhodinové mp3.

Změna velikosti čtecího bufferu se dá uskutečnit funkcí Sound_SetBufferSize(). Pro hodnotu nové velikosti platí stejné zásady, jako u Sound_NewSample().

```
int Sound_SetBufferSize(Sound_Sample *sample, Uint32 new_size);
```

V případě, že se nedá velikost změnit, bude se pracovat i nadále s původní. Při zkrácení budou data na konci zahozena a při prodloužení bude konec bufferu nedefinovaný do té doby, než se nahrají nová data.

Skoky na nové pozice

Základním skokem je přesun na začátek zvuku, který se vykoná funkcí Sound_Rewind(). Teoreticky by k chybě nemělo nikdy dojít.

```
int Sound_Rewind(Sound_Sample *sample);
```

Druhá funkce, Sound_Seek(), umožňuje přesun na libovolné místo definované časem v milisekundách od začátku.

```
int Sound_Seek(Sound_Sample *sample, Uint32 ms);
```

Některé dekodéry nemusejí přeskoky vůbec podporovat a některé pouze s určitými soubory, proto byste měli před dotazem otestovat flagy na SOUND_SAMPLEFLAG_CANSEEK. Pokud přesun selže, měl by se zvuk chovat, jako by volání nikdy nebylo provedeno. Při neošetřitelné chybě jsou flagy nastaveny na SOUND_SAMPLEFLAG_ERROR.

Ukázkové programy

Konverze zvuků

Program demonstruje přehrávání více zvuků najednou. Jeden bude hudbou ve smyčce na pozadí a druhý bude spouštěn vždy po stisku mezerníku. Při nahrávání funkcí LoadSound() se zvuky zkonvertují na stejný (hardwarový) formát.

SDL_sound

Program ukazuje nahrávání zvuku ve formátu .AU pomocí knihovny SDL_sound, jeho dekódování a následné přehrávání

(opět pro jednoduchost smyčka). Je zde použit .AU, ale naprosto stejným způsobem lze nahrávat zvukové soubory jakýchkoli jiných podporovaných formátů (.MP3, .OGG, atd.), stačí jen změnit jméno souboru.

Přehrávání zvuků pomocí SDL_mixer

Vše, co se týká SDL audio funkcí už máme probráno, takže se zkusíme podívat na rozšiřující knihovnu SDL_mixer.

SDL_mixer

Knihovna SDL_mixer poskytuje snadno použitelné funkce pro mixování zvuků a hudby. Je vhodná obzvlášť pro ty, kterým připadá standardní SDL audio API příliš nízkoúrovňové a strohé. Aby bylo se SDL_mixerem vše co nejjednodušší, přímo v knihovně lze nalézt podporu i pro nahrávání zvuků z formátů jako jsou .WAV, .AIFF, .VOC, .OGG, .MP3 a další, u některých ale jen s použitím externích dekodérů.

SDL_mixer je, stejně jako SDL, šířen pod licencí GNU LGPL. Veškeré rozhraní je deklarováno v hlavičkovém souboru SDL_mixer.h a při linkování programu je nutné přidat do příkazové řádky řetězec -lSDL_mixer. Všechny funkce z této knihovny začínají na předponu 'Mix_'.

Při zakládání aplikace byste si měli vždy rozmyslet, zda budete pro zvuky a hudbu používat SDL_mixer, nebo SDL audio funkce. Kombinace obou technik je sice možná, ale nemusí být zrovna šťastným nápadem. U SDL_mixeru byste se měli rozhodně vyvarovat volání funkcí jako SDL_OpenAudio(), SDL_CloseAudio(), SDL_PauseAudio(), SDL_LockAudio(), SDL_UnlockAudio() apod., mohou být konfliktní s jejich Mix_*() analogiemi.

Obecné funkce

Inicializace knihovny a současně i otevření audio zařízení se v SDL_mixeru provádí funkcí Mix_OpenAudio().

```
int Mix_OpenAudio(int frequency, Uint16 format,
                int channels, int chunksize);
```

Parametry by měly být jasné. Místo konkrétní frekvence může být použita symbolická konstanta MIX_DEFAULT_FREQUENCY, která má hodnotu 22050 Hz. Za druhý parametr, formát vzorků, je možné předat libovolnou z konstant definovaných v SDL, popř. MIX_DEFAULT_FORMAT má hodnotu AUDIO_S16SYS. U třetího parametru označuje jednička mono a dvojka stereo, chunksize definuje velikost každého mixovaného bloku.

Funkce oznamuje chybu vrácením -1, je-li vše v pořádku, vrátí nulu.

Zavření zařízení a deinicializace knihovny jsou uskutečňovány funkcí Mix_CloseAudio(). Mix_GetError() slouží pro získání řetězce obsahujícího popis poslední chyby a pomocí Mix_SetError() ji lze definovat.

```
void Mix_CloseAudio();

char *Mix_GetError();
void Mix_SetError(const char *fmt, ...);
```

V následujícím příkladu je otevřeno audio zařízení s předdefinovanou frekvencí a formátem, bude se jednat o stereo, a velikost bloků dat bude 1024 bytů.

```
// Inicializace
if(Mix_OpenAudio(MIX_DEFAULT_FREQUENCY,
                MIX_DEFAULT_FORMAT, 2, 1024) == -1)
{
        printf("Mix_OpenAudio(): %s\n", Mix_GetError());
        return false;
}

// Deinicializace
```

```
Mix_CloseAudio();
```

Parametry aktuálně otevřeného audio zařízení lze získat voláním funkce Mix_QuerySpec(), která při chybě vrací nulu.

```
int Mix_QuerySpec(int *frequency, Uint16 *format, int *channels);
```

Zvuky

Zvuky jsou v SDL_mixeru reprezentovány strukturou Mix_Chunk, jejíž popis není pro programování důležitý. Mělo by stačit předávat do funkcí ukazatele na její objekty.

Pro základní nahrávání zvuků ze souboru slouží funkce Mix_LoadWAV(). Předává se jí jméno souboru na disku a v případě úspěchu je vrácen ukazatel na zvuk, chyba je oznámena klasicky pomocí NULL. Druhá funkce, Mix_LoadWAV_RW(), se používá pro nahrávání ze SDL_RWops. Nenulová hodnota ve druhém parametru způsobí automatické zavření a uvolnění zdroje, když už není potřeba.

```
Mix_Chunk *Mix_LoadWAV(char *file);
Mix_Chunk *Mix_LoadWAV_RW(SDL_RWops *src, int freesrc);
```

Další možností, jak nahrávat zvuky do aplikace jsou dvě funkce Mix_QuickLoad_*(), které umožňují pracovat s daty v paměti. Už v době volání funkce musejí být zvuky ve výstupním formátu a v podstatě většinu chyb není testována, použití může být tudíž relativně nebezpečné. Vždy byste měli vědět, co děláte.

```
Mix_Chunk *Mix_QuickLoad_WAV(Uint8 *mem);
Mix_Chunk *Mix_QuickLoad_RAW(Uint8 *mem);
```

Po skončení práce s daným zvukem by se měl vždy pomocí Mix_FreeChunk() uvolnit.

```
void Mix_FreeChunk(Mix_Chunk *chunk);
```

Hlasitost zvuku se dá softwarově nastavit voláním Mix_VolumeChunk(), které se předává požadovaná hlasitost v rozsahu od nuly do MIX_MAX_VOLUME (=128). Návratovou hodnotou je předchozí hlasitost.

```
int Mix_VolumeChunk(Mix_Chunk *chunk, int volume);
```

Jak poznáme dále, je tato funkce pouze jednou z možností, jak nastavit výslednou hlasitost zvukového výstupu.

Kanály a přehrávání

SDL_mixer defaultně alokuje celkem osm kanálů, v každém z nich může být v jednom okamžiku přehráván právě jeden zvuk. Pokud je spuštěn zvuk v kanálu, ve kterém se už něco přehrává, původní se zastaví a je spuštěn nový. Počet kanálů se dá upravit funkcí Mix_AllocateChannels().

```
int Mix_AllocateChannels(int numchans);
```

Pokud bude požadovaný počet kanálů menší než původní, budou nejvyšší zavřeny a uvolněny. Návratovou hodnotou je počet nově alokovaných kanálů.

Podobně, jako šlo nastavit hlasitost zvuku, lze navíc nastavit i hlasitost kanálu. Za první parametr se předává pořadové číslo kanálu, v případě zadání -1 se operace provede nad všemi (platí obecně). Druhým parametrem se definuje požadovaná hlasitost.

```
int Mix_Volume(int channel, int volume);
```

Návratovou hodnotou je aktuální hlasitost kanálu, popř. průměr, pokud byly zvoleny všechny.

Příkaz na přehrávání zvuku poskytuje funkce Mix_PlayChannel(). Prvním parametrem se opět volí kanál, -1 zde má význam prvního volného kanálu. Druhý parametr specifikuje přehrávaný zvuk a třetí počet opakování + 1 (nula jedenkrát, jedna dvakrát atd.). Speciální hodnotou je -1, označuje nekonečnou smyčku.

```
int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops);

int Mix_PlayChannelTimed(int channel, Mix_Chunk *chunk,
                int loops, int ticks);
```

U druhé uvedené funkce je možné v posledním parametru zadat navíc časový limit v milisekundách. Pokud po uplynutí této doby zvuk stále hraje, bude automaticky ukončen.

Obě funkce vracejí číslo kanálu, ve kterém bude zvuk přehráván nebo -1 na znamení chyby. Většinou se jedná o to, že žádný kanál nebyl volný, ale může se také jednat o kritickou chybu.

Další možností, jak spustit přehrávání zvuku je funkce Mix_FadeInChannel(). Za poslední parametr se předává časový interval v milisekundách, během kterého zvuk postupně nabývá na síle.

```
int Mix_FadeInChannel(int channel, Mix_Chunk *chunk,
                int loops, int ms);
int Mix_FadeInChannelTimed(int channel, Mix_Chunk *chunk,
                int loops, int ms, int ticks);
```

Zvuk přehrávaný v kanálu se dá pozastavit voláním funkce Mix_Pause() a následně obnovit pomocí Mix_Resume(). Pro kompletní stopnutí slouží jedna ze tří následujících funkcí. Mix_HaltChannel() zastaví přehrávání ihned, Mix_ExpireChannel() až po uplynutí časového intervalu a Mix_FadeOutChannel() způsobí pro postupné odeznívání.

```
void Mix_Pause(int channel);
void Mix_Resume(int channel);

int Mix_HaltChannel(int channel);
int Mix_ExpireChannel(int channel, int ticks);
int Mix_FadeOutChannel(int channel, int ms);
```

Možná je to zbytečné připomínat, ale zvláště u těchto funkcí se lze velice často setkat s parametrem -1, který operuje nad všemi kanály.

Pomocí Mix_ChannelFinished() lze předat SDL_mixeru ukazatel na funkci, která se pak automaticky spustí, kdykoli se přehrávání v jakémkoli kanálu ukončí. Nikdy by se v ní neměly volat žádné audio funkce!

```
void Mix_ChannelFinished(void (*channel_finished)(int channel));
```

V následujícím příkladu se po ukončení přehrávání vypíše informační zpráva.

```
// Callback funkce
void ChannelFinishedCallback(int channel)
{
        printf("Kanál %d ukončil přehrávání.\n", channel);
}

// Předání ukazatele na funkci
Mix_ChannelFinished(ChannelFinishedCallback);
```

Následující skupina funkcí slouží pro dotazy na stav přehrávání, nemělo by být třeba je zdlouhavě popisovat. Pokud je za kanál předáno číslo -1, vrací první dvě funkce počet kanálů, které vyhověly dotazům. Poslední uvedená funkce vrátí při dotazu jednu z konstant MIX_FADING_IN, MIX_FADING_OUT nebo MIX_NO_FADING. Parametr -1 zde není validní.

```
int Mix_Playing(int channel);
int Mix_Paused(int channel);
Mix_Fading Mix_FadingChannel(int which);
```

Poslední funkce tohoto článku, Mix_GetChunk(), vrací ukazatel na zvuk, který je přehráván v definovaném kanálu, popř. byl přehráván jako poslední. Jelikož už může být zvuk uvolněn, nemusí být ukazatel v dané chvíli validní!

```
Mix_Chunk *Mix_GetChunk(int channel);
```

Ukázkové programy

SDL_mixer

Aplikace inicializuje SDL_mixer, otevře audio zařízení a nahraje zvuk. Začátek přehrávání je umožněn stiskem mezerníku a konec pomocí enteru. V obou případech je ponechána doba tří sekund na postupné nabírání na síle resp. odeznění.

Hudba a efekty

Ve 20. díle dokončíme popis knihovny SDL_mixer. Budeme se bavit především o hudbě a speciálních efektech, jako je nastavení rozdílné hlasitosti levého a pravého kanálu nebo simulace ztišení vlivem vzdálenosti zdroje zvuku od posluchače.

Hudba

Přehrávání hudby je v SDL_mixeru kompletně oddělené od normálních zvukových kanálů a tudíž musí mít své vlastní rozhraní. Hudba je reprezentován strukturou Mix_Music a stejně jako u zvuků i zde stačí znát pouze jméno, veškeré operace se provádějí pomocí API funkcí.

Pozn.: Rozhraní zvuků a hudby si je velice podobné, a proto zkusím popis trochu urychlit. Také nemám rád copy & paste články...

Hudba se nahrává funkcí Mix_LoadMUS() a po skončení práce se uvolňuje pomocí Mix_FreeMusic().

```
Mix_Music *Mix_LoadMUS(const char *file);
void Mix_FreeMusic(Mix_Music *music);
```

Samotné přehrávání začíná až po zavolání jedné z následujících tří funkcí. U poslední z nich se nezačíná hned od začátku, ale od libovolného místa skladby, jeho definice je ale bohužel závislá na typu zvukového souboru - viz Mix_SetMusicPosition() níže.

```
int Mix_PlayMusic(Mix_Music *music, int loops);
int Mix_FadeInMusic(Mix_Music *music, int loops, int ms);

int Mix_FadeInMusicPos(Mix_Music *music, int loops,
                int ms, double position);
```

Další způsob, jak přehrávat hudbu, je technika ne nepodobná SDL. Je nutné napsat mixovací funkci a poté na ni předat pomocí Mix_HookMusic() do SDL_mixeru ukazatel. V callbacku nikdy nevolejte SDL_mixer funkce ani SDL_LockAudio ().

```
void Mix_HookMusic(void (*mix_func)(void *udata,
                Uint8 *stream, int len), void *arg);
```

Parametry jsou v podstatě stejné jako u SDL callback funkce, do udata se bude předávat ukazatel z arg. Ten se také dá kdykoli zjistit pomocí Mix_GetMusicHookData().

```
void *Mix_GetMusicHookData();
```

Následující tři funkce slouží pro změnu hlasitosti, pozastavení a následné obnovení přehrávání.

```
int Mix_VolumeMusic(int volume);
void Mix_PauseMusic();
void Mix_ResumeMusic();
```

Funkci Mix_RewindMusic() lze použít pro skok na začátek hudby, ale pracuje pouze s typy .MOD, .OGG, .MP3 a nativním .MIDI.

```
void Mix_RewindMusic();
```

Pokud je to možné, pak Mix_SetMusicPosition() skočí na libovolné místo hudby, v případě úspěchu vrátí 0, neúspěch

oznamuje -1 (většinou způsobeno nepodporováním v dekodéru). Parametr position je závislý na typu zdroje, například u .OGG má význam pozice od začátku, ale u .MP3 je vztažen k aktuální pozici. Obě hodnoty jsou měřeny v sekundách.

```
int Mix_SetMusicPosition(double position);
```

Jedním z těch zajímavějších příkazů je Mix_SetMusicCMD(), který umožňuje použít pro přehrávání hudby libovolný přehrávač nainstalovaný v systému. Za parametr by měl být předán kompletní příkaz, jako kdyby se psal do příkazové řádky, na konec bude automaticky dosazeno jméno souboru s hudbou. Pro návrat zpět k internímu přehrávači stačí za parametr předat hodnotu NULL. Při úspěchu je vrácena nula, jinak -1.

```
int Mix_SetMusicCMD(const char *command);
```

Aby se dala hudba korigovat, musí daný přehrávač podporovat ovládání pomocí signálů SIGTERM (zastavení), SIGSTOP (pauza) a SIGCONT (obnovení). Změna hlasitosti nemá u externího přehrávače žádný efekt, smyčky lze implementovat opakovaným spouštěním. Tento příkaz není úplně portovatelný!

Následující ukázka je převzata z dokumentace SDL_mixeru.

```
Mix_Music *music = NULL;

if(Mix_SetMusicCMD("mpg123 -q") == -1)
{
        perror("Mix_SetMusicCMD");
}
else
{
        music = Mix_LoadMUS("music.mp3");
        if(music)
                Mix_PlayMusic(music, 1);
}
```

Přehrávání se dá zastavit funkcemi Mix_HaltMusic() a Mix_FadeOutMusic(). Pomocí třetí uvedené funkce lze předat do SDL_mixeru ukazatel na libovolnou callback funkci daného typu, která se automaticky spustí po zastavení hudby. Opět by se v ní neměly objevit žádné SDL_mixer příkazy ani SDL_LockAudio().

```
int Mix_HaltMusic();
int Mix_FadeOutMusic(int ms);
void Mix_HookMusicFinished(void (*music_finished)());
```

Zdrojový formát hudby se dá zjistit voláním funkce Mix_GetMusicType(), která vrátí jednu z konstant MUS_NONE, MUS_CMD, MUS_WAV, MUS_MOD, MUS_MID, MUS_OGG, MUS_MP3. Za parametr se může předat libovolný objekt s hudbou, u NULL se předpokládá dotaz na právě přehrávanou.

```
Mix_MusicType Mix_GetMusicType(const Mix_Music *music);
```

Použití tohoto příkazu je výhodné například spolu s Mix_SetMusicPosition() a podobnými funkcemi, které se s různými typy chovají odlišně.

A opět nezbytné funkce na dotazy...

```
int Mix_PlayingMusic();
int Mix_PausedMusic();
Mix_Fading Mix_FadingMusic();
```

Efekty

Následující funkce vykonávají na audio výstupu speciální efekty. Lze je použít buď výhradně na jeden kanál, pak se za první parametr dosazuje jeho číslo, nebo na kompletní zvukový výstup, tj. všechny kanály + hudba (tzv. postefekt), k tomu slouží symbolická konstanta MIX_CHANNEL_POST.

Je důležité si uvědomit, že používání efektů zvyšuje relativně velkou měrou náročnost aplikace. Například prohození levého a pravého stereo kanálu na výstupu za to vůbec nemusí stát, zaměnění bedniček na stole bude rozhodně efektivnější. Některé interní efekty mohou, v případě, že je definována systémová proměnná MIX_EFFECTSMAXSPEED, snížit svou náročnost, ale zároveň i kvalitu.

Všechny funkce pro efekty vracejí při chybě nulu, už to nebude dále uváděno.

Jedním ze základních efektů je definice rozdílné hlasitosti levého a pravého kanálu funkcí Mix_SetPanning(). Hlasitost se tentokrát specifikuje v rozmezí od 0 do 255.

```
int Mix_SetPanning(int channel, Uint8 left, Uint8 right);
```

Tento efekt pracuje výhradně ve stereo režimu. Aby byla celková hlasitost vždy stejná, je dobré označit jednu hodnotu za referenční a druhou k ní vztáhnout, jako je to ukázáno na následujícím příkladu.

```
// Optimální nastavení hlasitosti
Mix_SetPanning(channel, left, 255 - left);
```

Předání hlasitostí 255 za obě hodnoty bude mít za následek odregistrování (vypnutí) efektu.

Pomocí funkce Mix_SetDistance() lze simulovat změnu hlasitosti v závislosti na vzdálenosti posluchače od zdroje zvuku. Za parametr distance se dosazují hodnoty od 0 (nejblíže, nejhlasitěji) do 255 (nejdále, nejtišeji). I z největší vzdálenosti bude zvuk trochu slyšet.

```
int Mix_SetDistance(int channel, Uint8 distance);
```

Velice jednoduchý 3D zvuk lze emulovat funkcí Mix_SetPosition(). Pro parametr určující vzdálenost platí stejná pravidla, jako u Mix_SetDistance(). Úhel se definuje ve stupních v rozmezí plného úhlu, čili 360 stupňů. Číslo 0 odpovídá zvuku zepředu, 90 zprava, 180 zezadu a 270 zleva, mezihodnoty jsou samozřejmě možné, nicméně blízké hodnoty (podle dokumentace 1-7, 8-15 atd.) budou mít stejné účinky.

```
int Mix_SetPosition(int channel,
                Sint16 angle, Uint8 distance);
```

Předáním nuly za úhel i vzdálenost se efekt odregistruje. Mimochodem, pokud hledáte komplexnější techniky pro 3D audio, můžete zkusit například knihovnu OpenAL, rozhraním se velice podobá grafické OpenGL.

Funkcí Mix_SetReverseStereo() lze docílit prohození levého a pravého kanálu, nicméně zaměnění bedniček na stole bude mnohem méně náročné. Za flip je možné dosadit libovolnou nenulovou hodnotu, nulou se efekt odregistruje.

```
int Mix_SetReverseStereo(int channel, int flip);
```

Kromě předdefinovaných efektů, které byly právě popsány, poskytuje SDL_mixer rozhraní pro tvorbu libovolných nových. V podstatě stačí napsat dvě funkce a zaregistrovat je. Právě probrané interní efekty pracují naprosto stejným způsobem.

První z funkcí, které je potřeba naprogramovat, vykonává samotný efekt. SDL_mixer jí bude v parametrech předávat číslo kanálu, na němž je prováděna, ukazatel na buffer se zvukovými daty, jejich délku a uživatelský parametr. Úkolem je v podstatě načíst data ze streamu, nějakým způsobem je upravit a vložit je zpět.

```
typedef void (*Mix_EffectFunc_t)(int chan, void *stream,
                int len, void *udata);
```

Mějte na paměti, že data ve streamu už nejsou ve formátu specifikovaném v Mix_OpenAudio(), ale ve formátu zvukového zařízení, ten sice může, ale nemusí být stejný. Pro zjištění aktuálně používaného formátu slouží funkce Mix_QuerySpec().

Druhá uživatelská funkce se volá, když kanál dokončil přehrávání, byl zastaven nebo dealokován, popř. efekt byl odregistrován. Jejím úkolem je například resetovat interní proměnné nebo uvolnit dynamickou paměť. Pokud nic z toho není potřeba, je možné předávat místo ní hodnotu NULL.

```
typedef void (*Mix_EffectDone_t)(int chan, void *udata);
```

Efekt se registruje funkcí Mix_RegisterEffect(). Předává se jí číslo kanálu, ukazatele na funkce, které budou efekt vykonávat, a ukazatel na uživatelská data. Na jeho adresu se například mohou ukládat stavové parametry.

```
int Mix_RegisterEffect(int chan, Mix_EffectFunc_t f,
                Mix_EffectDone_t d, void *arg);
```

Efekty jsou vnitřně realizovány jako spojový seznam, při registrování se vždy vkládají na konec a vždy se spouští nad výstupy svých předchůdců. Nic nebrání tomu, aby byl jeden efekt registrován vícekrát, v takovém případě se účinky kumulují.

Po ukončení přehrávání je vždy spojový seznam daného kanálu resetován. Při každém volání Mix_PlayChannel*() je tedy nutné všechny efekty opětovně zaregistrovat.

Pomocí Mix_UnregisterEffect() lze libovolný efekt odregistrovat, předává se jí ukazatel na vykonávací funkci. Stejný ukazatel se hledá ve spojovém seznamu a odstraněn je vždy první nalezený výskyt, ostatní zůstávají zachovány.

```
int Mix_UnregisterEffect(int channel, Mix_EffectFunc_t f);
```

Funkce vrátí nulu například, pokud není kanál validní nebo nebyl efekt registrován. Pokud tedy chcete odregistrovat všechny efekty daného typu, není nic jednoduššího, než vložit volání této funkce do cyklu.

Úplně všechny efekty aplikované na kanál lze odregistrovat funkcí Mix_UnregisterAllEffects(), při jakékoli chybě je opět vrácena nula.

```
int Mix_UnregisterAllEffects(int channel);
```

Funkce Mix_SetPostMix() je v podstatě analogií Mix_RegisterEffect(MIX_CHANNEL_POST, ...), ale aplikuje se až na kompletní zvukový výstup, tedy po všech efektech registrovaných klasickou cestou, smixování zvukových kanálů a hudby dohromady a všech postefektech. Ihned po vykonání se stream posílá na audio zařízení, takže pokud plánujete implementovat grafické vizualizace, jste na správném místě.

```
void Mix_SetPostMix(void (*mix_func)(void *udata,
                Uint8 *stream, int len), void *arg);
```

Činnost callback funkce neskončí, dokud není audio zařízení zavřeno nebo není místo mixovací funkce předáno NULL.

Skupiny kanálů

Možná vám bude připadat seskupování kanálů relativně zbytečné, ale v některých případech se může teoreticky hodit. Díky němu lze aplikovat operace, jako je pauza nebo zastavení, na všechny kanály v dané skupině. Mimochodem s jednou skupinou jsme se už setkali, měla číslo -1 a obsahovala všechny kanály.

Při popisu funkcí typu Mix_PlayChannel(), jsme si řekli, že se při zadání parametru -1 použije libovolný volný kanál. Funkcí Mix_ReserveChannels() se rezervují kanály, aby je nebylo možno takto náhodně vybrat.

```
int Mix_ReserveChannels(int num);
```

Za parametr se předává požadovaný počet kanálů, ty se rezervují od nuly do num-1. Předání nuly rezervaci zruší. Návratovou hodnotou je počet opravdu rezervovaných kanálů, ten může být v závislosti na počtu alokovaných menší, než je požadováno.

Přidání kanálu do skupiny se vykoná voláním funkce Mix_GroupChannel(). První parametr označuje kanál a druhý jméno skupiny, může jím být libovolné kladné číslo včetně nuly. Opačný směr, tj. odebrání ze skupiny, se provede zadáním -1, v podstatě se kanál vloží do globální. Druhá funkce operuje nad více po sobě jdoucími kanály najednou.

```
int Mix_GroupChannel(int which, int tag);
int Mix_GroupChannels(int from, int to, int tag);
```

První z funkcí vrací při úspěchu 1 a při neúspěchu 0, druhá počet přidaných kanálů. Počet kanálů ve skupině se dá zjistit funkcí Mix_GroupCount().

```
int Mix_GroupCount(int tag);
```

První dostupný/nehrající kanál ve skupině lze najít voláním funkce Mix_GroupAvailable(). Mix_GroupOldest() slouží pro nalezení momentálně nejdéle hrajícího kanálu a Mix_GroupNewer() hledá nejnovější.

```
int Mix_GroupAvailable(int tag);
int Mix_GroupOldest(int tag);
int Mix_GroupNewer(int tag);
```

V případě, že není žádný kanál nalezen je vráceno číslo -1.

Odeznívání a následné ukončení přehrávání kanálů sdružených do skupiny lze provést pomocí Mix_FadeOutGroup(), čas se opět zadává v milisekundách. Druhá uvedená funkce způsobí okamžité zastavení.

```
int Mix_FadeOutGroup(int tag, int ms);
int Mix_HaltGroup(int tag);
```

Ukázkové programy

Hudba a efekty

Program demonstruje přehrávání hudby a hlavně zabudované efekty v SDL_mixeru. Cesta k hudbě se předává jako parametr programu. Ovládání:

- [+/-] - hlasitost
- [mezerník] - prohodí levý a pravý kanál
- [šipka doleva/doprava] - výstup z levého/pravého kanálu
- [šipka nahoru/dolů] - vzdálenost od zdroje zvuku
- [1,2,3,4,6,7,8,9] - pozice zdroje zvuku (úhel)

Zvuky ve hře

Program rozšiřuje ukázkový příklad ze 16. dílu (hra ve stylu Pacmana). Do scény jsou přidány objekty, které má hráč za úkol sbírat (reset pomocí R) a nějaké ty zvuky. Hudba je volitelná, stačí odkomentovat jedno define a nastavit cestu k

libovolnému souboru.



CD-ROM

Další oblastí knihovny SDL, kterou si popíšeme, bude API pro práci s CD-ROM. Po přečtení tohoto článku byste měli být schopni si vytvořit jednoduchý CD přehrávač, jenž zahrnuje přehrávání a pauzy, listování a pohyb ve skladbách a také vysouvání mechaniky pro vložení nového disku.

Inicializace CD-ROM

Aby bylo možné přehrávat hudbu z CD, je nejprve nutné ve funkci SDL_Init() zapnout symbolickou konstantou SDL_INIT_CDROM podporu CD mechanik. Tím je defakto základní inicializace hotová a může se přistoupit k vlastnímu programování.

Dále se musí položit dotaz, je-li v počítači vůbec nějaká CD mechanika, bez ní to opravdu nepůjde :-]. Funkce SDL_CDNumDrives() poskytuje jejich celkový počet a pokud je nutné zjistit také systémová jména, lze použít SDL_CDName(). Za parametr se jí předávají čísla z rozsahu od nuly do počtu všech mechanik a vrací ukazatel na řetězec jako je /dev/cdrom, D:\ apod. Jedná se pouze o informativní hodnotu.

```
int SDL_CDNumDrives(void);
const char *SDL_CDName(int drive);
```

Mechanika s daným pořadovým číslem, nultá je defaultní v systému, se otevře funkcí SDL_CDOpen(). Návratová hodnota představuje ukazatel na objekt struktury SDL_CD, které se budeme podrobně věnovat níže. Identifikátor se po skončení práce uvolňuje funkcí SDL_CDClose().

```
SDL_CD *SDL_CDOpen(int drive);
void SDL_CDClose(SDL_CD *cdrom);
```

Poslední otevřená mechanika se stane v aplikaci defaultní. To znamená, že se na ni lze při volání funkcí odkazovat argumentem NULL, který je předán místo identifikátoru SDL_CD.

Informace o CD

Po vložení CD do mechaniky je nutné funkcí SDL_CDStatus() aktualizovat obsah CD_ROM struktury.

```
CDstatus SDL_CDStatus(SDL_CD *cdrom);
```

Návratová hodnota poslouží pro získání stavových informací. Je jí výčtový typ, který může nabývat hodnot CD_TRAYEMPTY, CD_STOPPED, CD_PLAYING, CD_PAUSED a CD_ERROR. SDL také poskytuje makro CD_INDRIVE() podávající informaci, zda je v mechanice nějaký disk.

```
#define CD_INDRIVE(status) ((int)status > 0)

// Použití
if(CD_INDRIVE(SDL_CDStatus(cdrom)))
        ZacniPrehravat();
```

Struktura SDL_CD jednoznačně identifikuje otevřenou CD mechaniku a také podává informace o disku, který je do ní vložený. První dvě položky jsou dostupné obecně, ostatní jsou validní pouze s CD. Aktuálnost obsahu je dána posledním voláním funkce SDL_CDStatus().

```
typedef struct
{
        int id;                 // Privátní identifikátor
```

```
        CDstatus status;        // Stavové informace

        int numtracks;
        int cur_track;
        int cur_frame;
        SDL_CDtrack track[SDL_MAX_TRACKS+1];
} SDL_CD;
```

CD je obecně organizováno do jedné nebo více stop (v angličtině track), v naprosté většině případů je v každé z nich uložena jedna skladba. Stopy se dále skládají z určitého počtu framů, což jsou bloky dat o velikosti cca. 2 kB, které představují základní stavební jednotky CD. Při normální rychlosti se za jednu sekundu přehraje 75 framů. Tato hodnota je v SDL definována jako symbolická konstanta CD_FPS.

Numtracks uchovává počet stop na disku, cur_track obsahuje číslo aktuálně přehrávané stopy a cur_frame číslo přehrávaného framu ve stopě. Samozřejmě nesmí chybět ani pole stop. Jeho velikost je definována staticky konstantou SDL_MAX_TRACKS rovnající se 99.

Struktura SDL_CDtrack popisuje jednotlivé stopy na CD, pro pochopení by měly stačit komentáře.

```
typedef struct
{
        Uint8 id;       // ID stopy
        Uint8 type;     // SDL_AUDIO_TRACK nebo SDL_DATA_TRACK
        Uint16 unused;
        Uint32 length;  // Délka stopy ve framech
        Uint32 offset;  // Offset ve framech od začátku disku
} SDL_CDtrack;
```

SDL interně pracuje s framy, nicméně časové údaje lze velice jednoduše získat makrem FRAMES_TO_MSF(). Prvním argumentem je počet framů, který se má převést na čas, a do M, S, F budou uloženy minuty, sekundy a zbylý počet framů. Opačný směr je také možný - MSF_TO_FRAMES().

```
#define FRAMES_TO_MSF(f, M,S,F) \
{                               \
        int value = f;          \
        *(F) = value % CD_FPS;  \
        value /= CD_FPS;        \
        *(S) = value % 60;      \
        value /= 60;            \
        *(M) = value;           \
}

#define MSF_TO_FRAMES(M, S, F)  \
                ((M)*60*CD_FPS+(S)*CD_FPS+(F))
```

Pokud by byla potřeba pouze délka v sekundách, stačí jen obyčejné dělení frames / CD_FPS. V následujícím příkladě se vypíší informace o délce všech stop na CD.

```
int min, sec, fr;

SDL_CDStatus(g_cdrom);// Aktualizace informací
printf("Počet stop: %d\n", g_cdrom->numtracks);

for(int i = 0; i < g_cdrom->numtracks; i++)
```

```
{
        FRAMES_TO_MSF(g_cdrom->track[i].length, &min, &sec, &fr);
        printf("Stopa %d: %5d = %d:%2d, %2d\n", i,
                        g_cdrom->track[i].length, min, sec, fr);
}
```

Výstup bude u audio CD vypadat nějak takto.

```
Počet stop: 9
Stopa 0: 10087 = 2:14, 37
Stopa 1: 10568 = 2:20, 68
Stopa 2: 18475 = 4:06, 25
Stopa 3: 16778 = 3:43, 53
Stopa 4: 11059 = 2:27, 34
Stopa 5:  5423 = 1:12, 23
Stopa 6: 15167 = 3:22, 17
Stopa 7:  5851 = 1:18,  1
Stopa 8: 11906 = 2:38, 56
```

Přehrávání

Základní funkcí pro přehrávání je SDL_CDPlay(). Předává se jí počáteční frame a počet, který se má celkově přehrát. Pokud bude úspěšná vrátí nulu, jinak -1.

```
int SDL_CDPlay(SDL_CD *cdrom, int start, int length);
```

Mnohem častěji než SDL_CDPlay() se ale používá SDL_CDPlayTracks(), protože poskytuje snadnou volbu toho, co se má přehrávat.

```
int SDL_CDPlayTracks(SDL_CD *cdrom, int start_track,
                int start_frame, int ntracks, int nframes);
```

Start_track a ntracks označují první přehrávanou stopu a jejich celkový počet. Start_frame je offsetem ve framech od počáteční stopy a nframes offsetem od poslední. Pokud budou poslední dva parametry nulové, přehrává se až do konce CD. Datové oblasti jsou automaticky přeskakovány. Následující tři příklady byly přebrány ze SDL dokumentace.

```
// Přehraje se celé CD
if(CD_INDRIVE(SDL_CDStatus(cdrom)))
        SDL_CDPlayTracks(cdrom, 0, 0, 0, 0);

// Přehraje se poslední stopa
if(CD_INDRIVE(SDL_CDStatus(cdrom)))
        SDL_CDPlayTracks(cdrom, cdrom->numtracks-1, 0, 0, 0);

// Přehraje 15 sekund ze druhé stopy
if(CD_INDRIVE(SDL_CDStatus(cdrom)))
        SDL_CDPlayTracks(cdrom, 1, 0, 0, CD_FPS*15);
```

Následují nezbytné funkce každého přehrávání. Mimochodem, pokud se nezavolá SDL_CDStop(), CD se přehrává i po ukončení aplikace!

```
int SDL_CDPause(SDL_CD *cdrom); // Pozastavení
int SDL_CDResume(SDL_CD *cdrom);// Obnovení
int SDL_CDStop(SDL_CD *cdrom);  // Zastavení
```

Pomocí SDL_CDEject() lze disk vysunout z mechaniky.

```
int SDL_CDEject(SDL_CD *cdrom);
```

Ukázkové programy

CD přehrávač

Program implementuje velice jednoduchý CD přehrávač, který umožňuje přechody mezi stopami, posuny při přehrávání, pozastavení a také vysunutí mechaniky. Po spuštění programu s volbou -h nebo --help se zobrazí ovládání. Všechny informace se vypisují do konzole.

Vícevláknové programování

V dnešním díle o knihovně SDL se budeme věnovat podpoře tzv. vícevláknového programování. Podíváme se na vytváření nových vláken a samozřejmě také jejich synchronizaci, která nikdy nesmí chybět.

Jednovláknové a vícevláknové programy

Mít v programu více vláken může být velice výhodné. Jedno se stará o události, druhé o vykreslování a animace, třetí s pátým o cokoli jiného a všechno se to vykonává současně. Na stranu druhou si lze multithreadingem zadělat na tak obrovskou hromadu problémů, jaké si programátor "klasických" aplikací nedokáže ani představit.

Typickým příkladem jednovláknového programu je Hello, World. Na začátku se spustí main(), v ní se něco vypíše a pak se ukončí. V libovolném okamžiku běhu programu je možné zjistit, jaká instrukce právě proběhla a jaká bude následovat.

Spouštění vláken

Vícevláknový program také začíná funkcí main(), ale v určitém okamžiku se rozhodne, že by bylo vhodné spustit další vlákno. V případě SDL je tím okamžikem volání SDL_CreateThread(), které se předává ukazatel na libovolnou funkci, jejíž kód se bude v nově vytvořeném vláknu provádět. Vlákno se ukončí spolu s návratem z této funkce.

```
SDL_Thread *SDL_CreateThread(int (*fn)(void *), void *data);
```

Parametr typu void* je zde zvolen naprosto záměrně. Díky němu lze předat přes ukazatel data do funkce v podstatě cokoliv. Aby šlo pracovat s vlákny, je nutné inkludovat hlavičkový soubor SDL_thread.h, v němž se deklaruje vše potřebné.

Po průchodu funkcí SDL_CreateThread() se bude staré i nové vlákno vykonávat téměř současně. Slovo 'téměř' v tomto případě znamená, že na počítači s více procesory půjde (teoreticky) o paralelní běh. V případě jednoprocesorového systému se vlákna dynamicky přepínají, perioda je v jednotkách až desítkách milisekund, takže pro uživatele v podstatě neexistuje.

Pozn.: Teorie vícevláknového programování většinou pracuje také s tzv. procesy. Rozdíl mezi procesem a vláknem je ten, že jednotlivá vlákna sdílejí všechny systémové prostředky (paměť apod.), kdežto procesy jsou kompletně oddělené. Každý program uložený na disku se po svém spuštění stává procesem, může spouštěn další procesy a v nich vlákna. SDL vytváření procesů neumožňuje.

Hlavní vlákno by nikdy nemělo skončit dříve než všechna jím vytvořená vlákna. SDL pro tento účel poskytuje dvě funkce SDL_WaitThread() a SDL_KillThread(). První z nich čeká neomezenou dobu na ukončení a zároveň přebírá návratovou hodnotu, druhá funkce vlákno natvrdo zastaví.

Je-li to alespoň trochu možné, mělo by se vždy počkat na návrat z funkce spuštěné ve vláknu. Pokud například alokovalo dynamickou paměť, nemělo by ji šanci uvolnit.

```
void SDL_WaitThread(SDL_Thread *thread, int *status);
void SDL_KillThread(SDL_Thread *thread);
```

ID vlákna lze získat pomocí SDL_ThreadID() a SDL_GetThreadID(). První z funkcí uvažuje aktuální spuštěné vlákno a druhá libovolné předané.

```
Uint32 SDL_ThreadID(void);
Uint32 SDL_GetThreadID(SDL_Thread *thread);
```

V následujícím výpisu vytvoří funkce main() pracovní vlákno reprezentované funkcí vlakno(), obě se pak budou vykonávat paralelně. S našimi dosavadními znalostmi je vše naprogramováno správně, ale jak si ukážeme za chvíli, do kódu bude nutné ještě něco dopsat.

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

int vlakno(void *arg)
{
        // Nějaký kód
        for(int i = 0; i < 10000; i++)
                printf("vlakno()\n");

        return 0;
}

int main(int argc, char *argv[])
{
        // Inicializace SDL

        SDL_Thread *thread;
        if((thread = SDL_CreateThread(vlakno, NULL)) == NULL)
        {
                fprintf(stderr, "Nelze vytvořit vlákno: %s",
                                SDL_GetError());
                return 1;
        }

        // Nějaký kód
        for(int i = 0; i < 10000; i++)
                printf("main()\n");

        // Počká se na ukončení vlákna
        SDL_WaitThread(thread, NULL);
        return 0;
}
```

Výstup z programu jsem maličko upravil, ve skutečnosti stihne vlákno, před přepnutím do dalšího, zobrazit údajů mnohem více.

```
vlakno()
main()
main()
main()
vlakno()
vlakno()
main()
vlakno()
```

Synchronizace vláken

Do této chvíle bylo vše docela jednoduché, hlavním problémem u vláken je především jejich synchronizace a související integrita sdílených dat. Vrátíme-li se k předchozí ukázce, výstup programu bude ve skutečnosti vypadat spíše následovně.

```
main()
mvlakno()
vlakno()
vlakain()
```

```
mainno()
vlakno()
```

Nikde není řečeno, že se vlákna nemohou přepnout během vykonávání funkce printf(), takže se oba výpisy (ne)očekávaně slijí dohromady. Ve výsledku je text naprosto nečitelný. Vytvoření dalších vláken není nic složitého, těžká je spíše jejich synchronizace, aby nedocházelo k podobným stavům, jako v příkladu.

Jenom tak na okraj: nečekejte, že po přečtení tohoto článku se stanete expertem na paralelní systémy, k rozebrání tohoto tématu na alespoň trochu obstojné úrovni by možná nestačila ani několikasetstránková publikace. Jestli Vás mohu poprosit, berte tento článek spíše jako "populárně vědecké" seznámení...

V podstatě všechny nástroje pro synchronizaci jsou, velice zjednodušeně řečeno, jakési flagy řídící přístup do úseků programu, ve kterých může dojít k vzájemnému ovlivnění vláken. Už jsme se setkali se slitím výpisů v ukázkovém programu, typicky se jedná o přístup ke sdíleným (globálním) proměnným. Pamatujete-li si ještě na dvojici funkcí lock/unlock z grafiky popř. zvuků, musely se volat právě kvůli multithreadingu.

Pozn.: Výraz 'kritická sekce' není v textu používán záměrně. Ve Win32 API se jedná přímo o synchronizační prostředek, mohlo by se to plést.

Mutexy

Jeden z prostředků pro synchronizaci vláken představují tzv. mutexy, které jsou v SDL dostupné prostřednictvím struktury SDL_mutex. Ukazatel na nově vytvořený, odemknutý mutex je možné získat funkcí SDL_CreateMutex(), po skončení práce by se měl vždy pomocí SDL_DestroyMutex() uvolnit.

```
SDL_mutex *SDL_CreateMutex(void);
void SDL_DestroyMutex(SDL_mutex *mutex);
```

Pro zamknutí mutexu slouží funkce SDL_mutexP(), respektive její alias SDL_LockMutex(). Je-li už mutex zamknut jiným vláknem, vykonávání této funkce probíhá do té doby, než je mutex odemknut. Zamykání je navíc násobné, takže počet zamknutí musí odpovídat počtu odemknutí. V případě úspěchu vrátí funkce 0 a neúspěchu -1 (platí obecně u všech funkcí, dále už to nebude zmiňováno).

```
// Zamknutí
#define SDL_LockMutex(m) SDL_mutexP(m)
int SDL_mutexP(SDL_mutex *mutex);

// Odemknutí
#define SDL_UnlockMutex(m) SDL_mutexV(m)
int SDL_mutexV(SDL_mutex *mutex);
```

Uvedeme si jeden velice důležitý poznatek, který nemusí být na první pohled vidět. Pokud používáte dva různé mutexy (platí i pro ostatní synchronizační prostředky), měli byste si dávat pozor na pořadí jejich zamykání. Symbolicky naznačené pořadí

```
// VLÁKNO 1
lock(A);
lock(B);
// Přístup ke sdíleným prostředkům
unlock(B);
unlock(A);

// VLÁKNO 2
lock(B);
lock(A);
```

```
// Přístup ke sdíleným prostředkům
unlock(A);
unlock(B);
```

může způsobit tzv. deadlock projevující se kompletním zamrznutím programu. Po nakreslení příkazů jednotlivých vláken vedle sebe, by mělo být vše jasné.

```
VLÁKNO 1                VLÁKNO 2

...                     ...
lock(A);                ...
...                     ...
...                     lock(B);
lock(B);                ...
...                     lock(A);
...                     ...
wait(B);                wait(A);
wait(B);                wait(A);
```

Pozn.: Mutexy si lze zjednodušeně představit jako bool hodnotu s určitým, pevně stanoveným rozhraním. Obyčejné proměnné však pro synchronizaci vláken nelze používat, protože kompilátor nemusí přeložit ani obyčejné přiřazení jedinou (atomickou) instrukcí procesoru. Naproti tomu, rozhraní synchronizačních prostředků zaručuje, že během testu a následného zamknutí nemůže dojít k přepnutí vláken.

Následuje příklad na aplikaci mutexů včetně jejich vytváření a rušení.

```
// Globální proměnné
SDL_mutex *g_mutex;
int g_promenna;

// Vytvoření mutexu (inicializace)
g_mutex = SDL_CreateMutex();

// Zamknutí mutexu
if(SDL_mutexP(g_mutex) == -1)
{
        fprintf(stderr, "Nelze zamknout mutex\n");
        // Vhodná reakce
}

// Přístup ke sdíleným prostředkům
g_promenna = 173;

// Odemknutí mutexu
if(SDL_mutexV(g_mutex) == -1)
{
        fprintf(stderr, "Nelze zamknout mutex\n");
        // Vhodná reakce
}

// Zrušení mutexu (deinicializace)
SDL_DestroyMutex(g_mutex);
```

Z příkladu je vidět, že i obyčejné přiřazení do proměnné, ke které přistupují dvě odlišná vlákna, musí být ohlídáno

mutexem. Nechtějte se dostat do situace, kdy musíte odladit vícevláknový program, který z neznámého důvodu a pokaždé na jiném místě záhadně padá. Největším problémem je, že běh vícevláknové aplikace nelze nikdy identicky zopakovat, pozdější ladění probíhá na defakto úplně jiném programu.

Semafory

Další synchronizační proměnnou je semafor, v SDL je reprezentován strukturou SDL_sem. Semafory v sobě zahrnují číslo, které se při zamknutí atomicky dekrementuje a při odemknutí atomicky inkrementuje. Pokud je hodnota semaforu záporná, bude vlákno při zamykání automaticky zablokováno.

Semafor se vytváří funkcí SDL_CreateSemaphore() a ruší SDL_DestroySemaphore(). Jedním ze způsobů využití počáteční hodnoty je specifikace maximálního počtu vláken, které mohou vykonávat určitou činnost - aby například nedošlo k přetížení systému.

```
SDL_sem *SDL_CreateSemaphore(Uint32 initial_value);
void SDL_DestroySemaphore(SDL_sem *sem);
```

Funkce SDL_SemWait() pozastaví vlákno do té doby, než se hodnota semaforu dostane do kladných hodnot. Po průchodu funkcí je následně dekrementována. U druhé uvedené funkce, SDL_SemTryWait(), je činnost stejná, ale vlákno nebude nikdy zablokováno. Nečeká se na vpuštění, ale místo toho je ihned vrácena konstanta SDL_MUTEX_TIMEDOUT, podle které se programátor rozhoduje, co udělat dál.

Třetí varianta reprezentovaná SDL_SemWaitTimeout() je opět téměř stejná, jako předešlé. Na rozdíl od nich čeká na vpuštění pouze stanovený počet milisekund a poté opět vrací konstantu SDL_MUTEX_TIMEDOUT. Dokumentace uvádí, že je na některých platformách implementována cyklem, který každou milisekundu testuje hodnotu semaforu, což není zrovna efektivní.

```
int SDL_SemWait(SDL_sem *sem);
int SDL_SemTryWait(SDL_sem *sem);
int SDL_SemWaitTimeout(SDL_sem *sem, Uint32 timeout);
```

Po opuštění oblasti, ve které se přistupuje ke sdíleným prostředkům, by se měla zavolat funkce SDL_SemPost(). Dojde ke zvýšení hodnoty semaforu a případnému odblokování některého z čekajících vláken. Podobně, jako u mutexů, by se měla vždy volat ve dvojici s úspěšně provedenými wait funkcemi.

```
int SDL_SemPost(SDL_sem *sem);
```

Hodnotu semaforu lze kdykoli získat funkcí SDL_SemValue(). Nikdy by se však neměla používat pro rozhodnutí o přístupu ke sdíleným prostředkům, protože se nejedná o atomickou operaci.

```
Uint32 SDL_SemValue(SDL_sem *sem);
```

Jako příklad je uveden pokus o zamknutí pomocí SDL_SemTryWait(). Je-li semafor zamknut jiným vláknem, funkce se sice ukončí ihned, ale kód vlákno ke sdíleným prostředkům nepustí.

```
// Pokus o zamknutí
int res = SDL_SemTryWait(my_sem);

// Chyba
if(res == -1)
        return CHYBA_PRI_ZAMYKANI;

// Už zamknut jiným vláknem
if(res == SDL_MUTEX_TIMEDOUT)
        return SEMAFOR_ZAMKNUT;
```

```
/*
 * Operace se sdílenými prostředky
 */

// Odblokování
SDL_SemPost(my_sem);
```

Podmíněné proměnné

Podmíněné proměnné (anglicky condition variables) jsou reprezentovány strukturou SDL_cond a vytvářejí se funkcí SDL_CreateCond(). Pro jejich zrušení slouží SDL_DestroyCond(). Díky nim lze implementovat o něco komplexnější podmínky řídící vykonávání vláken.

```
SDL_cond *SDL_CreateCond(void);
void SDL_DestroyCond(SDL_cond *cond);
```

Zamykání je v tomto případě o něco složitější než u jiných synchronizačních prostředků. Funkce SDL_CondWait() přebírá v prvním parametru podmíněnou proměnnou a ve druhém libovolný mutex. Ten by měl být před vstupem do funkce zamknutý. Protože je po průchodu funkcí automaticky odemknut, je na programátorovi, aby ho opětovně uzamknul. Čekání druhé uvedené funkce je časově omezené, po vypršení intervalu vrací konstantu SDL_MUTEX_TIMEDOUT.

```
int SDL_CondWait(SDL_cond *cond, SDL_mutex *mut);
int SDL_CondWaitTimeout(SDL_cond *cond, SDL_mutex *mutex, Uint32 ms);
```

Voláním funkce SDL_CondSignal() se restartuje jedno z vláken, která čekají na na odemknutí podmíněné proměnné a v případě SDL_CondBroadcast() jsou restartovány všechna.

```
int SDL_CondSignal(SDL_cond *cond);
int SDL_CondBroadcast(SDL_cond *cond);
```

Literatura

Jak už jsem zmínil v průběhu textu, je tento článek spíše úvodem do problematiky vícevláknového programování. Pokud vás zaujala, v knihách uvedených níže, můžete najít mnohem podrobnější informace.

Mark Mitchell, Jeffrey Oldham, Alex Samuel: Pokročilé programování v operačním systému Linux (procesům a vláknům se věnují kapitoly 3, 4 a 5). Anglickou verzi této knihy lze také stáhnout z webu http://www.advancedlinuxprogramming.com/, je šířena pod licencí Open Publication License.

Jeffrey Richter: Windows pro pokročilé a experty. Jedná se o trochu starší knihu, která se sice věnuje ještě Win 95/NT, ale rozhodně stojí za to. Mimochodem, už ji asi nekoupíte, zkusil bych spíše nějakou knihovnu.

Ukázkové programy

Moc se omlouvám, ale v tomto dílu žádný ukázkový program nebude. Je to hlavně proto, že s multithreadingem nemám moc praktických zkušeností a netroufám si napsat žádný větší program, který by stál za to.

Druhým důvodem a v tuto chvíli o něco podstatnějším je, že momentálně nemám na disku žádný operační systém a článek dopisuji z live CD Slaxu, kde není ani gcc natož SDL. Původně jsem si myslel, že mi 'strýček' Debian spadl, ale po neúspěšných pokusech s novou instalací (Debian Stable, Ubuntu, Gentoo) odhalil memtest problémy s RAM. Ach jo... :-(

SDL_RWops, SDL_Overlay + vše, na co se zapomnělo

V dnešním, závěrečném, díle o knihovně SDL se pokusím shrnout všechny věci, na které jsem během psaní seriálu pozapomněl popř. kterým jsem se z důvodu mé neznalosti nevěnoval pozornost. Mimo jiné se budeme věnovat SDL_RWops, YUV video overlay, nahrávání sdílených knihoven za běhu aplikace a proměnným prostředí.

SDL_RWops

SDL_RWops je technika, kterou SDL poskytuje pro načítání obrázků a zvuků (obecně libovolných dat) z paměti namísto z diskových souborů.

Pokud umíte používat např. některou z knihoven pro komprimaci, díky SDL_RWops je možné importovat obrázky z archivu úplně stejně, jako by byly uloženy přímo na disku. Nebo, jste-li schopni napojit aplikaci k vysílání internetového rádia, naprogramování přehrávače bude otázkou pouhé chvíle. Fantazii se meze opravdu nekladou.

Základní funkcí pro vytvoření SDL_RWops je SDL_RWFromFile(), která slouží pro práci s klasickými soubory. První parametr specifikuje diskovou cestu a druhý označuje mód otevření analogický parametru standardní funkce fopen() - "r" pro čtení, "w" pro zápis, atd.

```
SDL_RWops *SDL_RWFromFile(const char *file,
                          const char *mode);
```

Funkce SDL_RWFromFP() je analogií SDL_RWFromFile(), v prvním parametru přebírá namísto řetězce se jménem deskriptor otevřeného souboru. Pokud nebude druhý parametr nulový, SDL soubor po skončení práce automaticky uzavře.

```
SDL_RWops *SDL_RWFromFP(FILE *fp, int autoclose);
```

Pozn.: Dokumentace uvádí, že SDL_RWFromFP() není pod Win32 dostupná. Na této platformě údajně nemohou být soubory otevřené aplikací použity dynamicky linkovanou knihovnou.

Jádrem SDL_RWops jsou funkce SDL_RWFromMem() a SDL_RWFromConstMem(), které vytvářejí SDL_RWops z dat uložených v paměti, resp. v konstantní paměti. Předává se jim ukazatel na tuto paměť a její velikost.

```
SDL_RWops *SDL_RWFromMem(void *mem, int size);
SDL_RWops *SDL_RWFromConstMem(const void *mem, int size);
```

Funkce SDL_AllocRW() alokuje paměť pro prázdnou SDL_RWops strukturu a SDL_FreeRW() slouží pro její uvolnění. Používají se téměř výhradně při vytváření SDL_RWops z nějakého nestandardního zdroje. Všechna vnitřní data vráceného objektu se musí inicializovat manuálně, příklad je možné najít v dnešním ukázkovém programu, pracuje se v něm se ZIP archivem.

```
SDL_RWops *SDL_AllocRW(void);
void SDL_FreeRW(SDL_RWops *context);
```

Struktura SDL_RWops obsahuje ve svém nitru atribut rozlišující typ obsahu a union ukládající data. Verze stdio slouží pro souborové SDL_RWops a mem pro paměťové. S poslední položkou, unknown, by se mělo operovat při uživatelské alokaci pomocí výše zmíněné funkce SDL_AllocRW().

```
typedef struct SDL_RWops
{
        Uint32 type;
```

```
        union
        {
                struct
                {
                        int autoclose;
                        FILE *fp;
                } stdio;
                struct
                {
                        Uint8 *base;
                        Uint8 *here;
                        Uint8 *stop;
                } mem;
                struct
                {
                        void *data1;
                } unknown;
        } hidden;

        int (*read)(struct SDL_RWops *context, void *ptr, int size, int maxnum);
        int (*write)(struct SDL_RWops *context, const void *ptr, int size, int
num);
        int (*seek)(struct SDL_RWops *context, int offset, int whence);
        int (*close)(struct SDL_RWops *context);
} SDL_RWops;
```

Poslední čtyři položky struktury jsou ukazatele na funkce, které poskytují přesuny na jiná místa v paměti, čtení, zápis a uvolnění dat. Nemusí se volat přímo, lze použít makra níže.

```
#define SDL_RWread(ctx, ptr, size, n)   (ctx)->read(ctx, ptr, size, n)
#define SDL_RWwrite(ctx, ptr, size, n)  (ctx)->write(ctx, ptr, size, n)
#define SDL_RWseek(ctx, offset, whence) (ctx)->seek(ctx, offset, whence)
#define SDL_RWtell(ctx)                 (ctx)->seek(ctx, 0, SEEK_CUR)
#define SDL_RWclose(ctx)                (ctx)->close(ctx)
```

Makra se chovají v podstatě stejně, jako standardní funkce ze stdio. Parametr ctx je ukazatel na SDL_RWops, ptr adresa bufferu, z/do kterého se čte/zapisuje, size počet bytů v bloku a n počet načítaných/zapisovaných bloků. Návratovou hodnotou je počet načtených/zapsaných bloků dat nebo -1 při chybě. Parametr whence ze SDL_RWseek() může nabývat konstant SEEK_SET, SEEK_CUR, SEEK_END.

Jenom pro pořádek: poslední z maker, SDL_RWclose(), by mělo být zavoláno po skončení práce s libovolným SDL_RWops. Jedinou výjimkou jsou taková SDL_RWops, u kterých bylo požádáno o automatické uzavření.

Otevřete-li si některý z hlavičkových souborů SDL, zjistíte, že v podstatě všechny funkce pracující se soubory představují pouze aliasy na načítání ze SDL_RWops. To samé platí pro rozšiřující knihovny, jako jsou SDL_image, SDL_sound, SDL_ttf a další. Například SDL_LoadBMP() je pouze souborová specializace SDL_LoadBMP_RW().

```
SDL_Surface *SDL_LoadBMP_RW(SDL_RWops *src, int freesrc);

#define SDL_LoadBMP(file) \
        SDL_LoadBMP_RW(SDL_RWFromFile(file, "rb"), 1)
```

Vypisovat seznam všech těchto funkcí je v podstatě zbytečné. Většinou by mělo stačit přidat ke jménu příponu '_RW' a

místo řetězce se jménem předat ukazatel na SDL_RWops. Pokud nebude tato technika úspěšná, v některém z hlavičkových souborů lze vždy najít přesnou deklaraci.

Než se se SDL_RWops úplně rozloučíme, nelze neuvést odkaz na tento výborný tutoriál (anglicky).

YUV video overlay

YUV video overlay je grafická struktura, která poskytuje hardwaru přímý přístup do paměti obrázku. Zjednodušeně řečeno, místo, aby se při zobrazování všechny pixely zdlouhavě kopírovaly na určité místo na grafické kartě, program pouze oznámí jejich adresu v paměti a o nic dalšího se nestará. Mnohem vyšší rychlost předurčuje použití u přehrávání videa, jak je patrné už z názvu.

```
typedef struct
{
        Uint32 format;          // Formát
        int w, h;               // Rozměry
        int planes;             // Počet rovin (obyčejně 1 nebo 3)
        Uint16 *pitches;        // Pole pitch
        Uint8 **pixels;         // Pole ukazatelů na data pro každou rovinu
        Uint32 hw_overlay:1;    // Hardwarově akcelerovaný?
} SDL_Overlay;
```

Kromě pixelů jsou všechny položky pouze pro čtení, k těm se ale může přistupovat až po zamknutí struktury. Atribut format může nabývat následujících hodnot, více informací lze najít na této stránce.

```
#define SDL_YV12_OVERLAY 0x32315659 // Planar mode: Y + V + U
#define SDL_IYUV_OVERLAY 0x56555949 // Planar mode: Y + U + V
#define SDL_YUY2_OVERLAY 0x32595559 // Packed mode: Y0+U0+Y1+V0
#define SDL_UYVY_OVERLAY 0x59565955 // Packed mode: U0+Y0+V0+Y1
#define SDL_YVYU_OVERLAY 0x55595659 // Packed mode: Y0+V0+Y1+U0
```

Overlay se vytváří funkcí SDL_CreateYUVOverlay(). Její parametry definují rozměry, formát a surface, na kterém bude zobrazen. Vzhledem k tomu, že je overlay vytvořen v hardwaru, bude při zobrazení oblast surfacu pod ním přepsána a její obsah není definován. Pro následné uvolnění slouží SDL_FreeYUVOverlay().

```
SDL_Overlay *SDL_CreateYUVOverlay(int width, int height,
                Uint32 format, SDL_Surface *display);

void SDL_FreeYUVOverlay(SDL_Overlay *overlay);
```

Při přímém přístupu k pixelům je vždy nutné overlay uzamknout.

```
int SDL_LockYUVOverlay(SDL_Overlay *overlay);
void SDL_UnlockYUVOverlay(SDL_Overlay *overlay);
```

Overlay se zobrazuje funkcí SDL_DisplayYUVOverlay(), pozice a velikost cílové oblasti se specifikuje obdélníkem dstrect. Pokud bude mít overlay jinou velikost než cílová oblast, bude automaticky roztáhnut (max. 2x). Funkce vrací v případě úspěchu nulu.

```
int SDL_DisplayYUVOverlay(SDL_Overlay *overlay,
                        SDL_Rect *dstrect);
```

Pozn.: Z mého výkladu bylo asi poznat, že toho o overlay-ích moc nevím :-(. Něco málo informací, včetně několika odkazů, lze najít v diskuzi k osmému dílu, kde se toto téma probíralo.

Little/big endian

Hlavičkový soubor SDL_endian.h deklaruje funkce pro práci s daty ve formátech little a big endian, tyto dva pojmy se vztahují k pořadí jednotlivých bytů ve vícebajtových proměnných. Na některých platformách se ukládají důležitější byty na nižší adresy a na jiných je tomu právě naopak. Vzhledem k tomu, že je SDL multiplatformní, a tedy dostupné na obou typech systémů, je přítomnost těchto funkcí naprosto zásadní.

Aby mohla aplikace jednoduše zjistit, na kterém typu systému běží, poskytuje SDL symbolickou konstantu SDL_BYTEORDER, která může být nastavena buď na SDL_LIL_ENDIAN nebo na SDL_BIG_ENDIAN.

Používáte-li pro načítání obrázků, zvuků a ostatních dat standardní SDL funkce, nemusíte se teoreticky o podobné záležitosti vůbec starat. Problémy však mohou nastat, pokud si píšete vlastní loadery. API je relativně jednoduché, a proto odkazuji zájemce, vzhledem k místu, na výše zmíněný hlavičkový soubor.

Proměnné prostředí

SDL poskytuje dvojici funkcí SDL_putenv() a SDL_getenv(), které umožňují zápis a čtení hodnot do/z proměnných prostředí. Při zápisu se předává řetězec ve formátu "jméno=hodnota", čtení by mělo být jasné.

```
int SDL_putenv(const char *variable);
#define putenv(X) SDL_putenv(X)

char *SDL_getenv(const char *name);
#define getenv(X) SDL_getenv(X)
```

V shellu je možné definovat proměnné určitých názvů, kterými lze změnit standardní chování SDL. V tomto seriálu jsme se už setkali se SDL_VIDEODRIVER a SDL_AUDIODRIVER specifikující video a audio ovladače, je jich však mnohem více. Podrobný seznam je možné najít v první sekci SDL dokumentace pod pojmem SDL_envvars.

Dynamické knihovny

Většinou se služby z externích knihoven poskytují aplikaci při překladu, v SDL je však možné zpřístupňovat knihovny i za běhu programu. Dynamická knihovna se nahrává funkcí SDL_LoadObject(), v jediném parametru se jí předává řetězec se jménem a cestou. Pro uvolnění slouží funkce SDL_UnloadObject().

```
void *SDL_LoadObject(const char *sofile);
void SDL_UnloadObject(void *handle);
```

Ukazatel na funkci nacházející se ve sdílené knihovně je možné získat pomocí SDL_LoadFunction(). Parametry definují handle knihovny, ve které se má hledat, a řetězec se jménem funkce. Knihovna musí zůstat zavedená do paměti po celou dobu používání, pointer by přestal být validní.

```
void *SDL_LoadFunction(void *handle, const char *name);
```

Informace o procesoru

A ještě bonus na závěr: Hlavičkový soubor SDL_cpuinfo.h obsahuje několik funkcí, kterými lze zjistit vlastnosti procesoru v počítači. Co která dělá si jistě domyslíte sami.

```
SDL_bool SDL_HasRDTSC();
SDL_bool SDL_HasMMX();
SDL_bool SDL_HasMMXExt();
SDL_bool SDL_Has3DNow();
SDL_bool SDL_Has3DNowExt();
SDL_bool SDL_HasSSE();
```

```
SDL_bool SDL_HasSSE2();
SDL_bool SDL_HasAltiVec();
```

Ukázkové programy

Obrázky ze ZIP archivu

Program je modifikací ukázkového příkladu ze 13. dílu, obrázky se teď načítají pomocí SDL_RWops ze ZIP archivu, jinak žádná větší změna. Aby šel program zkompilovat, musí být v systému nainstalovaná knihovna zziplib. Je šířena pod licencí GNU LGPL a pracuje pod několika operačními systémy včetně GNU/Linuxu a MS Windows.



Pokračování

Jak jsem zmínil na začátku, toto je poslední díl našeho seriálu o knihovně SDL. Popravdě zbyly ještě dvě témata, která jsem chtěl původně zařadit, ale už se jim věnovat nebudu.

Prvním z nich je rozšiřující knihovna SDL_net pro implementaci síťových her. Bohužel jediné, co o ní v současné době vím, je to, že existuje - na komplexní článek docela málo.

Druhým tématem měla být tvorba GUI. Pro SDL existuje hned několik knihoven na tvorbu tlačítek, editboxů a podobných věcí, většinu z nich lze najít v menu libraries na libsdl.org. Další možností by mohlo být napojení SDL aplikace na GTK nebo QT, popř. minulý týden jsem objevil rychle se rozvíjející C++ knihovnu Guichan podporující SDL, Allegro a OpenGL (dohromady nebo zvlášť). I toto rozsáhlé téma ale nechávám na samostudium.