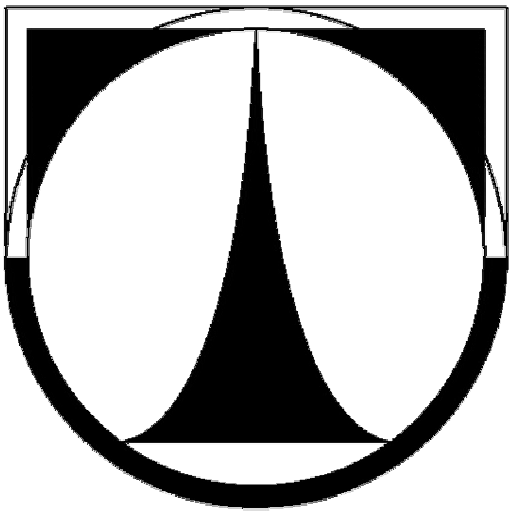
TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta strojní

Katedra aplikované kybernetiky



BAKALÁŘSKÁ PRÁCE

V Liberci 2010

Dan Kryštof

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta strojní

Studijní program B2341 - Strojní inženýrství

Zaměření Inženýrská informatika

Katedra aplikované kybernetiky

APLIKACE PRO ZTVÁRNĚNÍ PROSTOROVÉ SCÉNY S VYUŽITÍM KNIHOVNY OPENGL

Autor: Dan Kryštof
Vedoucí BP: Ing. Michal Moučka, Ph.D.

Rozsah práce a příloh:

Počet stran : 47

Počet obrázků : 19

Počet tabulek : 2

Počet příloh : 4

V Liberci leden 2010

ORIGINALNI ZADANI

ANOTACE

Předmětem této bakalářské práce bylo seznámení se s problematikou vytvoření grafické aplikace, schopné vykreslit dynamicky se měnící 3D scény v reálném čase.
Pro tyto účely bylo nutné nejdříve poznat rozhraní OpenGL , na kterém bude aplikace postavena.
První část práce se tak pokouší vzájemně srovnat a ohodnotit výhody a nevýhody dvou dnes nejpoužívanějších API rozhraní OpenGL a DirectX.
Další část se již zabývá pouze OpenGL, vysvětluje základní principy, datové typy a funkce. Tyto informace jsou pak následně použity jako podklad pro další vývoj.

Výsledkem programové části je pak realizována aplikace napsaná v jazyce C++ a využívající funkce OpenGL.

KLÍČOVÁ SLOVA: API, OpenGL, DirectX, GDI, WIN32, GLUT, jazyk C++, vytváření vláken

ANNOTATION

The subject of this bachelor's thesis was to do familiar with the issue of creating graphic application , able to render dynamically changing 3D scene in real-time. For these purposes was necessary to firstly know the interface OpenGL, on which application will be built.
The first part of this work is trying to compare and evaluate advantages and disadvantages of two contemporarymost popular API interfaces OpenGL and DirectX.
The next part deals only with OpenGL, explains the basic principles, data types and functions. This information is then used as a basis for further development.

The result of the program is then implemented application written in C++ and using OpenGL functions.

KEY WORDS: API, OpenGL, DirectX, GDI, WIN32, GLUT, language C++, creating threads

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že Technická univerzita v Liberci (TUL) má právo na uzavření licenční smlouvy o užití mé BP a prohlašují, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití,jsem si vědom povinnosti informovat o této skutečnosti TUL;v tomto případě má TUL právo ode mne požadovat úhradu nákladů,které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Dne:5.1.2010 v Liberci .....

Dan Kryštof

Poděkování

Na tomto místě bych rád poděkoval Ing. Michalu Moučkovi, Ph.D. za odbornou pomoc a vedení, za cenné připomínky a odborné rady, kterými přispěl k vypracování této bakalářské práce.

Dále děkuji své rodině za všestrannou podporu poskytovanou při studiu i při psaní této bakalářské práce.

OBSAH:

1 Předmluva

3D grafika a různé 3D digitální technologie se v dnešním světě staly, ať už ve větší či menší míře, součástí mnoha výrobních procesů. Počínaje hojným využíváním v zábavním průmyslu, přes televizní reklamní znělky, konče konstrukční výpočty a simulace používané v architektonice či strojírenském průmyslu.

Díky moderní počítačové animaci je tak možné ve filmu nebo počítačové hře navštívit úžasné fantastické světy, kdy je mnohdy velmi těžké rozpoznat, co je ještě skutečné, a co je již počítačová grafika. Počítačové simulace však neslouží jen pro zábavu, ale i ku prospěchu člověka. Architekti se tak například mohou projít po budově, která se ani nezačala stavět, v laboratořích si vědci mohou nasimulovat procesy, které by byly za normálních okolností nebezpečné nebo přinejmenším velmi drahé. To, co bylo dříve považováno za oblast science fiction dnes nabývá velmi reálných rozměrů.

Toto masivnější rozšíření počítačové grafiky, by však nikdy nebylo možné bez velkého pokroku v oblasti elektroniky a počítačů, který nyní zažíváme. Dnešní grafický hardware dosáhl takové úrovně, že je schopen během jedné sekundy vykreslovat stovky milionů polygonů a provádět miliony operací za vteřinu. Dalším výrazným kritériem se stává poměrně snadná dostupnost velmi kvalitního HW vybavení i pro laickou veřejnost.

Aby však bylo možné využít veškerý potenciál, který současný hardware poskytuje, je nejprve potřeba řada programátorů, kteří vytvoří softwarové rozhraní, díky kterému již neuvidíme pouze jedničky a nuly, se kterými počítač a grafická karta pracuje, ale získáme dokonalý obraz na monitor.Většina výrobců grafických karet proto vydává nemalé výdaje nejen na vývoj budoucího hardwaru, ale i na vývoj softwarového rozhraní. O tom, které z nich bude nejlépe vyhovovat zadání, již pojednávají následující kapitoly.

1.1 Cíle práce

Tato práce je rozdělena na dvě větší části, teoretickou a praktickou.
Teoretická část má za úkol seznámit čtenáře s dostupnými metodami používanými pro vykreslování 3D grafiky. Na začátku jsou je nejprve stanoveny cíle, které by konečná aplikace měla splňovat, a jejich možný způsob řešení. Po této prvotní analýze se práce důkladně zaměřuje na popis rozhraní OpenGL a DirectX. Pokouší se čtenáři objasnit základní rozdíly mezi nimi, shrnout jejich hlavní odlišnosti a nastínit důvody, proč bylo zvoleno právě OpenGL pro konečnou aplikaci.

Druhá polovina je pak věnována praktickému popisu aplikace založené na použití jádra OpenGL, způsob návrhu jednotlivých součástí a popis mého uvažovaní při řešení jednotlivých problémů. Konečným cílem této části by tak měla být úplná představa o tom, jak jednotlivé části pracují, jak spolu kooperují a dále jak aplikaci používat.
V závěru je pak vyhodnocen celkový vývoj aplikace, zda byly či nebyly splněny jednotlivé cíle stanovené na začátku.

Co se týká praktické aplikace, mělo by se jednat o grafický terminál, schopný dynamicky reagovat na změnu obsahu dat vykreslované scény. Uživatel pak musí mít absolutní volnost v rozsahu pohybu po okolí. Data by měla aplikace přebírat z externího zdroje, aby při jejich změně nemohlo nedopatřením dojít k zásahu do programového kódu.

Většinu textu použitých pro zpracování mé práce jsem čerpal z odborných online článků, neboť jsou pravidelně aktualizovány. Navíc na paralelních fórech k těmto článkům je možná i komunikace se samotným autorem, který může poskytnou i některé doplňující informace. Přehled všech použitých online zdrojů je uveden v seznamu literatury.

1.2 Motivace

Trojrozměrná počítačová grafika je téma mně velmi blízké, neboť mám již bohaté zkušenosti s 3D modelovacím nástrojem Cinema4D firmy Maxon.

Proto jsem uvítal možnost napsat svou bakalářskou práci právě na toto téma.Vždy jsem se chtěl podívat na takovou aplikaci i z jiného úhlu, než jen jako pouhý uživatel, zajímalo mě jak funguje takové vykreslovaní, poznat jaký je rozdíl při vykreslování pod OpenGL nebo DirectX atd. Nakonec zjistit, kolik hodin úsilí a znalostí je třeba vynaložit k vývoji vlastního softwaru, to jsou zkušenosti, které se dají v budoucnu jistě uplatnit.

1.3 Použití vizualizačních programů v technické praxi

Již dávno se i strojírenství při konstrukci a vývoji přesunulo ze světa 2D do třetí dimenze, neboť zobrazování 3D grafickou podobou je pro uživatele mnohem příjemnější než „pouhé“ výkresy.

Předními konstrukčními softwary v této oblasti jsou AutoCAD a 3DS MAX od firmy Autodesk, ProEngineer od PTC nebo Catia, diky kterým lze velmi jednoduše vytvořit velmi reálné koncepty budoucího výrobku, které pak jsou nepostradatelným pomocníkem při odvozování detailního návrhu výrobku a jeho procesu výroby a montáže.

Z uživatelského hlediska jsou si pak všechny tyto aplikace velmi podobné. Po spuštění aplikace se v centrální částí aplikace nachází hlavní okno, ve kterém je perspektivní náhled na konstruovaný výrobek, kolem dokola jsou lišty obsahující více či méně důležité nástroje potřebné pro modelování. Některé ze jmenovaných softwarů disponují i tak propracovanými nástroji, že jsou schopny např. simulovat podmínky, ve kterých se bude vyrobený díl nacházet. Mohou tak pomoci najít a zoptimalizovat případná nebezpečná místa bez potřeby vlastní fyzické výroby.

Co mají ale tyto aplikace po většinou společné, je rozhraní, díky kterému přistupují ke grafickému adaptéru, nebývá ani zvláštností, že je možné si vybrat hned ze dvou– jak OpenGL tak i DirectX. Preferovaný vykreslovací rozhraní přímo ovlivňuje výkon i kvalitu, a proto je důležité vybrat si to vhodnější pro ten konkrétní systém a hardware.

Technické aplikace se tak povětšinou využívají k tvorbě nových grafických návrhů součástí, to již je patrno z předešlého textu. Dále však jde počítačovou konstrukci velmi dobře využít i na úpravu již vyrobených součástí. Ve strojírenství se při běžné výrobě nejdříve vytvoří CAD výkres a teprve z něj se pak vyrobí konečná součást. Pokud se ale použije tzv. *zpětné inženýrství* (z anglického Reverse Engineerin) je postup práce úplně opačný, máme k dispozici hotovou součást a chceme k ní vytvořit CAD model. Zásadním předpokladem pro použití této techniky je získání prostorových souřadnic součástky, nasnímaných z reálného světa. Abychom toho byli schopni, používají se všemožné 3D skenery, které ze snímaného povrchu vytvoří mrak bodů, tyto body se potáhnou velmi jednoduchou polygonovou sítí, ta se následně ještě vyhladí a převede do CAD formátu.

Různé obdoby zpětného inženýrství se používají v mnoha odvětvích (například zkoumání práce různých počítačových programů, rekonstrukce zničeného dílu apod.).
Nejčastěji se však tak tato technika využívá při vytváření designových studií prototypů a konceptů. Tento způsob výroby počítá ve vytvoření hliněného nebo dřevěného modelu, který je pro pozdější výrobní proces nutné převést zpět do digitální podoby. Jako dobrým příkladem použití tohoto způsobu výroby jsou dnes moderní automobilky, kde se nejdříve vyrobí prototyp jako hliněný model, který se pak zpátky převádí do PC pro následné zpracovaní a testy.

2 Analýza

Důležitou fází vývoje aplikace je z poskytnutých informací ze zadání vybrat ty nejpodstatnější vlastnosti, které je potřeba splnit. Kromě ryze technických parametrů je potřebné myslet především na budoucí použití aplikace, neboť tu si programátor nevyvíjí sám pro sebe, nýbrž pro koncového uživatele. Musí se zvážit, jaké služby by mu měla aplikace poskytovat, zvolit co nejjednodušší ovládání a zajistit možnost snadného pozdějšího rozšíření. Z této analýzy již pak nebude těžké navrhnout tu nejvhodnější variantu řešení z hlediska požadavků zadání.

a) Programovací jazyk

Při volbě programovacího jazyku je potřeba zjistit, co jsou konkrétní jazyky schopny provést, zda například zvládají práci s třídami, jaké datové typy používají, které funkce podporují atd. Pro vývoj 3D grafických aplikací a knihoven se nejčastěji používají jazyky:

- Java3D**,**
- jazyk C#**,**
- jazyk C a C++**,**
- Pascal atd.

Pro vývoj finální aplikace byl zvolen programovací jazyk C++, jelikož je nejpoužívanější pro vývoj 3D grafických aplikací, a většina dostupných knihoven je napsána právě v tomto jazyce. Dále je tento jazyk schopen zpětně podporovat i kódy napsané v jazyce C, což je další velmi užitečná vlastnost.

Co se týká vývojového prostředí, je použito prostředí code::block verze 8.02. Důvodem volby právě tohoto prostředí je jeho freeware licence a velké množství dokumentace na stránkách autora a také to, že již základní verze obsahuje téměř všechny knihovny potřebné k práci se základním jádrem OpenGL.

Jazyk C++

Protože jsem se s tímto jazykem doposud nesetkal, bylo potřeba seznámit se s jeho vlastnostmi a použitím zcela od začátku.

Jazyk C++ [1] je nadmnožina jazyka C. Vznik jazyka C se přisuzuje společnosti Bell Laboraties 1972, jmenovitě programátoru Dennisu Ritchie. Jelikož měl při práci na svých projektech negativní zkušenosti s tehdy používanými programovacími jazyky, rozhodl se vytvořit si jazyk vlastní. Do něj chtěl zakomponovat rychlost nízkoúrovňového jazyka s hardwarovým přístupem vyššího programovacího jazyka a tím zajistil patřičnou přenositelnost. Na základě zkušeností ze starších programovacích jazyků tak vytvořil první verzi jazyka C.

Další vývoj se zaměřil na vydaní normy, která by sjednotila a definovala veškeré standardy jazyka C. V roce 1984 tak vychází první norma ANSI C.

Samotný Jazyk C++ se pak zrodil, stejně jako C, v Bell Labs, jeho autorem však nebyl Dennis Ritchie, nýbrž Bjarne Stroustrup. Ten se soustředil na vývoj a implementaci jazyka, který by byl stejně výkonný a přenositelný jako C, navíc by byl schopen vytvářet knihovny a obsahoval podporu pro objektové programování. To vyústilo v roce 1980 vydáním jazyka, který byl nazván C with Classes, kde již z názvu jde poznat že se jednalo o „nadstavbu“ C s podporou tříd. Programový kód pak byl kompilován speciálním překladačem Cpre zpět do formy C. Tento přístup byl zvolen pro tehdejší dostupnost jazyka C.

V červenci 1983 se jazyk přejmenoval na C++ a s ním i překladač na Cfront. Toto označení, pod kterým ho známe dodnes vychází z operátoru přírůstku používaného v C, který přičítá k hodnotě proměnné 1. Tento název přesně odkazuje na to, že se jedná o rozšířenou verzi C.

Jazyk C++ se tak stal jedním z nejdůležitějších a nejoblíbenějších jazyků 80 let a slibuje, že jeho vývoj bude výrazně pokračovat i v 21 století.

b) Používané platformy

Volba cílové platformy (ať už hardwaru nebo operačního systému), pro kterou je konečné rozhraní určeno, je rozhodně velmi důležitá položka, neboť co je u jednoho systému považováno za samozřejmost u druhého může být nepřekonatelný problém a naopak.

Proto se zde nabízí otázka: bude se aplikace vyvíjet primárně pouze pro jeden operační systém a platformu, nebo ponecháme určitou přenositelnost mezi nimi? Volba jazyka je také velmi úzce spjata s volbou platformy. Musí se brát v potaz, že ne každý jazyk bude 100% kompatibilní s danou platformou (systémem) a je třeba s tím počítat před začátkem vývoje.

Primárně tak bude aplikace vyvíjena pro Windows, jelikož se jedná o nejrozšířenější operační systém. Při implementaci aplikace se ale bude částečně počítat i s pozdější možností přenositelnosti na jiné platformy (např. LINUX),a to díky použití knihovny GLUT.

c) Způsob zobrazování vstupních dat

Jelikož mnohé moderní aplikace jsou přímo vyvíjeny a uzpůsobeny pro speciální oblast využití, máme možnost si zvolit i způsob zobrazovaní výstupních dat. Určitě totiž nebudou mít stejné způsoby vykreslovaní dat softwary vyvíjené pro automobilové designéry, kteří si mohou dovolit renderovat jeden dokonalý obrázek automobilu i několik hodin, jiné nároky zas budou mít aplikace simulující různé fyzikální procesy, kde je zapotřebí zobrazování dat v reálném čase.
Na výsledné zobrazení tak lze nahlížet ze dvou stran:

- Aplikace bude schopna vykreslovat naprosto fantastické fotografie tvořené z milionů polygonů, které budou prakticky k nerozeznání od skutečnosti. Za tuto dokonalost, ale budeme muset zaplatit formou neúměrně dlouhého výpočtu. Takový přístup je nazývaný *statické zobrazovaní* dat.

- Druhou možností je vykreslovaní v reálném čase neboli *real-time* grafika.
  Povrchy těles jsou většinou vykreslovány z menšího počtu elementů než u statických obrázků, proto se objekty mohou jevit jako hranaté, ale zato získáme nesrovnatelně rychlejší výpočet jednotlivých obrazů (pro získání plynulého obrazu je potřeba minimálně 15 obrazů za sekundu, ideálně pak 25 obrazů za s).

Obě z těchto technik nabízí úplně jiný přístup k pojetí vykreslování a každá se hodí pro jinou oblast použití. Ze zadání práce vyplívá, že se vývoj zaměří na tvorbu aplikace, která bude zobrazovat data v reálném čase.

d) Formát vstupních dat

Vykreslované modely mohou být vytvořeny na počítači člověkem pomocí 3D grafického modelovacího studia, podle dat získaných měřicím přístrojem z reálného světa nebo na základě počítačové simulace. Data všech modelů, ať už vytvořené jakoukoliv způsobem, je potřeba roztřídit, zpracovat a uložit do vhodného formátu. Dnes existuje mnoho různých datových typů, do kterých lze prostorová data ukládat, každý je postavený na odlišných pravidlech a jiném způsobu zápisu. Je tak nasnadě zjistit, jak snadno lze uložená data transformovat zpět na prostorové souřadnice.
Nejpoužívanější 3D grafické softwary dnes pracují s některým z těchto grafických formátů:

.3ds (software Autodesk 3DS MAX),
.dwg (použitý u většiny aplikací AutoCAD),
.dxf (speciální formát pro export mezi CAD aplikacemi),
.obj (wavefront technologie-formát schopný ukládat i animovaná data podporovaný většinou dostupných aplikací),
.x (formát DirectX).

Po několika předběžných testech však nevyhovoval ani jeden z popsaných formátů. Pro řešení tohoto problému bylo nutné navrhnout úplně nový datový typ, 100% optimalizovaný pro potřeby aplikace.Jako zcela dostačujícím datovým formátem se ukázal být textový soubor, neboť nebude pevně zakomponován do programového kódu, což bylo jedním z požadavků zadání, z něj lze velmi snadno číst a editovat ho.

.

e) Cena a dostupnost knihoven

Jedním z důležitých požadavků při vývoji je, předběžně si stanovit cílovou finanční částku, kterou je možné do projektu investovat. Cena vývoje a produktu je pak jedním z nejobjektivnějším kritériem při konečném výběru koncového uživatele.

Bohužel ani v tomto bodě to není s posuzováním užitečnosti jednotlivých rozhraní a knihoven tak jednoznačné, neboť lze nalézt i velmi kvalitní grafické knihovny, které jsou poskytovány autorem zcela zdarma. Pro co nejsnazší možnost šíření je proto aplikace tvořena pouze z volně šiřitelných knihoven, aby nemohlo dojít k pozdějším problémům se zakupováním licence apod.

2.1 Dostupná rozhraní

V této kapitole bude důkladně rozebráno téma API, co to vlastně je, jak obecně pracují a nakonec se zodpoví otázka volby programového rozhraní pro tvorbu konečné aplikace.

API (anglicky Application Programming Interface) je často spojováno pouze s procesy prohýbajícími v počítači. API ale není vždy vytvářeno primárně pouze pro PC, jedná se hlavně o prostředek komunikace mezi dvěma (a více) subjekty, např. mezi tvůrcem aplikace a jejím uživatelem, kde se uživatel ani nemusí přímo zajímat o programový kód. Konečná aplikace pro něj může být jakási black box, on pouze potřebuje komunikovat s aplikací pomocí nějakého grafické výstupu, který pro něj bude snadno srozumitelný a pravě to poskytuje API.

Pro vytvoření grafického prostředí pak PC neobsahuje pouze jedno, ale hned několik typů API každé pro jiné účely a jiný hardware. Operační systém má své API, Windows má WIN API respektive WIN32 (WIN64) a všechny programy vytvářené ve Windows musí nezávisle na použitém programovacím jazyce komunikovat prostřednictvím něj. WIN 32 tak obsahuje nejen základní funkce, ale i funkce pro vytváření uživatelského rozhraní, což bude v pozdější fázi velmi důležité.

Pro další text je ale nejpodstatnější API pro grafické karty, to je tvořeno sadou funkcí, které poskytuje systém, nebo samotná karta. Rozhraní pro všechny grafické adaptéry jsou standardizované, aby vývojář nemusel řešit, na kterém adaptéru se scéna bude zobrazovat a mohl se zabývat pouze scénou samotnou.Vývoj dospěl dokonce do takové fáze, že většina výrobců HW přizpůsobuje konstrukci svých karet přímo budoucímu použiti jednotlivým API.

Pokud má karta GPU (Graphics Processing Unit), musí mít i nějakou instrukční sadu.
GPU na grafické kartě je obdoba hlavního procesoru CPU v PC. I když z historického hlediska jsou CPU a GPU postaveny na stejném principu, postupem času se konstrukčně i výkonově značně rozešly.

GPU oproti CPU je výkonnější v oblasti paralelního zpracovávaní úloh. Grafické karty jsou totiž konstruovány pro zpracování co nejvíce dat v co nejmenším čase– obraz na monitoru je tvořen z milionů bodů které musí být vypočítány a zobrazeny současně. Oproti tomu se od CPU žádá maximální výkon, nehledě na typ a množství zpracovávaných úloh.

Z konstrukčního hlediska tak CPU sice mají mnohem méně jader (při psaní této práce jsou maxima 4 fyzická jádra s 8 procesory), pracují však na vyšší frekvenci, tím pádem disponují i větší cache. Při výpočtu více programových vláken za použití pouze jednoho jádra, vše obstará vysoká frekvence procesoru, neboť za stejnou jednotku času se zpracuje více dat sériově.

Naproti tomu dnešní high-endové grafické karty disponují 480-1600 výpočetními jednotkami, každé s podstatně nižší frekvenci a tedy i potřebnou velikost cache. To je skvělé pro aplikace které podporují práci na více procesorech a potřebují zpracovávat mnoho dat současně. Pokud ale zvládá využít pouze jeden procesor, výkon okamžitě klesá na rychlost poskytovanou jediným jádrem a to je díky horším parametrům oproti CPU mnohem pomalejší.

Sada používaných instrukcí pro GPU je uspořádána do jakéhosi seznamu - pipeline. Pipeline je rozdělena do mnoha menších operací jako je vytvoření 2D mnoha částí, které se nazývají polygony, jejich barevnost, nasvícení atd. Na konci je pak všechno složeno, a odesláno na výstup kde se vše vykreslí na monitor.

Při studiu tématu implementace vlastního rozhraní jsem poznal, že v současné době se k vývoji přistupuje způsobem „skládání“ aplikace již z hotových knihoven.
Tento postup má mnoho výhod:

- jednak se hodně urychlí vývoj, protože již nemusíme znovu navrhovat algoritmy, které již někdo vymyslel dávno před námi,
- je zaručena vysoká spolehlivost vyplívající z předešlých použití v jiných projektech,
- díky použití standardizované knihovny je zaručena přenositelnost i na různé platformy.

Proto je zbytečné kompletně si programovat zcela nového API, kdy i profesionálním mnohačlenným týmům může takový vývoj zabrat několik let. Mnohem výhodnější je využít služeb některého již hotové grafické knihovny.

Většina graficky náročných počítačových aplikací využívá pro vykreslení 3D grafiky v počítači se systémem Windows dvě možnosti: buď rozhraní OpenGL, nebo rozhraní DirectX. V počítačích se systémem Linux a počítačích Mac je možné použít pouze rozhraní OpenGL.
V dnešní době jsou obě na vysoké úrovni a kolem každého z nich se vytvořila ohromná komunita příznivců, kterým vyhovuje pravě ten přístup jejich API a za žádných okolností na něj nedají dopustit. Pojďme se teď podívat trochu do historie na to, jak postupoval vývoj jednotlivých rozhraní a na to, kam dále směřuje.

3 Historie OpenGL a DirectX

Na začátku 90.let vydává firma SGI (Silicon Graphics Inc - do té doby známý vývojář grafických stanic) API s názvem IRIS GL, které bylo navrženo s důrazem na to, aby bylo použitelné na různých typech grafických akcelerátorů a bylo funkční i v tom případě, že grafický akcelerátor úplně chyběl. Za této situace se využil plně softwarový výpočet.
IRIS se tak díky své inovativní koncepci stal jakýmsi „standardem“ na poli 3D.

Konkurence však brzy přišla s vlastním standard PHIGS, který se sice v mnoha aspektech ukazoval jako slabší než IRIS GL, ale s nárůstem dalších konkurentů používající toto rozhraní a s projevujícím se stářím IRIS GL začalo SGI ztrácet svůj prim. Přišli tedy na svou dobu s velmi radikálním řešením. Uvolnili svůj IrisGL jako opensource (volně šiřitelný otevřený programový standard, ten tak lze zdarma používat včetně zdrojového kódu a licence software) s cílem rozšířit ho do co nejvíce firem. Tento velice odvážný tah měl posléze za následek vydání prvního OpenGL [8] standardu, který přímo vycházel z Iris GL.

OpenGL byl, a stále je inovativní ve všech směrech- zavedl normalizovaný pohled na veškerý grafický hardware, a změnil přístup k odpovědnosti za další vývoj z programátorů aplikací na výrobce grafického hardwaru. Mnoho druhů grafických karet od různých výrobců tak začaly najednou mluvit jednotným jazykem. Tento počin měl nakonec významný pozitivní dopad při vývoji softwaru na všemožné platformy.

Po tomto úspěchu a zisku mnoha zkušeností založilo SGI v roce 1992 konsorcium ARB (architectural review board), do kterého patří firmy jako SGI, DEC, HP, IBM, Intel, Microsoft, Sun Microsystems a další, které udržovali a specifikovali standardy OpenGL do budoucna.

Jednou z mnoha zakládajících firem byl i Microsoft, který se však po vydání Windows 95 v roce 1995 odtrhl a vydal se svou vlastní cestou v podobě konkurenčních **DirectX.** Původně bylo OpenGL využíváno pouze v průmyslové a obchodní sféře, ale s příchodem Windows 95 a velkým boomem zábavního průmyslu se stalo též herním API. Největší zásluhu na zviditelnění OpenGL měla firma iD Software a její hlavní programátor John Carmack, který vydal první akcelerovanou hru Quake právě na OpenGL.

Další, důležitým datem v historii OpenGL je rok 2006, kdy SGI odchází z vedení vývoje. Ten převzala Khronos Group a vytvořilo novou skupinu OpenGL ARB, jejíž zakládajícími členy byly povětšinou lidé z nVidia, AMD a Appple, ale i výrobci softwaru tj. Blizzard, TransGaming, CodeWeavers, a jiní, kteří mají dnes plnou kontrolu nad další specifikací OpenGL. Zakládající členové se shodli na tom, že OpenGL je tady již 13 let a je potřeba ho přepracovat kompletně od začátku, aby obsahoval vše, co dnešní hardware může poskytnout. Proto v nedávné době vyšla OpenGL verze 3, což je úplně nové API, které již poprvé není zpětně kompatibilní. Dalším přínosem tohoto spojení je záruka dobré podpory, neboť za vývojem stojí přímo výrobci karet, a ne jako drivery od SGI.

OpenGL 3.0 však bylo zklamáním pro vývojáře her, kteří doufali v API s vlastnostmi, které by ukončilo dlouhodobou nadvládu DirectX v této oblasti. Khronos však využívá OpenGL 3.0 převážně v profesionální sféře v aplikacích jako je CAD/CAM a k tvorbě jejich 3D obsahu. Dnes tak neexistuje žádný větší vývojář her, který by herní engine pod OpenGL vyvíjel nebo provozoval. Poslední herním počinem založeným na OpenGL tak zůstává Doom 3 od Id Software.

Poslední verze OpenGL, kterou lze na internetu najít je 3.2, upravená oproti 3.0 pro větší výkon, lepší vizuální kvalitu, akcelerované zpracování geometrie a jednodušší portování Direct3D-aplikací. Dále byly zakomponovány všechny funkce od OpenGL 1.0, díky čemuž je opět možná zpětná kompatibilita se starším kódem, zároveň využívající funkcionalitu verze 3.2.
OpenGL tak nyní dosáhlo na úroveň Direct3D 10.1 a v některých ohledech má potenciál ho i překonat.

Historie DirectX [2] se datuje do roku 1995, kdy Microsoft vydává svůj nový OS Windows 95, od kterého velmi očekával, že výrazně zasáhne do vývoje ve všech dosavadních odvětvích počítačového světa. Okamžitě po jeho vydaní v roce 1995 se však projevili určité nedostatky v konstrukci grafického rozhraní GDI, které systém používal pro vykreslování grafických dat.

První pokus o jejich napravení se jmenoval WinG. Jednalo se o množinu funkcí, které umožňovaly přímý přístup do grafické paměti (s GDI to nebylo možné). Postupným vývojem vzniklo Game SDK, které se za nějakou dobu přejmenovalo na DirectX. Direct X tak spojoval kompromis mezi rychlým přístupem k HW, ale taktéž zachoval hardwarovou nezávislost systému.

Tato první zkušební DirectX verze nebyla zrovna přijata s velkým nadšením. Vývojáři zvyklí na DOS měli dosti velké problémy s novou architekturou tohoto rozhraní. Nabývali tak dojmu, že se jim snaží Microsoft vnutit jejich novou používanou architekturu Windows za každou cenou a chce je odlákat od zaběhlých a osvědčených technik, které znali z DOS. Postupem času však na DirectX začalo přecházet více programátorů a stalo se tak standardním nízkoúrovňovým rozhraním pro vývoj grafických (hlavně pak herních) aplikací na Windows.

Přichází r.1996 a nastupuje druhá generace DirectX, která jako nejzásadnější novinku přináší podporu 3D grafiky a to díky knihovně Direct3D. Další verze 3.0 se již stala skutečně dokonale fungujícím prostředím a od roku 1997 se veškeré vydané hry a aplikace vyvíjely přednostně v DX.

V průběhu let 1997 a 1998 se objevují první plně 3D hry, které používaly technologii HW akcelerace. DX se tak již dostal na takovou úroveň, že se mohl začít rovnat i s konkurenčním OpenGL.

Microsoft však svoje API nenechával zastarávat a prakticky každý rok vycházela obměněná verze, kterou se posunovala herní hranice grafiky stále o něco výše. Radikální změnu zažilo rozhraní v r. 2000 s příchodem verze 8, která byla „ořezána“ a celkově zjednodušena (zvláště pak Direct3D) a dala se tak využít i při programování aplikací, ve kterých dříve vysoce dominoval OpenGL.

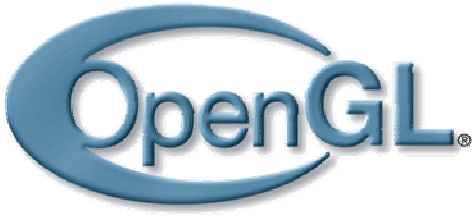
Další velkou změnou prošli DirectX před časem s příchodem verze 10. Tato verze je postavena na úplně nových standardech, které mají jen málo společného s předešlou verzí, což ale znamená, že nadále již nejsou podporovány systémy XP a starší.

Mnozí považovali tento krok pouze jako populistické gesto jak zvýšit prodej nového os Vista, nicméně celá věc má i své logické opodstatnění. DX10 je natolik odlišné API od DX9, že vyžadovalo i značné úpravy OS které by byli na XP nemyslitelné, z tohoto důvodu byl vydán i nový systém Vista.

Společně s 10 se na trhu objevuje i nová generace grafik s odlišnou architekturou čipů, která se i nadále stále více zrychluje a zlepšuje. DX10 ale nebyl zcela dokonalí a Microsoft v době jeho vydaní do něj nestačil implementovat vše, co zamýšlel. Následuje upgrade na verzi 10.1 který již vše dává do pořádku.

Nyní přichází nový Windows 7 který sebou opět přináší i novou verzi DX s pořadovým číslem 11. DX 11 přímo vycházejí z 10.1 a budou dostupné na operační systémy Vista, Windows7 a novější.

3.1 OpenGL



Obr.3.1 logo OpenGL

Samotné rozhraní je v současné verzi tvořeno z 250 grafických a modelačních funkcí (200 jich tvoří samotné jádro a zbylých 50 jsou rozšiřující utility), které obstarávají veškeré operace potřebné k vytvoření aplikace.

OpenGL je definován jako programové rozhraní implementované výrobci grafických karet přímo do svých ovladačů karty. Pokud jejich HW přímo nepodporuje nějakou funkci ze standardu OpenGL, je takováto funkce provedena alespoň softwarově byť za cenu mírného zpomalení výkonu. Obdobně se postupuje pokud grafický akcelerátor úplně chybí nebo pokud tvůrce aplikace věří, že výpočet proběhne rychleji než na grafickém hardwaru – např. pokud má karta hardwarové možnosti osvětlení, ale je pomalejší než ta samá operace prováděná CPU, pak může ovladač karty využít softwarový výpočet CPU.

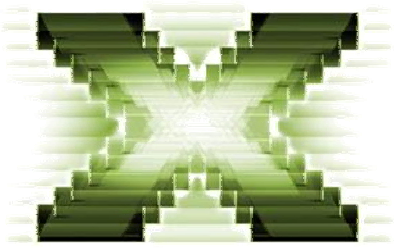
Knihovna OpenGL je koncipována jako systémově nezávislé, tzn. že v jeho jádře není přímo specifikováno, jaký systém nebo jaké rozhraní se musí používat. Lze se tak s ním setkat na stanicích, kde je nainstalovaný OS Windows, bez problému poběží na jakékoliv distribuci Linuxu, či na Apple PC s jejich MAC OS. Dokonce je možné se s nimi setkat i v aplikacích vyvíjených pro některé herní stanice jako je Playstation 3 firmy Sony.

Takováto přenositelnost je možná díky některým řešením které jsou sice již standardem mnohých API, ale v případě OpenGL jsou dotaženy do detailů. Například většina API sice neobsahuje žádné příkazy pro práci s okny či zpracovávání uživatelských vstupů, (funkce pro vytváření a rušení oken poskytuje operační systém nebo se použije k tomu vytvořená nadstavbová knihovna), ale OpenGL je jedno z mála rozhraní která tento fakt zcela využívá ke své prospěchu a stává se plně platformově nezávislé.

Za určité omezení může být považováno vypuštění příkazů pro popisování kompletních 3d objektů. Samotné jádro je tak schopno vykreslovat pouze primitiva jako jsou body, hrany, polygony.Tento způsob vykreslování je však nejlépe chápán pro grafické karty, výsledný model tak lze mnohem rychleji vykreslit. Pokud je přeci jen potřeba vkládat složitější tvary lze použít knihovnu GLU, která obsahuje popis základních geometrických tvarů.

Pzn. Co se týká podpory OpenGl v systémech Windows jedná se o dosti sporné téma. Postoj Microsoft je totiž k OpenGL nemilosrdný a tvrdě se jej snaží v plné míře nahradit vlastním DirectX. S grafickým urychlováním pomocí technologie OpenGL byly vždy ve Windows problémy. Začalo to již při vydaní Windows X, kdy Microsoft rozhodl, že ho podporovat nebude. Úplně stejný problém nastal před časem po vydaní Windows Vista. Výrobci čipů pro grafické karty ATI a nVidia se rozhodli, že to nemohou jen tak nechat (už kvůli podpoře OpenGL v Linuxu). Východiskem tak je stažení aktualizovaných ovladačů ze serveru výrobce čipu, kde je podpora OpenGL pro Windows již implementována.

3.2 Direct X



Obr.3.2 Logo DirectX

Pokud se pracuje v systému Windows a je potřeba vytvořit jakoukoli grafickou aplikaci, jsou zde hned dvě možná řešení. První možností je sáhnout po základní nástroji operačního systému Windows nazývaném GDI (Graphics Device Interface). Základním cílem GDI je podpora grafiky nezávislé na zařízení. Tento přístup měl umožňovat programátorovi používat dostupné funkce bez nutnosti znalosti typu konkrétní grafické karty.

Když chce aplikace vytvořená ve Windows něco vykreslit, tak zavolá systémovou funkci. Jako parametr ji určí oblast okna kam se má vykreslovat. Systém pošle obratem oknu aplikace zprávu s žádostí o překreslení obsahu okna. Dokud se okno nepřekreslí, systém žádné další zprávy neposílá. Operační systém se pak pomocí ovladačů postará o zajištění výpočtů. Výsledky se ukládají do paměti karty.

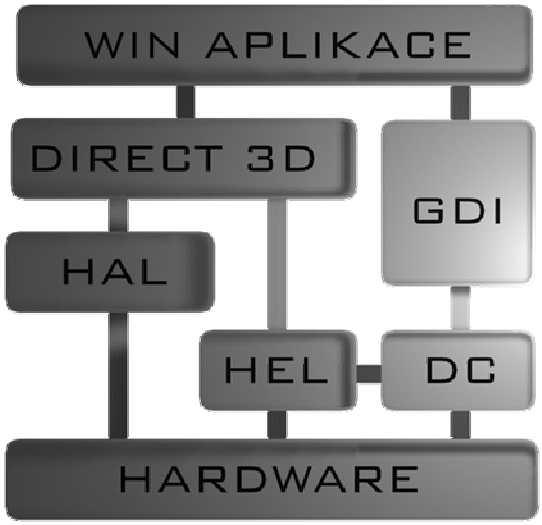
Pomocí funkcí GDI, si již aplikace překreslí data uložená v paměti karty přímo na monitor. Aplikace pak volá vykreslovací funkci GDI při každé změně obsahu okna– když to však dělá moc často, systém nechá překreslit okno jen „občas“ , to jak často, záleží na rychlosti počítače. Obecně se tak může stát, že hardware zobrazí obraz částečně starý a částečně nový, protože „monitor nepočká“, až se připraví celý nový obraz. To může způsobit občasné probliknutí vykreslovaného obrazu.

Zásadním nedostatkem GDI je však jeho velmi nízká rychlost renderování oproti ostatním technikám. Je pomalé protože nepřistupuje přímo k hardwaru, ale k API operačního systému, oproti tomu DirectX úplně OS obchází. Dalším nevýhodou je minimální použití HW akcelerace a dále závislost na použitém rozlišení.

Další volbou je grafické rozhraní, které se označuje jako DirectX. Na úvod je třeba předeslat, že DirectX je mnohem robustnější a rychlejší než práce s GDI.

DirectX [9] je rozhraní vytvořené pro vykreslování 2D a dnes hlavně 3D grafických aplikací a obzvláště pak multimediálních her pro systém Windows a nyní i herní stanice XBox.Toto rozhraní má zprostředkovávat komunikace mezi aplikací a grafickou kartou, kde veškerá tato komunikace probíhá na abstraktní úrovni která je sice velmi podobna původnímu rozhraní GDI, avšak zásadním rozdílem mezi Direct3D a GDI je, že DirectX je již přímo připojen k ovladačům monitoru (odtud název Direct neboli Přímo, což je přesně to, jak DirectX pracuje) a má tak mnohem lepší vykreslovací parametry.

DirectX rozhraní je vždy na všech PC shodné a odpovídá danému standardu pro právě nainstalovanou verzi. Musí být tak schopen vykonat veškeré funkce, které aktuální verze nabízí i za podmínky, že by HW toho normálně nebyl schopen (teoreticky tam musí být schopen vykreslit i moderní efekty na hardwaru, který je normálně nepodporuje - požadované výpočty by pak měl provést sám) .



Obr.3.3 Na obrázku je tak vidět možné uspořádání aplikace, která využívá funkce DirectX společně s GDI, kdy obě mají přístup ke grafické kartě a mohou zprostředkovat její funkce.

HEL (Hardware Emulation Layer), tato vrstva do sebe naváže dostupné ovladače karty, aby s ní mohl oboustranně komunikovat a zjišťovat, jaké funkce je ještě karta schopna zvládnout, a o které se musí postarat sám DirectX. Pokud tedy danou funkci karta nepodporuje výpočet se provede softwarově pomocí této vrstvy, pokud ano, zavolá se vrstva HAL, která již požádá HW.

Poslední vrstva která se zde využívá je HAL (Hardware Abstraction Layer), vytváří spojení mezi hardwarovým zařízením a softwarovým vybavením počítače. Hlavním úlohou této vrstvy je skrýt detaily v ovládání a přístupu k rozdílným typům zařízení, a definovat standardní rozhraní tak, že emuluje novou abstraktní vrstvu s jednoduchými funkcemi. Komunikace na abstraktní úrovni tak nejen zjednodušuje vývojářům práci tím, že jednotlivé aplikace nemusejí být upravovány vždy na konkrétní typ hardwaru, ale umožňuje aplikaci používat i zcela nové funkce a zařízení, o kterých se v době vzniku programu ještě ani nemohlo uvažovat (díky abstrakci není z hlediska programovaní např. rozdíl mezi čtením dat z pevného disku, DVD, serveru, flash....).

Značnou nevýhodou DirectX je jejich striktní použití pouze na stanicích se systémem Windows. Práce na Windows tak může být bez potíží, ale pokud je třeba spustit aplikaci na UNIX/Linux narazí se na obrovský problém. Nyní již existují různé emulátory, ale stále není 100% záruku na funkčnost kódu, a už vůbec ne na identickou rychlost aplikace.

Za možnou výhodu oproti OpenGL lze považovat komplexnost služeb které DirectX poskytuje.Obsahuje totiž komplexní soubor knihoven, které poskytují i funkce nad rámec pouhého vykreslování 3D grafiky. Nabízí tak funkce pro obsluhu 2d a 3D grafiku, zvuku, vstupů, výstupů, což právě ocení hlavně vývojáři herních aplikací neboť vše co potřebují zde mají na jednom místě. DirectX se tak obsahuje:

- *DirectDraw*: knihovna starající se o zobrazení 2D grafiky (ploch) a mapování textur. Po nástupu 3D tato knihovna prakticky ztrácí ve vývoji svůj význam.
- *Direct3D*: přímý nástupce DirectDraw, který se v dnešní době stal nejhlavnější součástí DX a řídí veškerý grafický výstup aplikací.
- *Direct Sound*: jak už název říká půjde o rozhraní určené pro správu a přehrávání zvuku, zvukových stop s podporou 3D digitálního zvuku. Současné moderní zvukové karty tak zvládnou simulovat i 3D zvukové efekty apod.
- *Direct Input*: komunikátor mezi všemi možnými vstupními zařízeními jako klávesnice, myš, joystyk.
- *Direct setup*: knihovna zajišťující správu aktuální verze DX a její případný upgrade.

3.3 Výhody a nevýhody:

DirectX	OpenGL
Direct3D je objektově orientované	OpenGL byl původně koncipován jako strukturovaný, nyní jsou již i verze objektové
Vývoj nejnovějších funkcí je ve společné kompetenci největších výrobců čipů grafických karet AMD/ATI, nVidia a společnosti Microsoft	Skupina Khronos stojící za OpenGL se bohužel na vývoji shaderů přímo nepodílí, naopak se zpožděním standardizuje (zavádí do API) novinky z této oblasti.
Vyhrazen striktně pro OS Window	Je multiplatformní
Direct3D vždy drželo krok s novým hardwarem a nové vlastnosti jsou přidávány vždy s novou verzí	OpenGl přidává nové funkce do svého jádra pomocí rozšiřujících utilit
Obsahuje komplexní soubor knihoven specializovaný na vývoj herních aplikací, poskytuje knihovny pro vývoj grafiky, zvuku, ovládání..	OpenGL je jen a pouze pro rendering, vše navíc je potřeba vytvořit jinými nástroji (knihovnami),samotné API však poskytuje funkcionalitu pro široké pole 3D aplikací, nikoli pouze jen pro hry.
ovladače pro DirectX tvoří 2 společnosti – jak dodavatel HW + Microsoft	pro OpenGL je to pouze dodavatel HW
DX jsou primárně určeny pro C++, C#, existuje i verze podporující Java 3D	OpenGl je podporován většinou dnešních programovacích jazyků například Fortran, Object Pascal či Java

3.4 Shrnutí použití obou API

Obě rozhraní vznikly v odlišné době, jsou postaveny na jiných základech, byli zkonstruované pro jiné účely a každé má své klady i zápory.

OpenGL je již od počátku knihovna zaměřená na procesuální 3D grafiku, zatímco DirectX je celý soubor knihoven pro práci s grafikou, zvukem atd. Jeho primárním účelem je poskytnout vývojáři her kompletní sortiment služeb pro jeho aplikaci.

Toto rozdělení je částečně dáno i jejich „společnou“ historii. Hodně profesionálních grafických studií vyvíjela své aplikace na pracovních stanicích SGI s jejich API IrixGL. Po vydání OpenGl pak plynule přešli na tento standart. Navíc většina tehdy dostupných karet ze začátku podporovala pouze OpenGL.

Direct na rozdíl od OpenGL si nikdy nekladly za prvořadý cíl výrazně ovládnou trh s profesionální grafikou, namísto toho se soustředila hlavně na vývoj aplikací pro herní průmysl.

V současné době již DirectX technologicky převýšil OpenGL a tvůrci her mají stále menší chuť a motivaci kompilovat své aplikace pro obě platformy, proto si logicky vyberou technologicky vyspělejší DirectX.

Pomalu se tak zažil fakt, že inženýrské aplikace (CAD/CAM, Catia apod.) využívají OpenGL vzhledem ke snazší implementaci, multiplatformovosti a menší nutnosti fotorealistického real-time3D zobrazení. Z dnešních renomovaných softwarů využívá jádro OpenGl např.3D Studio MAX, Maya, LightWave. Z technické oblasti (CAD aplikace) pak bych zmínil CATIA.

Zatímco herní (codemasters,EA) či obecně multimediální vývojáři softwaru spíše sáhnou po rozhraní DirectX.

Po zvážení všech dosavadních skutečností se volba OpenGl do zadaní této práce byla velmi vhodná. Jednak je možné ho použití na více platformách, pak dobrá podpora jazyka C++ a velké množství online literatury zabývající se tímto tématem. Další předností je jeho vývoj, který je přímo zaměřen na převážné použití v technické aplikace.

4 Standardy OPEN GL

Před začátek popisu praktického použití jádra [10] , bude jistě dobré se seznámit s některými specifikacemi které openGL zavádí. Nejdříve demonstrační příklad jedné z funkcí:

Př. `glNormal3f( 0.0f, 0.0f, 1.0f);`

Všechny použité funkce vždy začínají prefixem knihovny do které patří. Např. všechny funkce vycházející z knihovny GLUT začínají prefixem glut, v našem případě má funkce Normal svou definici v knihovně gl. Po prefixu se zapisuje název funkce, ten většinou přímo koresponduje s vlastnostmi použité funkce - např. glColor3f nastavuje RGB složky barvy, obdobně lze odvodit význam glVertex, glNormal...

Za názvem funkce ve většině případů následuje počet parametrů které daná funkce má. Na konec se ještě zadává písmeno udávající datový typ očekávaných parametrů (stručný přehled datových typů které OpenGL používá viz tabulka níže).

Pokud je název funkce složen z více než jednoho slova, zapisuje se formou tzv. velbloudího stylu zápisu, každé nové slovo názvu funkce začíná velkým písmenem.

např.`glEnable`

Odlišný přístup se volí při zápisu konstant, ty se zapisují většinou velkými písmeny. Pro oddělování slov v názvu proměnné se proto užije znak ‚_‘ :

např. `GL_DEPTH_TEST`

Pro maximální nezávislost na systému jsou zavedeny nové datové typy:

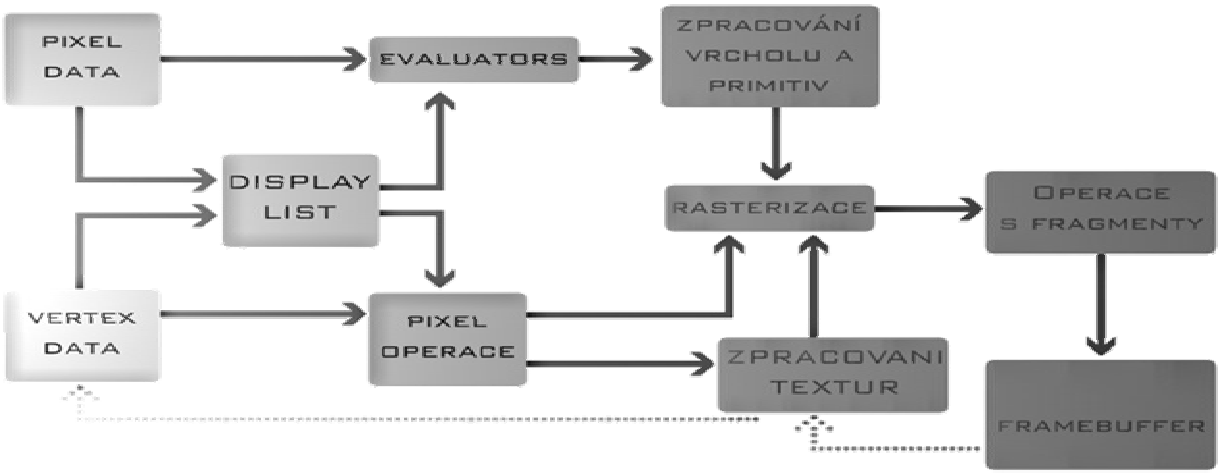
Zkratka dat.typu v OGL	Datový typ v C	Definice v OGL
i	int	GLint
s	short	GLshort
d	double	GLdouble
b	Signed char	GLbyte
f	float	GLfloat

4.1 Architektura OpenGL

V této kapitole bude podrobně popsán vykreslovací proces API, co je potřeba udělat, aby byl na konci vykreslen správný obraz.

Celé rozhraní je koncipováno jako stavový automat. Každá stavová proměnná má na začátku přidělenu nějakou hodnotu, aplikace pak během vykreslování tyto stavy mění tak, abychom dosáhli zobrazení, jaké požadujeme. Aktuální stav platí tak dlouho, dokud není jiným příkazem změněn nebo není ukončena aplikace.

Ať už se vytváří pouze prázdné okno nebo vykreslují složité tvary je třeba zachovat určitou posloupnost procesů, která se nazývá pipeline. Pipeline není striktně daná, je ale dobrým pomocníkem jak správně poskládat následující sled operací.



Obr.4.1 schéma funkce OpenGL

Diagram ukazuje jak jednotlivé vrcholy prochází vykreslovacím procesem podobně jako automobily na montážní lince. Na jeho konci je tzv. framebuffer, do kterého se zapisují jednotlivé pixely v podobě, v jaké budou zobrazeny na obrazovce. V průběhu tohoto procesu procházejí vrcholy řadou mezi zastávek, kde je postupně transformují, obarvují
(příp. otexturují), nebo na nich vypočítají osvětlení, atd.

4.2 Display list

Prvním krokem kterým se začíná je seřazení všech dat, ať vertexových nebo pixelových do určitého seznamu (display listu). *Display list* obsahuje informace o veškerých objektech, které mají být ve finále zobrazeny. Seznam je možné snadno naplnit libovolným počtem objektů nebo jejich skupin, lze je snadno upravovat, kopírovat, manipulovat s nimi nebo pokud nám nevyhovují, tak i snadno smazat. Když je seznam naplněn, data v něm uložená jsou odeslána k dalšímu zpracování.

Vyčíslení objektů (evaluators)

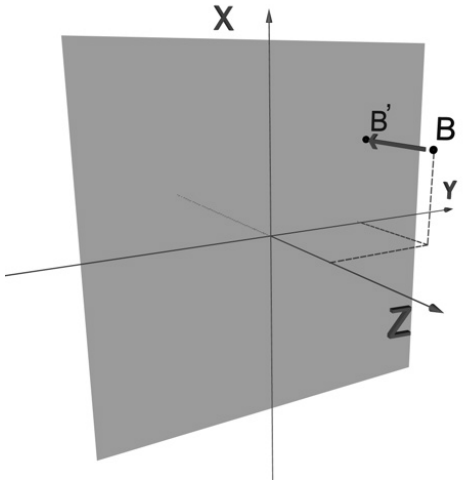
U všech složitějších tvarů (parametrické křivky,složené povrchy) je zapotřebí vypočítat umístění jednotlivých bodů, ze kterých se vytvoří trojúhelníky (tento proces se nazývá pasterizace).

Po vyhodnocení všech vlastností řídích bodů hledaných ploch se na evaluátoru objeví žádané souřadnice jednotlivých vertexů. Současně se zpracováváním bodů a polygonů se vytvoří i normálové vektory a počítá se osvětlení.

Jelikož zpracovávaná 3D data jsou nakonec zobrazována na 2D monitoru je potřeba je transformovat z prostorových na souřadnice monitoru. To se provede přenásobením všech bodů projekční maticí o velikosti 4x4.

Použití matice 4x4 vyplívá z použití *homogenních souřadnic*. Zde je každý bod (hrana, vektor) tvořen ze 3 souřadnic prostoru `[ x, y, z ]` a homogenní souřadnicí w udávající jeho váhu (*weight*– odtud je zřejmé i její jednopísmenné označení). V případě že tato 4. složka není zadána, OpenGL ji doplní hodnotou 1 pro bod a 0 pro vektor (prostorová data se tím nijak nezmění)

Transformace mezi 3D a homogenními souřadnicemi se pak provede součinem prvních 3 složek složkou 4. Bod o souřadnicích (X,Y,Z) pak bude mít v homogenní soustavě tvar (XW, YW, ZW, W). Zpátky do kartézských souřadnic se dostaneme podělením prvních 3 složek složkou 4.



Obr.4.2 Proces transformace do homogenních souřadnic si lze představit jako vržení stínu bodu `B` do průmětny `z = 0` svazkem paprsků rovnoběžných s osou `z`.

Pokud je použita textura, nechá se vytvořit i texturovací souřadnice, je provedeno oříznutí scény zadanou ořezávací rovinou na velikost projekčního okna (monitoru) a zbylé vertexi jsou vyřazeny z dalšího zpracování.

4.3 Zpracování pixelů

U pixelů se dekódují informace o jejich umístění a barvě z formátu, který používá aplikace do formátu, kterému rozumí grafická karta (tento postup spočívá v rozložení tří, popř. čtyř barevných složek, změně kontrastu, měřítka).

Po provedení těchto operací se přechází *k rasterizaci*- tento proces spojí zatím samostatně probíhající části- geometrickou a obrazovou, následkem čehož dojde k převedení projekce trojrozměrné scény na „2D“ rastrový obrázek.
Po vytvoření 2D obrazu již lze nanášet na jednotlivé body objektu jejich definovanou barvu, osvětlení a pokud má přidělen texturu, tak v závislosti na normálovém vektoru i namapovat texturu.

Výsledek práce rasterizace však ještě nejsou pixely nýbrž fragmenty konečného obrazu. Tyto fragmenty v sobě nesou informace o barvě, hloubce, pozici v přiřazené textuře. Všechny fragmenty pak ještě projdou serií testů a kontrol a pokud ji splní, teprve pak se stanou obrazovými pixely. Pixely jsou odeslány do framebufferu ze kterého již složený obraz lze rovnou zobrazit v okně nebo jej načíst zpět do hlavní paměti procesoru a jednotlivé pixely dále upravovat pomocí přidaných rozšíření.

4.4 Framebuffer

Framebuffer [3] je určitá část paměti grafické karty vyhrazená k dočasnému uložení již vypočtených snímků nebo spíše jejich částí. Data z Frame Bufferu jsou poté zobrazeny přímo na monitoru. Informace v této paměti se obvykle skládají z číselných hodnot barev pro každý pixel (obrazový bod, který se zobrazuje) na obrazovce.

Framebuffer je složen z dalších 4 dílčích podřízených bufferu, kde každý je přímo určen pro specifickou práci. Některý obsahuje informace o barvě pixelu, další o jeho hloube apod. Proto je nutné před konečným načtením dat zadat, který ze 4 bufferů bude použit a specifikovat jeho bitovou hloubku.

Pro další potřeby vykreslovaní je asi nejdůležitější ten, který v sobě uchová informace o barvě všech fragmentů (*colour buffer*). V případě základního zobrazení součásti se používá pouze jeden colour buffer, je však možné jich inicializovat hned několik. Počet je pak dán jen kapacitou grafického HW, rozlišením jaké zvládne monitor, barevnou hloubkou atd.

V moderních grafických HW lze použít alespoň 2 colour buffery, které tvoří základ pro tzv. double buffering technologii. Ta je založena na principu vykreslování do virtuálního bufferu (tzv.přední– front bufer) se současným zobrazováním do druhého (zadního– back bufferu). Výstupem této operace pak může být soubor stereo obrázků generovaných speciálně pro ne zrovna běžné výstupní periferie (3D brýle, stereo monitory, apod.).
Je ale dobré brát na vědomí, že každý další buffer zabírá více a více paměti a zmenšuje tím kapacitu grafického HW, která se zároveň používá pro zpracování textur a display listu. Z tohoto důvodu je pak vhodné povolovat pouze ty, které jsou pro aplikaci bezpodmínečně potřebné.

Další z řady bufferu je *depth buffer* někdy také nazývaný Z-buffer. Z českého překladu již vyplývá, že půjde o bufferu uchovávající hodnotu hloubky pro každý pixel. Hloubkový buffer poskytuje informace potřebné pro vykreslení viditelných částí těles, tzn. zda jsou vzdálenější plochy překryty plochami bližšími nebo naopak. Nastavení funkce a velikosti hloubky lze při vykreslování ještě měnit a může se tak docílit zobrazení různých řezů a ploch.

Při pasterizaci se načte souřadnice Z-osy (osa jdoucí „do nebo ven z“ monitoru) a porovnává se s souřadnicí vkládaného fragmentu objektu, již uloženou ve framebufferu. Pokud aplikace dospěje k závěru, že tato souřadnice je menší než ta původní ve framebufferu, tak se ona souřadnice přepíše, pokud je větší tak to znamená, že je fragment ve výsledné scéně zakrytý a je vyřazen.

U určování této hloubky je velmi podstatným parametrem bitová hloubka-např. u 8bitové hloubky lze rozlišit vzdálenosti 256 fragmentu, u 16 bitové je to už 65536 atd. Proto se dnes již převážně používají 16 a 24 bitové hloubky.

Stencil buffer-Poměrně běžné také bývá nechat si nějaký kus paměti vyhrazen pro jakési šablony. Určuje nám pozice na obrazovce, ve kterých je dovoleno vykreslovat. Pro nejsnazší popis celkové práce tohoto bufferu lze uvést příklad aplikace, která simuluje jízdu autem. Vytvoříme si palubní desku a veškeré ovládací prvky, a ty uložíme do Stencil buffer. Ten se pak postará aby se při pohybu auta aktualizoval pouze obraz viditelný vně okna a nikoli i zbytek obrazovky s interiérem.

Accumulation buffer-obdobně jako colour bufer v sobě uchovává data o barvách RGBA, narozdíl od něj je však schopen hromadit (spojovat) více obrázků do sebe. Výsledkem pak může být efekt rozmazaného rychle se točícího obrobku, vřetena, listu vrtule a jiné.

4.5 Použití nadstavbových knihoven

Jak šlo poznat z předešlých kapitol, samotné jádro OpenGL obsahuje velmi silné nástroje pro renderování grafiky, nikoli však příkazy pro samotnou práci s okny. Naneštěstí ale výsledná aplikace určitě bude vyžadovat alespoň základních mechanizmů pro vytvoření okna a práci s nimi. Proto jsou společně se základním jádrem použity i některé nadstavbové knihovny přidávající nové funkce a možnosti (např. správa oken, poslední shadery, vyhlazování hran apod.). Jestliže je zavolána některá z funkcí rozšiřující knihovny, celý algoritmus se načte, a automaticky transformuje na jednotlivé příkazy potřebné pro knihovnu OpenGL. Přidání jakékoliv rozšiřující knihovny se provádí standardním příkazem.

```
#include <požadovaná knihovna>
```

4.5.1 GLU knihovna

GLU (GL utility) je součástí prakticky každé implementace OpenGL. Tato knihovna doplňuje mechanizmy zapouzdření, provádí mimo jiné i generování povrchu těles. Použitím této knihovny se tak mnohem zjednoduší a zrychlí práce především s NURBS křivkami a plochami, vykreslování kvadrik (koule, válce, kužele, disku,...). Dalším přínosem je funkce pro zjištění chyb při běhu aplikace a jejich příčin.

Knihovna GLU však nemá klasický přístup k zobrazování chyb jako jazyk C, kdy chybová část kódu vrátí specifickou hodnotu. (např. pokud je vše správně, vrátí se hodnota 0
při chybě 1). Zde je při zjištění chyby zavolána předem připravená funkce callback, která již zvládne na danou situaci patřičně zareagovat (funkci je pak současně předána i hodnota vyskytlé chyby).

4.5.2 Knihovna GLUT a WIN32

Ani knihovna GLU bohužel neřeší problém s vytvořením okna pro aplikaci.
Proto byla pro OpenGL sestavena další rozšiřující knihovna GLUT
(OpenGL Utility Toolkit). GLUT [11] definuje a implementuje aplikační rozhraní pro tvorbu oken a jednoduchého grafického uživatelského rozhraní, přičemž jako třešnička na dortu je také systémově nezávislá, tj. pro práci s okny se na všech systémech používají vždy stejné funkce, které mají stejné parametry.

> **Pozn.** Skutečnost, že lze OpenGL spustit na různých platformách, neznamená u stejné aplikace vždy naprosto identický výsledek - pokud se udělá přímé srovnání stejných rastrových obrázků na dvou platformách, může se ukázat mírné rozdíly v jednotlivých barvách. Tento jev je způsoben různou prezentací čísel barev od jednotlivých výrobců, souřadnic jednotlivých textur nebo její bitové hloubky. Geometrie a uspořádání objektu se však ve výsledku nikdy nezmění!

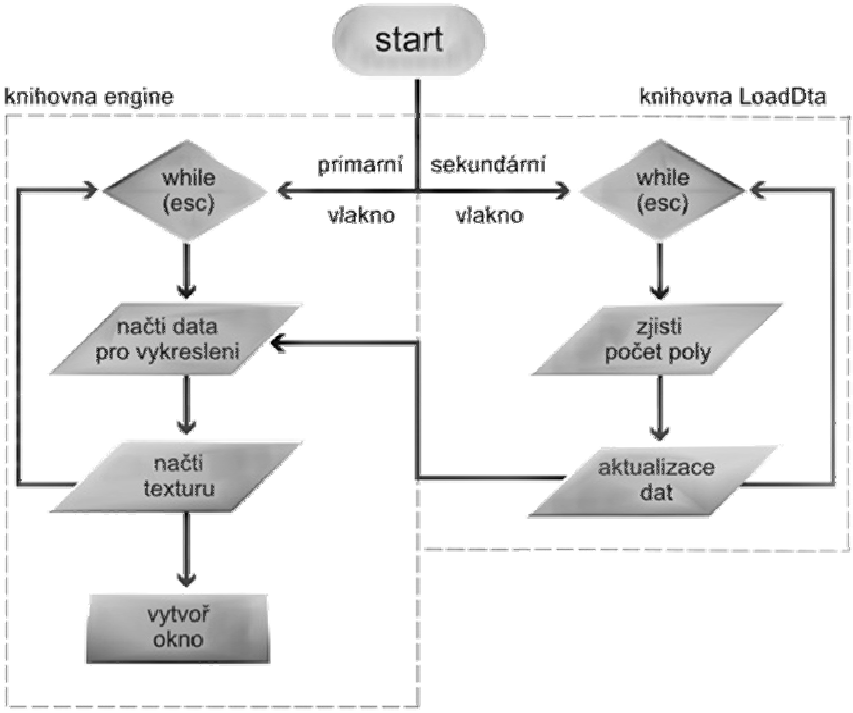
Další možností, která je k dispozici, je použit rozhraní poskytované systémem. Jelikož vývoj probíhá v operačním systému Windows jedná se samozřejmě o rozhraní WIN32 s kombinací knihovny windows.h. WIN32 bylo popsáno již v předešlých kapitolách, proto jen zmíním že obsahuje veškeré funkce, datové typy a struktury, které nám umožní manipulovat s vytvořeným oknem, aniž bychom tyto funkce museli sami zdlouhavě vymýšlet.
O použití tohoto způsobu vytvoření okna OpenGL lze ovšem uvažovat jen a pouze v operačních systémech Windows, protože funkce které tato knihovna používá pro nastavení render contextu nejsou podporovány v ostatních operačních systémech!!!

Oba způsoby mají něco do sebe, a nebylo by zcela korektní zaměřit se pouze na jeden z nich, ve výsledku tak není vytvořena pouze jedna verze aplikace, ale hned dvě. Jedna vytváří okna pomocí WIN32 – ta je optimalizována přímo pro OS Windows a druhá verze používá pro vytvoření okna univerzální knihovnu GLUT. Použití knihovny GLUT tak umožňuje za určitých podmínek konvertibilitu i do systému UNIX. Zároveň tak lze porovnat rozdíl v implementaci a délce kódu obou způsobů tvorby oken.

5 Realizace vykreslovací aplikace

Tato kapitola již opouští teoretickou část a přechází se k vlastní programové části projektu. Na následujících řádcích jsou rozebrány použité funkce, je popsán způsob jejich volání a umístění v jednotlivých knihovnách tvořících celý program.

Obě verze aplikace fungují podle zjednodušeného vývojového diagramu na obr. 5.1. Program je rozdělen do několika samostatných knihoven.



Obr. 5.1 Stručný vývojový diagram funkce aplikace

Obě verze aplikace pak mají shodnou většinu těchto knihoven, nejzásadnější rozdíl mezi nimi je tak v knihovně engine, ve kterých jsou použity rozdílné rozhraní pro tvorbu oken a dále v nově použité knihovně GLUT.

Nejprve tedy budou popsány prvky, společné pro obě verze a až poté bude objasněn zásadní rozdíl v engine knihovnách.

5.1 Knihovna Start

Knihovna start zajišťuje spuštění celé aplikace. Při implementaci jsem narazil na problém, nutnosti souběžného zpracování a načítání dat z externího zdroje se současným vykreslováním scény. Po konzultaci jsem řešení nalezl, a to formou vláken [5] (anglicky threads). Vlákna se používají v tom případě, kdy potřebujeme, aby program pracoval na několika úkolech současně.

Vlákna již nejsou součástí jazyka C nebo C++ nebo OpenGL, ale jsou přímou součástí používaného OS (za určitých podmínek dokonce lze vytvářet i vlákna čistě aplikačně bez OS- je nutná pouze vláknová knihovna (thread library), která je zodpovědná za jejich zprávu).

Pro sekundární vlákno jsem si vytvořil funkci *Thread,* ve které je definován cyklus, který má za úkol prohlížet soubor body.txt obsahující vykreslovaná data, a hledat symbol # pro začátek nového polygonu. Tento cyklus má pak nastavený časový interval 0,5s kdy aktualizuje proměnou na začátku souboru reprezentující celkový počet polygonů v souboru.

V main funkci této knihovny je pak spuštěno vlákno, ve kterém probíhá moje funkce *Thread* do té doby, než dojde k jejímu ukončení stisknutím klávesy Esc. Dalším zajímavým řešením je spouštění knihovny engine v primárním vlákně, které se provádí pomocí funkce systém. Funkce má pouze jediný argument (textový řetězec), který určuje jakou aplikaci chceme spustit. Po spuštění si knihovna engine zjišťuje počet polygonů, které má vykreslit a dochází k prvnímu vykreslení okna.
Podrobnější popis vytvoření a zprávy vláken ve Windows se nachází v příloze této práce.

5.2.1 Knihovna LoadData

Vytvořit knihovnu, která bude schopna dynamicky načítat data z externího zdroje byla jedna z největších výzev před kterou projekt stál. Současně se ale muselo jednat o co nejjednodušší a nejrychlejší proces. Proto pro ukládání dat nebyl použit nějaký zbytečně složitý formát, ze kterého by bylo nutné je ještě nějak konvertovat, nýbrž jako nejvhodnějším řešení se ukázalo být vytvořit si zcela vlastní formát. Jak již bylo popsáno dříve, jako nejvhodnější volba se ukázalo být použití formátu *.txt.

Pro jednoduchost řešení jsem se proto rozhodl celou vizualizaci řešit pomocí vykreslování trojúhelníků, které jsou pak načítány ze souboru umístěného na adrese `/data/body.txt`.

Načítání funkce *LoadDta* nejprve alokuje dostatečné místo pro všechny budoucí polygony, posléze spustí for cyklus, který běží od 0 do n, kde n je počet polygonů aktualizovaný knihovnou start.

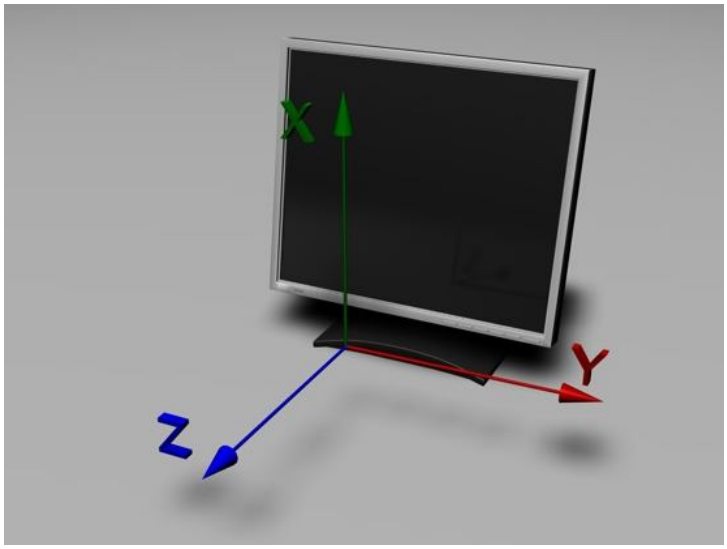
Začíná čtení dat ze souboru, které probíhá od shora dolů, kdy každé 3 následující řádky jsou automaticky považovány za nový polygon. Z každého jednotlivého řádku se pak načte trojice prostorových souřadnic tvořící jeden bod.

Jelikož jsou mezi těmito souřadnicemi i mezery a prázdné řádky, čímž by mohlo dojít k chybné interpretaci dat, je tento fakt ošetřen výjimkou, která tyto znaky ignoruje. Pro souřadnice bodů je pak v této funkci definována vlastní struktura VERTEX, do které jsou ukládány.

5.2.2 Struktura datového souboru

To, odkud jsou data načítána je tedy již jasné, v dalším textu bude ještě vysvětleno v jakém formátu se data mají zapisovat.

Tak jako ve většině moderních grafických prostředích (např. AutoCad,ProEngenner) i v OpenGL se používá kartézský souřadný systém s počátkem X, Y, Z: 0, 0, 0 ležící ve středu projekčního plátna. Osa X nabývá pozitivních hodnot směrem doprava, Y hodnoty se zvětšují směrem nahoru na monitoru a osa Z má kladný směr od počátku souřadného systému k pozorovateli (tedy ven z monitoru) a záporný směr od pozorovatele.



Obr.5.2 Schéma souřadného systému použitého v OpenGL

Popis povrchu probíhá formou polygonální sítě [6], která je tvořena množinou hran, hrany jsou tvořeny vždy z 2 a více bodů atd. Z praktického hlediska je pro popis takového povrchu zapotřebí ohromné množství trojic [X, Y, Z] reprezentující body na povrchu. Důležité ale je, jak tyto trojice souřadnic zapisovat.

Struktura datového souboru tedy vypadá takto:

Na prvním řádku je číslo udávající počet trojúhelníků tvořících scénu, tento údaj se mění automaticky takže není potřeba ho nějak upravovat.

Na dalších řádcích je pak uveden seznam bodů jednotlivých polygonů. Tento seznam má následující pravidla:

- Každý samostatný polygon je tvořen 3 řádky o 3 souřadnicích x, y, z.
- Na konci prvního řádku každého nového polygonu je znak # - napsaná funkce v knihovně *Start* tento znak hledá a v případě že tento znak bude chybět, nebude daný polygon rozpoznán.
- Při dodržení těchto pravidel je pak přidávání polygonů objektu možné kdykoliv za běhu aplikace.

Obecný zápis jednoho polygonu vypadá:

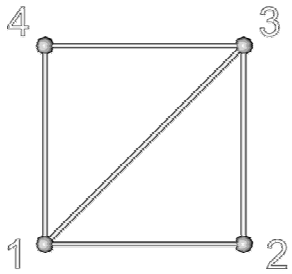
```
Př :            Bod1 X  Bod1 Y Bod1 Z  #
                Bod2 X  Bod2 Y Bod2 Z
                Bod3 X  Bod3 Y Bod3 Z
```

Praktický zápis souřadnic jednoduchého modelu, složeného ze 2 navazujících polygonů se provede:

```
(1)             -0.5  0.0  -0.56  #    //začátek prvního polygonu
(2)             -0.5  0.0   0.55
(3)             -0.5  0.7   0.55

(4)             -0.5  0.0  -0.56  #   //začátek druhého polygonu
(5)             -0.5  0.7  0.55
(6)             -0.5  0.7  -0.56
```

Při důkladném prozkoumání předcházejícího příkladu si lze velmi snadno povšimnou nejzásadnějšího problému, který se zde vyskytuje. Při porovnání řádků (1) a (3) s řádky (4) a (5) je vidět že jsou naprosto shodné i když se jedná o 2 nezávislé polygony. A proč tomu tak je? Důvod se skrývá v návaznosti 2 sousedních polygonů.
Jelikož pro vytvoření polygonu jsou potřeba 3 souřadnice a sousední polygony mají vždy 2 body shodné musí se tyto body „duplikovat" jako počáteční body navazujícího polygonu.

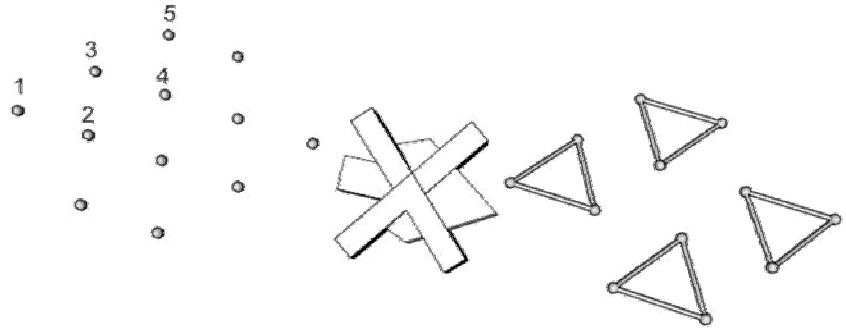


Obr.5.3 Ukázka 2 sousedních polygonů

Lépe to lze pochopit z obrázku. Při zadávání souřadnic prvého polygonu se začne v levém spodním rohu. Takže první řádek bude tvořit souřadnice bodu 1, jestliže se pak bude pokračovat proti směru hodinových ručiček, tak další jsou body 2, 3. Jelikož ale druhý polygon má 2 body shodné, tak ať už se začne kdekoli, tak při zachování stejného směru postupu, bude mít druhý polygon souřadnice 1, 3, 4 (tzn. že v tomto případě mají oba polygony stejné souřadnice bodů 1 a 3).

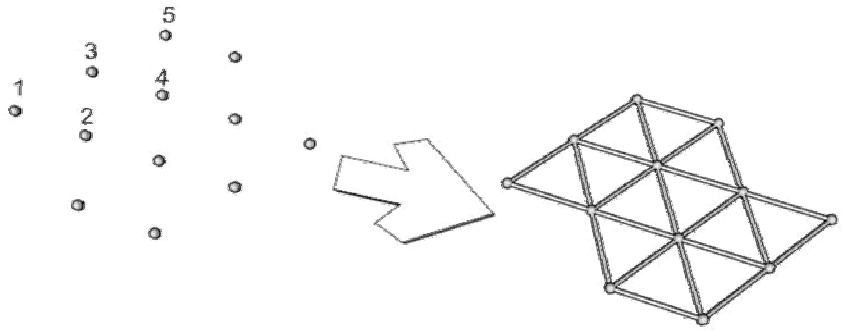
Pokud se tato úprava souřadnic neučiní, polygony se sice vykreslí, ale nebudou na sebe přímo navazovat.

Příklad takové nenávaznosti polygonů je na obrázku 5.4. Zde se při vykreslovaní začne pro vytvoření polygonu načítat vždy trojice bodů tzn. začne vykreslovat B1, B2, B3 čímž vytvoří první polygon, jelikož ale souřadnice nejsou duplikovány začne vykreslovat další polygon o souřadnicích B4, B5 a tím dochází k chybnému vykreslení a sousední polygony na sebe nebudou navazovat.



5.4 Chybně interpretované souřadnice polygonů.

Aby byli polygony dobře interpretovány je třeba je zadávat ve formě:
B1, B2, B3 další bude B3, B2, B4 pokud je tedy již zapsán jeden polygon, stačí vzít jeho 2 souřadnice, k nim přidat novou 3.souřadnici a tím se vytvoří polygon další. Ve výsledku tak stačí pro vykreslení 2 polygonu pouze souřadnice 4, nikoli 6 bodů.



5.5Správně vykreslené souřadnice

Jelikož při psaní této kapitole ještě není zcela specifikováno, v jakém tvaru budou prostorové souřadnice dodávány a samotná triangulace (Pomocí triangulaci lze vytvořit z množiny bodů pomocí různých matematických algoritmů a rovnic polygonovou síť tvořenou vždy z trojic bodů trojúhelníků) je velmi obsáhlé téma na další bakalářskou práci, proto výsledná aplikace zatím neobsahuje kód pro triangulaci. Pokud by ale později byla vyžadována, lze ji implementovat do modulu LoadDta, který se stará o zprávu a načítání souřadnic scény.

Zde *končí popis společné části implementace* obou aplikací. Další popis se zaměřuje na popis rozdílů v konstrukci aplikací používající standardní knihovny WIN32 a univerzální GLUT knihovny.

5.3 Knihovna Engine na bázi WIN32

Popis této knihovny bude spíše více o teorii, než o praktickém kódu, neboť většinou bývá lepší vědět jak a proč to tak funguje, než jenom jak to udělat.

Knihovna Engine je nejdůležitější a nejrozsáhlejší část celé aplikace. Obsahuje celou implementaci vytváření oken WIN32 a jádra OpenGL do aplikace.

5.3.1 Vytvoření okna

Vytvoření a správa okna aplikace je jeden z podstatných úkolů které musí aplikace zajišťovat. V tomto případě se využije pro vytvoření okna OpenGL rozhraní WIN32 API [13]. Win 32 ve spojení s DirectX nebo právě OpenGL se využívá k vývoji mnohých grafických programů pro operační systém Windows.

V první části aplikace se importují veškeré knihovny. Tento proces slouží k zpřístupnění všech funkcí dané knihovny. Mimo běžnějších knihoven jako windows.h , math.h, stdlib.h jsou připojeny i ryze OpenGl- gl.h,glut, glaux.h a vytvořené knihovny jako LoadDta.h.

Dále je potřeba definovat výchozí nastavení okna, ve kterém aplikace poběží. Toto nastavení se provádí ve funkci **InitGL()**. To probíhá zcela v režii funkcí z knihovny gl. Nastavuje se zde tak barva pozadí, typ použitého stínování, dále se volí použité textury a zapínají světla.

Následující funkce **DrawGLScene** má za úkol číst data z externího souboru k čemuž využívá knihovnu LoadDta. Načtená data jsou předána ze struktury funkce LoadDta do funkce DrawGlScene, kde dochází k jejich přepočítání na jednotlivé polygony. Tento cyklus probíhá stále opakovaně dokud není převed i poslední polygon - to je zajištěno for cyklem kde horní hranice je aktuální počet polygonů v souboru. Celý takto vytvořený seznam polygonů se posílá do hlavní funkce WinMain. Jelikož se scéna při pohybu neustále mění, je tento postup cyklicky opakován.

V další části programu je definována funkce **CreateGLWindow**, která bude zodpovědná za vytvoření WIN okna aplikace. Stará se také o to, v jakém režimu okno poběží (díky této funkci lze přepínat mezi okenním a fullscreen modem), nastaví jeho rozlišení, pozici okrajů při spuštění, typ použitého barevného stylu, barevnou hloubku obrazu, a vůbec vše co jsi s touto oblastí lze představit.

Když lze okno vytvořit musí být i způsob pro jeho řádné ukončení. Pro tyto případy je zde vytvořena funkce **KillGLWindow**. Ta je volána vždy před koncem programu i v případech výskytu nějaké nenadálé chyby. V této funkci jsou tak definovány příkazy pro ošetření možných eventualit, které mohou nastat při běhu programu. Pomocí těchto výjimek pak program ví, jak má na nečekanou situaci reagovat. Uvolní renderovací kontext, kontext zařízení a handle okna. Poté již dojde k řádnému uvolnění paměti.

Pokud celý proces proběhně bez problémů a nenastane žádná výjimka, dostane se na řadu funkce WinMain. **WINAPI WinMain** je poslední, ale zároveň i hlavní funkce

v programu. Tato funkce využívá služeb všech předešlých funkcí, zároveň obsluhuje veškeré zprávy pro okno. V prvé řadě vytvoří okno s uživatelským rozhraním, to provede zavoláním funkce CreateGLWindow, které předá parametry podle toho, jaký styl okna uživatel zvolí. Pak již probíhá vykreslování dat, a jsou přebírány události z fronty.

Události se do našeho okna standardně dostávají zachycením zpráv, které Windows posílá vždy konkrétnímu oknu aplikace, pro niž je určena. Aplikace na tuto zprávu pak reaguje přednastavenou funkcí tzv. procedurou okna. Uvnitř této funkce jsou zaregistrovány všechny zprávy, které nás zajímají a také to, jak na ně má reagovat. Ty stovky a tisíce zpráv, na které nechceme v programu nijak specificky reagovat, prostě necháme systému, ať se o ně postará. Tato smyčka probíhá cyklicky stále dokola, dokud není přijata zpráva pro ukončení aplikace. Pokud se tak stane, zalová se funkce **KillGLWindow** a okno je ukončeno.

Na tomto postupu je dobře vidět, že oknu jsou nejprve nastaveny jeho parametry (popisek, velikost, pozadí atd.), poté je vytvořeno pomocí, již zmiňované funkce **CreateGLWindow,** a až pak je teprve spuštěno v hlavní smyčce WinMain.

5.3.2 Použití textur

V předcházejícím popisu ještě nebyla zmíněna jedna důležitá část, bez které by na konci bylo zobrazeno pouze prázdné okno. Pro vykreslení viditelných povrchů těles, je potřeba přiřadit jednotlivým bodům a hranám nějakou barvu.

Technika, která to dělá se nazývá texturování, ta dvou-rozměrné obrázky (textury) obalí kolem tří- dimenzionálních objektů. O správné načtení textur se stará funkce:

LoadGLTextures(), která jednak kontroluje, zda vůbec zdrojový obrázek existuje, ale hlavně je zde definován název a umístění použité textury:

TextureImage[0]=LoadBMP("Data/Mater.bmp")

Pokud je tedy potřeba změnit název použité textury nebo její umístění, provede se to změnou parametru předané tomuto příkazu.

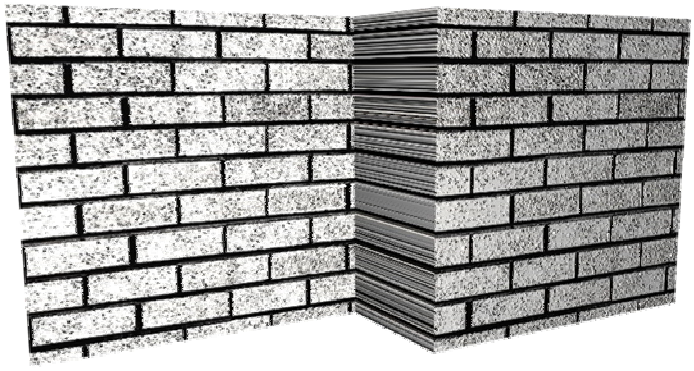
Aby bylo možné texturu nabalit na model, je ji třeba nejdříve tzv. namapovat. Mapování textury je proces, kdy každému bodu povrchu modelu je přiřazen bod na textuře.

Z počátku bylo mapování řešeno ručním zadáváním texturových souřadnic ke všem bodům scény. Pro vykreslení povrchu tělesa tak bylo potřeba 5 souřadnic: 3 prostorové souřadnice bodu + 2 souřadnice textůrové. Jelikož by toto řešení bylo v praxi velmi nepraktické, muselo se řešení hledat ve zcela jiné rovině.

Po prozkoumání možností OpenGL je pro mapování materiálů použito *automatické generování souřadnic* [O4] textury. Nejčastěji se tento způsob využívá v CAD systémech při zobrazování těles s velmi členitým povrchem. Pro účely automatického generování tak byla do této funkce zakomponován i příkaz:

```
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
```

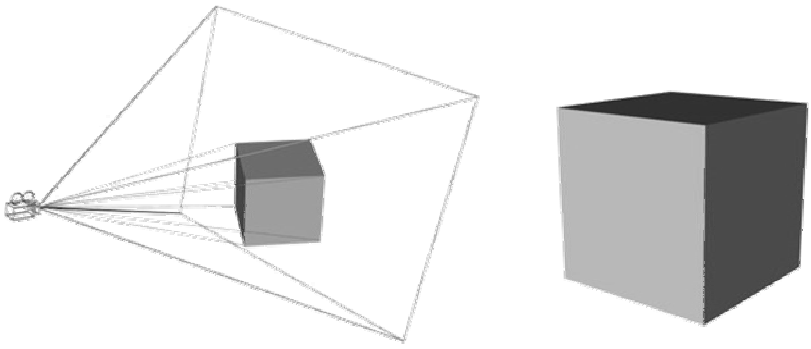
Zde jsou první dva parametry méně podstatné, pro ty co zajímají jejich význam odkazuji na dokumentaci kódu. Důležitý je až parametr 3, ten udává, jak se budou texturové souřadnice mapovat. Zde je použito lineární mapování což znamená, že textura je mapována na stěny v rovině X, Y a v Z se jakoby natáhne.



5.3.3 Kamera

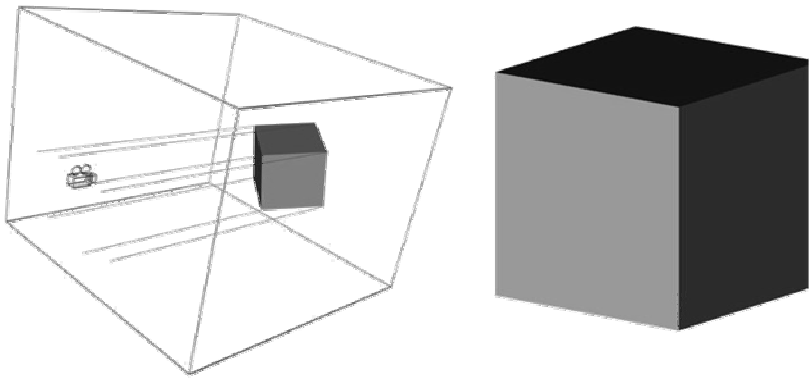
Aby se bylo jak ve scénu dívat je třeba si vytvořit alespoň jednu kameru.
Účelem kamery je definovat velikost 3D prostoru, který bude použit pro vykreslení na obrazovku a dále vyřadit z obrazu tu část, která je již mimo viditelnou scénu. Kvalita výsledného obrazu je odvozena od toho, jakou projekci kamery si zvolíme- OpenGL podporuje hned dvě, *perspektivní* a *ortogonální* [6].

a)Perspektivní projekce: Při tomto způsobu promítaní vycházejí všechny promítací paprsky z jednoho bodu - ten se nazývá střed promítání. Nejmarkantnější známkou této projekce je vznik deformací.Vzdálenost objektů od středu promítání přímo ovlivňuje jejich konečnou velikost průmětu, a proto čím je objekt vzdálenější od objektivu kamery, tím je ve výsledné scéně menší.



Obr. 5.7 Kužel perspektivní kamery + výsledný obraz tělesa

b)Ortogonální projekce: Zde jsou promítací paprsky rovnoběžné.
Ortogonální (někdy nazívané též rovnoběžné) promítaní se využívá převážně v technických CAD aplikacích, neboť je zde rozhodující zachovávat rovnoběžnost jednotlivých čar nebo úhlů mezi nimi. Vzdálenost objektu od počátku nijak neovlivňuje velikost výsledného průmětu.



Obr.5.8 Schéma zobrazovaného prostoru ortogonální kamery + výsledný obraz tělesa

V konečné realizaci kódů je použita **Perspektivní** projekce promítání jelikož je nejpodobnější lidskému vnímání prostoru.

Pohyb kamery ve scéně je řešen poněkud odlišným způsobem, než je tomu v reálném světě, zde se totiž nepohybuje pozorovatel, ale hned celá scéna. Při stisku klávesy vlevo/vpravo je celá scéna otočena okolo kamery v opačném směru než je rotace kamery.
Tento způsob byl zvolen kvůli jeho jednoduchosti a faktu, že k otáčení kamery pak postačí pouze jediný příkaz glRotatef(). Obdobně se postupuje i při pohybu vpřed a vzad. Tentokrát se celý svět posune za použití funkce glTranslatef(), v opačném směru něž je pohyb kamery.

5.3.5 Osvětlení a stíny

Stíny nám umožňují mnohem lepší představu o okolním prostoru a přinášejí zcela nové informace o vzájemné poloze a tvaru předmětů. Z tohoto důvodu byla do konečné aplikace zakomponována alespoň nejjednodušší schopnost simulace světla a stínu.

Pro výpočet osvětlení je použit tzv. *Phongův osvětlovací model*. Tento model má výpočet osvětlení co nejrychlejší, ale současně, ve výsledné scéně působí přirozeně.
Phongův osvětlovací model rozkládá počítané světlo do tří základních světelných složek: ambientní složky (*ambient light*), difúzní složky (*diffuse light*) a odlesků (*specular light*). Význam jednotlivých prvků je pro tuto práci nepodstatný, a proto ho nebudu dále rozebírat.

Co ale podstatné je, jak se do aplikace takové světlo přidá. Před použitím osvětlení ve scéně je potřeba ho globálně povolit. To se provede příkazem void glEnable(GL_LIGHTING).
Dalším krokem je definice barevných složek světla.

GLfloat LightAmbient[]= { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat LightDiffuse[]= { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[]= { 0.0f, 0.0f, 2.0f, 1.0f };

Posléze jako většina procesu, se použitá světla musí zapnou:

glEnable(GL_LIGHT&)

Parametr & pak určuje konkrétní světlo které chceme zapnout

Pokud jsou světla nastaveny přiřadí se jim i typ stínu který budou počítat. Pro potřeby aplikace zcela vyhovuje použití nejjednoduššího *konstantního stínování.* Tento postup počítá pro každý trojúhelník jednu barvu, a celá ploška je při vykreslování touto barvou vykreslena.
Pro nastavění stínování je použit příkaz:

glShadeModel(GL_SMOOTH);

5.4 Knihovna Engine na bázi GLUT

Tato druhá verze aplikace byla vytvořena až dodatečně z důvodu nesporných výhod které GLUT může nabídnout. Aplikaci tak zůstane zachována možnost pozdější přenositelnosti na jinou platformu plus další výhody, o kterých se zmíním v dalším textu.

Kromě již zmiňované multiplatformosti je další výhodou knihovny GLUT i její značná jednoduchost a celkový přístup oproti WIN32. Některé funkce jsou prováděny automaticky a většina z funkcí má dokonce přednastaveny přijatelné implicitní hodnoty, takže pokud to zcela není vyžadováno, lze se i některým zdlouhavým nastavením vyhnout a je možné i složitější program vytvořit snadněji než s WIN32. Pokud se přímo porovná délka kódu aplikace vytvořené ve WIN32 a GLUT, tak jasně vede GLUT s 360 řádky kódu oproti dvojnásobku 700 ve WIN32.

Jelikož je princip práce totožný s předešlým případek tak jen ve zkratce:

V aplikaci, které se chce používat knihovna GLUT, je zapotřebí nejprve tuto knihovnu inicializovat a teprve poté je možné používat její funkce.

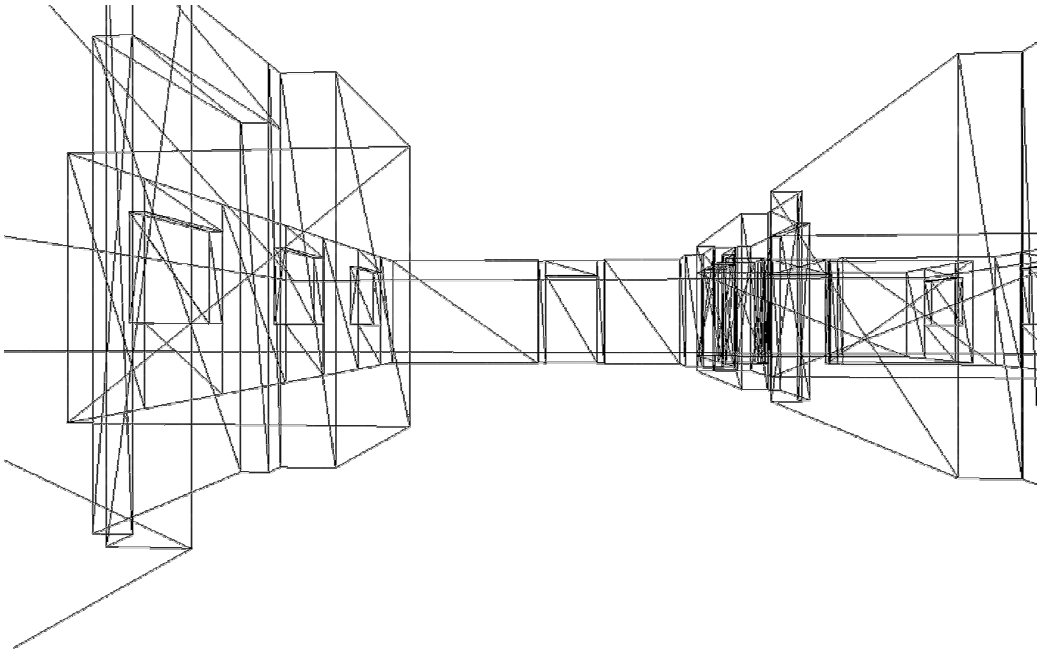
- **init()-** shodně jako u aplikace s WIN32, se i zde začíná inicializací vytvářeného okna (nastaví se velikost okna, barva pozadí, atd.)
- **render()-** obdoba DrawGLScene v předešlém případě
- **keyboard() a void special_keys()-**Ve většině aplikací je před spuštěním hlavní smyčky programu zaregistrovat callback funkce, které bude patřičně reagovat na zprávy došlé od systému a na vstupy od uživatele. Zde jsou definovány veškeré funkční klávesy a k nim je přiřazen kód který mají provádět.
- **main()-** Po registraci funkcí je možné spustit hlavní funkci. Jedná se opět o nekonečnou smyčku, ve které program získává zprávy od uživatele i od operačního systému a reaguje na ně voláním různých callback funkcí.

  Po inicializaci knihovny GLUT v hlavní smyčce, je již možné vytvořit okno, k čemuž se používá funkce glutCreateWindow(). Je zapotřebí si opět uvědomit, že vytvořené okno se nezobrazí ihned, ale až za běhu hlavní smyčky jako v předešlém případě.

Jak je vidět oba způsoby jsou si velmi podobné, ještě aby ne když mají oba za úkol stejnou věc.

5.4.1 Způsob vykreslování

Do aplikace byla zařazena i možnost zobrazit model ve třech módech: Plošně tzn. povrch potažený texturou, v drátěném modelu a pouze body. Protože každému procesu může lépe vyhovovat jiné zobrazení, může si uživatel vybrat pro něj nejpřijatelnějšího způsobu zobrazení.



Obr.5.9 program přepnutý v režimu hran (pozn.barvy obrázku jsou oproti skutečnosti invertovány pro lepší zobrazení hran v tomto textu)

6 Současný stav implementace

Konečná aplikace byla postavena na co nejjednodušší bázi za použití knihovny OpenGL a je schopná dynamicky vykreslovat libovolnou scénu uloženou v externím souboru. Avšak v porovnání s profesionálními programy obsahuje pouze základní funkce, které toto API poskytuje. I přes neustalé pokroky a vývoj jsem během vývoje narazil na problémy, které byly nad rámec mých současných časových možností a vyžadovaly by mnohem podrobnější studium těchto algoritmů, a ve výsledku by nebylo možné stihnout do doby odevzdání bakalářské práce dané algoritmy implementovat.

Dalším faktorem proč je konečná aplikace ochuzena o některé „specializované“ funkce je počet programátorů pracujících na projektu, protože na vývoji složitějších 3D aplikací pracuje vždy celí team.

6.1 Co je již hotovo

Ve výsledku se mi podařilo vytvořit aplikaci na základech OpenGL, která je schopna vykreslovat dynamicky se měnící scénu. Scéna je načítána z externího souboru, uživatel ji může kdykoli za běhu editovat a přidávat nové polygony. Jedinou omezující podmínkou je zadaní souřadnic pro všechny 3 body polygonu. Navíc se mi podařilo implementovat i různé režimi zobrazení a simulaci osvětlení čímž lze ještě zlepšit přehled o zobrazené okolní scéně.

6.2 Co je možné zlepšit

Prostoru k dalšímu pokračování vývoje je tedy mnoho. Zajímavým rozšířením by určitě bylo vytvoření modulu detekující kolize kamery s objekty ve scéně. V konečné aplikaci tento prvek sice chybí, jelikož ale v reálném prostředí při srážce stejně (většinou) snímač neprojde stěnou, nebudou polygony za zdí vykresleny, pohled zůstane před stěnou, a tím se tento nedostatek částečně kompenzuje.

Dalším dobrým rozšířením by byla již zmíněná triangulace polygonu. Díky jejímu použití by bylo možné ještě snadněji získávat souřadnice složitějších ploch. Stačila by pak jakékoliv počáteční povrchová data a aplikace by je automaticky transformovala do podoby trojúhelníků.

Určitě by bylo ku prospěchu aplikace i vytvoření knihovny pro ovládání kamery myší. Na této možnosti jsem již začal pracovat ale bohužel není v takové fázi aby šla do termínu odevzdání správně implementovat.

6.1 Závěr

Cílem této práce bylo seznámit se s aktuálním řešením 3D zobrazovacích aplikací, důkladněji se pak zaměřit na ty, které jsou založené na rozhraní OpenGL. V úvodu jsem tak porovnal toto oblíbené API s jeho nejzásadnějším konkurentem DirectX. Poukázal jsem na všechny jejich přednosti a nevýhody, které sebou může přinést jejich praktické použití.

Co se týká druhého bodu zadání, tedy praktické aplikace navrhnul jsem a implementovat realtime vykreslovací aplikaci na základech OpenGL, dokonce hned ve dvou verzích.

Pro obě verze je vytvořena jedna hlavní scéna, ve které je možné demonstrovat dynamické vykreslování podle toho, kolik si uživatel vloží polygonů. Snažil jsem se o takovou implementaci, aby její prezentace byla co nejjednodušší a nejefektivnější.
Navržené ovládání je co nejjednodušší a odpovídá standardu většiny dnes používaných aplikací.

Obsáhlým popisem API jádra OpenGL a funkční implementací si myslím že se mi podařilo splnil základní myšlenku zadání bakalářské práce.

Jelikož je však téma vytvoření realtime vykreslovací aplikace velice obsáhlé a daleko převyšuje možnosti jedné bakalářské práce, a vývoj softwaru nikdy nekončící proces, proto ani současný stav této práce nelze chápat jako uzavřenou kapitolu. Součástí kódu je tak popis všech procesů stěžejní pro další následovníky kteří by měli zájem o rozvoj této aplikace.

Pro mě samotného byla tato práce velmi přínosná. Prohloubil jsem své znalosti z oblasti programování a počítačové grafiky a dále:

- Prostudoval současné použití mnoha realtime zobrazovacích aplikací v praxi, to jak funguji, v čem vynikají a jak bych mohl využít její vzor k řešení svého problému.

- Provedl jsem důkladnou analýzu všech konstrukčních funkcionalit potřebný pro konečný vývoj aplikace. To ke konci obnášelo prostudovat a naučit se používat zcela nový jazyk C++ i zprávu WIN32 aplikací.

- Naučil jsem se alespoň základy jak zacházet s API OpenGL a DirectX. Ukázal jsem jejich stručnou historii, která je z velké části společná, základní principy práce obou API, vlastnosti a jejich hlavní klady a zápory.
  OpenGL API jsem také později použil v programové části práce.

- Připravil a z velké části i implementoval dynamický systém vykreslování prostředí. Při tomto vývoji jsem natvrdo poznal, jak je náročné vytvořit a hlavně odladit takovou grafickou aplikace, aby plně fungovala dle představ a to vše v jednom člověku (v teamu by vývoj postupoval určitě rychle a vyřešili by se i problémy popsané již dříve) s velmi rychle se blížícím termínem odevzdání.

7 Seznam použité literatury

[1] PRATA, S. *Mistrovství v C++:*. 3rd ed. Praha : Computer Press, 2007. ISBN 978-80-251-1749-1

[2] BRAŽINA, David. *Aplikace grafických informačních systémů.* Ostrava: Ostravská univerzita, 2003. 52 s. ISBN 80-7042-869-4.

[3] SHREINER, D.; WOO, M.; NEIDER, T. *OpenGL – Průvodce programátora*. Praha : Computer Press, 2006. ISBN 80-251-1275-6.

[4] HEROUT, P. *Učebnice jazyka C*. 2nd ed. České Budějovice : Kopp, 1993. ISBN 80-85828-02-2.

[5] RICHTER, J. *Windows pro pokročilé a experty*. Praha : Computer Press, 1997. ISBN 80-85896-89-3.

[6] VĚCHET, V. *Vybrané statě z počítačové grafiky (skriptum)*. 1st ed. Liberec: Technická univerzita v Liberci, 2006. ISBN 978-80-7372-178-7.

[7] ŽÁRA, J.; BENEŠ, B.; FELKEL, P. *Moderní počítačová grafika*. 2nd ed. Praha : Computer Press, 2005. ISBN 80-251-0454-0.

[8] *OpenGL.* [online].
URL: < http://opengl.navajo.cz/>

[9] FORMÁNEK,J. *DirectX* [online]. [cit. 2002-01-20].
Dostupné z: <http://casopis.programator.cz/r-art.php?clanek=54>

[10] TIŠNOVSKÝ,P. *Grafická knihovna OpenGL* [online]. [cit. 2004-02-24].
Dostupné z: <http://www.root.cz/clanky/graficka-knihovna-opengl-1/>.

[11] TIŠNOVSKÝ,P. *GLUT* [online]. [cit. 2004-05-27].
Dostupné z: <http://www.root.cz/clanky/glut-1/>.

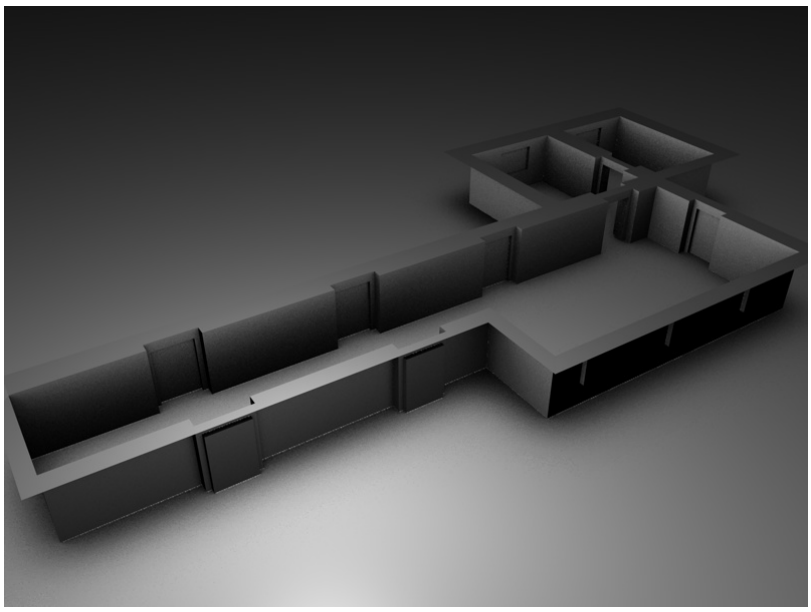
[12] MOLOFEE,J. *Setting Up An OpenGL Window* [online].
Dostupné z: <http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=01>

[13] SKÁLA,K. *Win32 API* [online]. [cit. 2006-01-01].
Dostupné z: <http://programujte.com/?akce=clanek&cl=2005122808-win32-api-uvod>

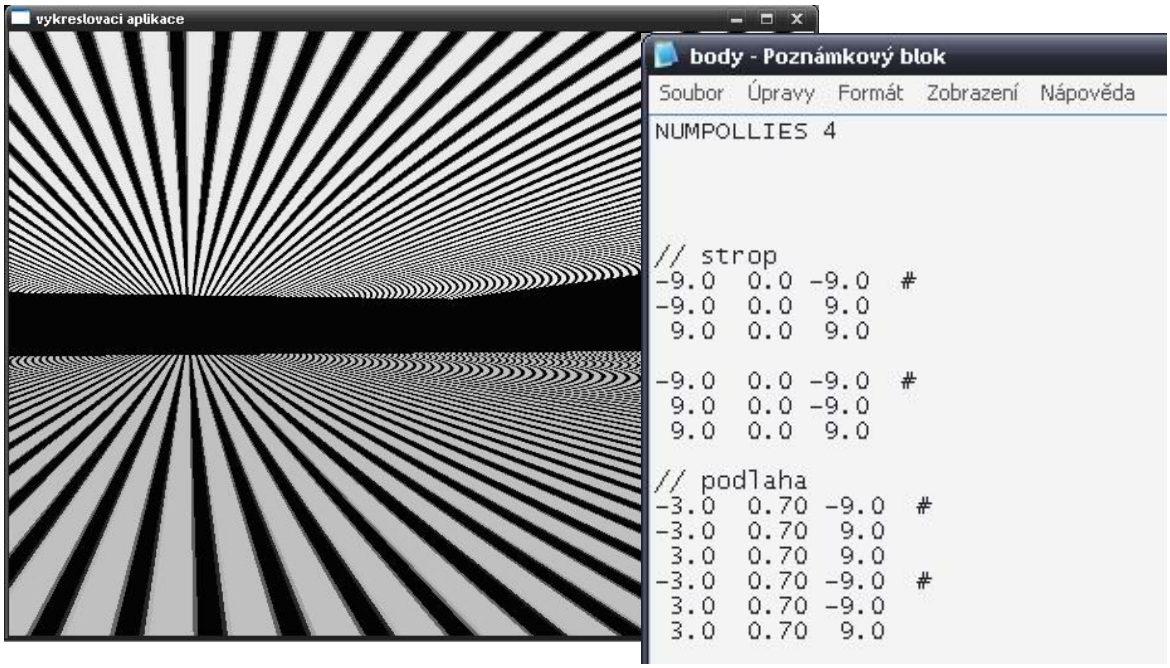
Přílohy

A) Obrázková galerie

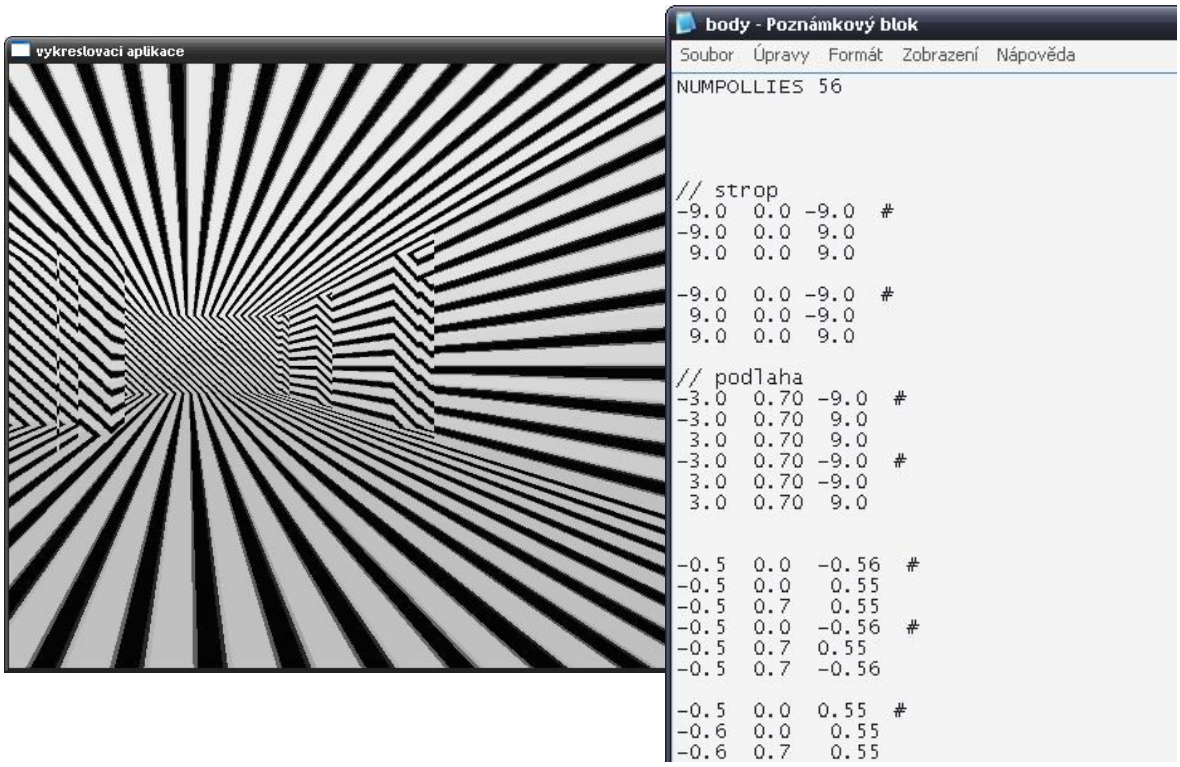
Tato příloha obsahuje obrázky z běhu programu.
Zdrojový soubory k níže uvedenému programu je součástí distribuce knihovny.



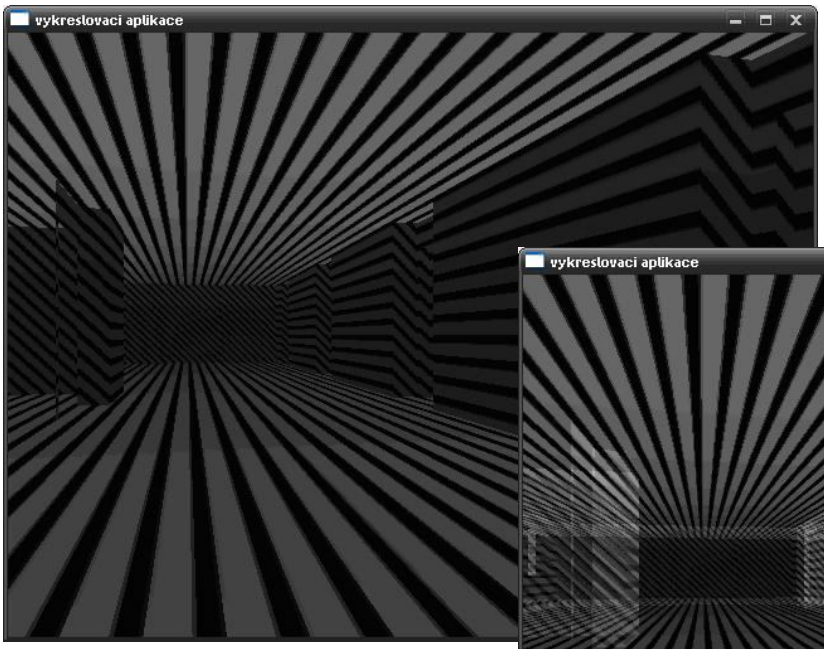
Obr.A1 Celkový pohled na ukázkovou scénu. Ta je složena z 294 polygonů a byla vytvořena v 3D modelačním programu Cinema 4D od firmy Maxon.Data scény potřebné pro aplikaci jsou uloženy v souboru Data/input.txt.



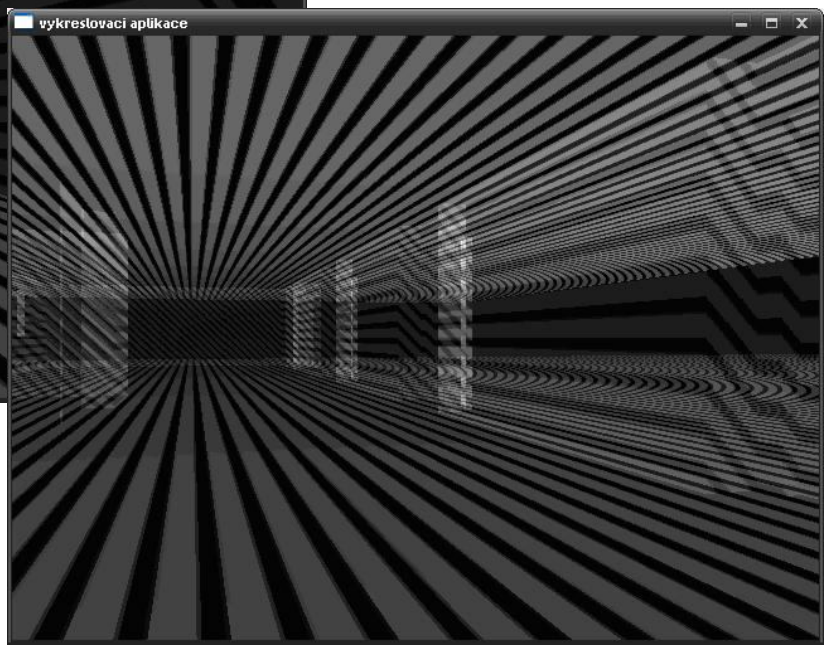
Obr.A2 Stav po spuštění aplikace-zatím jsou v souboru body.txt pouze polygony tvořící podlahu a strop scény.



Obr.A3 Do souboru je vložen libovolný počet polygonu (např. z ukázkové scény), po uloženi souboru je okamžitě aktualizovaný počet polygonu a dojde k jejich vykreslení v okně.



obr.A4 Odpovídá stavu na obr.3. Nyní však se zapnutým osvětlením (klávesa L).



obr.A5 Na posledním obrázku je případ, kdy je zapnuto světlo i s blendingem (klávesa B)

B) Tvorba vláken ve WIN

Můj popsaný způsob vytvoření vláken je funkční na platformě Windows, ale vlákna obdobně pracují i na systémech UNIX.

Pokud spustíme jakoukoliv aplikaci, systém vytvoří proces a vyhradí mu určitý adresní prostor a vytvoří vlákno, ve kterém pak bude probíhat zpracování kódu procesu. Každá spuštěná aplikace bez výjimky má tak alespoň jeden proces a ten má alespoň jedno vlákno, ve kterém probíhá výpočetní tok.

Vlákna [5] jsou vytvářena v rámci daného procesu a viditelné pouze uvnitř něj. Klasický proces je proces tvořený pouze jedním tzv. primárním vláknem. Toto primární vlákno se vždy vytváří zcela automaticky bez jakékoli nutnosti zásahu programátora. Toto primární vlákno je však schopné vytvořit i další vlákna - a to je to, co potřebuji. Pak již naše aplikace neběží pouze v jednom vlákně, ale lze procesy rozdělit do dvou samostatné pracujících částí. Výpočet pak probíhá tak, že systém přiděluje procesor střídavě mezi jednotlivé procesy, ve výsledku se tak zdá, že oba procesy probíhají současně. Takováto aplikace, kde uvnitř jednoho procesu může současně běžet několik vláken se pak nazývá tzv. multithreadovou.
Z popisu je evidentní se použití nemusí omezovat pouze na vytváření dvou vláken ale lze jich vytvořit libovolné množství.

Vytvoření vlákna

Př.

```
#include<iostream>
#include<windows.h>

DWORD WINAPI ThreadProc ( LPVOID lpParameter )
        {
Return 0;
        }
int main(){
DWORD dwThreadID;
HANDLE hThread = CreateThread(
0,                                  //security atributes
0,                                  //stack size
ThreadProc,          //thread start function name
0,                   // thread argument start function
0,                          //starting parametr
&dwThreadID);               //return thread ID
return 0;
              }
```

security atributes- nastavení úrovně zabezpečení objektu, pokud není potřeba jinak nastavuje se standardně na NULL

stack size- při vytvoření každé vlákno získá svůj adresový prostor (stack). Stack je automaticky uvolněn, když vlákno samo skonči, ale není uvolněn, když je ukončeno cizím vláknem. Velikost stacku nám tak říká, kolik paměti se má rezervovat pro provádění toku. Pokud použijeme hodnotu 0, tak funkce *CreateThread* použije při rezervování zásobníku hodnoty, které poskytne EXE soubor linkeru. Standardně je tato velikost 1MB

thread start function name- adresa, kde začíná kód, který bude vlákno vykonávat
např. jméno funkce

thread argument start function- jedná se o úplně stejný argument funkce použitý při její definici výše. Tento parametr jej tak předá prováděné funkci při jejím zavolání

starting parametr- dodatečné informace, potřebné při spouštění nového vlákna. Tento parametr nabývá pouze 2 hodnot
Pokud má hodnotu 0, vlákno se začne okamžitě vykonávat. Pokud mu ale zadáme hodnotu *CREATE_SUSPENDED* systém sice vytvoří vlákno, vytvoří jeho zásobní a veškeré jeho náležitosti avšak vlákno pozastaví, takže to se ihned nespustí

return thread ID- jedná se o adresu proměnné DWORD, do níž vytvářené vlákno uloží identifikační číslo nového prováděného procesu

C) Požadavky na spuštění a popis ovládání

Instalace
Instalace není zapotřebí, stačí celou aplikaci uložit na disk a spustit soubor START.

Aplikace byla testována na této sestavách:

Procesor: Intel® Core Duo T2300E
Paměť: 512MB
Grafická karta: NVIDIA GeForce GO 7300
Operační systém: Microsoft Windows XP servis pack 2

Procesor: AMD Athlon 62 3000+ Venice 5939
Paměť: 2047MB
Grafická karta: NVIDIA GeForce 8800GT
Operační systém: Microsoft Windows XP servis pack 3

Aplikaci by neměl být problém spustit i na nižší konfiguraci. Největší požadavky jsou na grafickou kartu a při zvyšující se velikosti vykreslované scény i na velikost operační paměti.

Ovládání:

Šipky: Pohyb

Page UP/DOWN: pohled nahoru/dolu

F1: přepnutí mezi fullscreen/oknem

L: Světla

B: Blending

E: režim hran (E z anglického edge)

P: režim bodů (point)

Escape: Konec aplikace

D) Seznam příloh v elektronické podobě

Součástí práce je CD-ROM, který obsahuje veškeré elektronické přílohy.

Příloha č.1: Bakalářská práce - text
Příloha č.2: Aplikace založená na OpenGL s využitím WIN32
Příloha č.3: Aplikace založená na OpenGL s využitím GLUT
Příloha č.4: Popis ovládání a vytváření polygonů v aplikaci