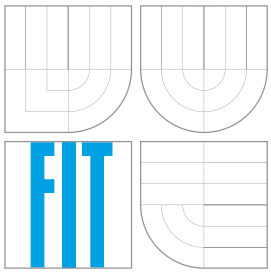
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MĚŘENÍ VÝKONNOSTI GRAFICKÉHO AKCELERÁTORU

PERFORMANCE EVALUATION OF GRAPHICS ACCELERATOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MILAN DVOŘÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADOVAN JOŠTH

BRNO 2010

Abstrakt

Tato práce se zabývá měřením výkonnosti grafických akcelerátorů. Popisuje vlastnosti současných grafických akcelerátorů a existující řešení měření jejich výkonu. Navrhuje vlastní metodiku pro měření výkonu a popisuje způsob implementace aplikace v OpenGL s využitím knihoven GLUT a GLEW. V poslední části se testuje několik grafických karet, zkoumají se faktory ovlivňující testování a diskutují se naměřené výsledky.

Abstract

This bachelor thesis deals with a performance evaluation of graphics accelerators. Properties of the current graphics accelerators and existing solutions for evaluating their performance are summarized. An own methodology for performance evaluation is proposed and the implementation of an OpenGL aplication with GLUT and GLEW libraries is described. In the last part, several graphic cards are tested, the influence of various factors is examined, and the measured results are discussed.

Klíčová slova

OpenGL, vertex, pixel, fragment, shader, benchmark, texturování, multitexturování, GLUT, GLEW, profilování, pipeline, 3DMark.

Keywords

OpenGL, vertex, pixel, fragment, shader, benchmark, texturing, multitexturing, GLUT, GLEW, profiling, pipeline, 3DMark.

Citace

Milan Dvořák: Měření výkonnosti grafického akcelerátoru, bakalářská práce, Brno, FIT VUT v Brně, 2010

Měření výkonnosti grafického akcelerátoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radovana Joštha.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Milan Dvořák
16. května 2010

Poděkování

Tímto bych chtěl poděkovat Ing. Radovanu Jošthovi za poskytnutou pomoc a konzultace při tvorbě této práce.

Obsah

Kapitola 1

Úvod

Grafické zobrazení dat je velice důležitým prvkem při komunikaci počítače s člověkem. Člověk je totiž schopen zrakem zpracovat mnohem větší množství informací než ostatními smysly. Proto prošla počítačová grafika v posledních letech bouřlivým vývojem. Na trhu je mnoho grafických akcelerátorů od různých výrobců s různými specifikacemi a každých několik měsíců se objevují další. Porovnávání kvality jednotlivých grafických akcelerátorů je komplexní a nejednoznačné. Existuje sice několik grafických benchmarků, ale ty bývají většinou komerční, rozsáhlé a fungují pouze pod novými operačními systémy Windows.

Cílem práce je vytvořit program, který bude měřit výkon grafických adapterů. Naměřené výsledky by měly být použitelné při srovnávání různých typů grafických karet.

První kapitola je teoretická a zabývá se různými technikami měření výkonnosti počítačů a jejich součástí. Dále pojednává o vlastnostech současných grafických akcelerátorů a o existujících řešeních měřících jejich výkonnost.

Druhá kapitola obsahuje návrh řešení. Identifikuje výkonově kritické operace grafických akcelerátorů a navrhuje měření výkonnosti těchto operací. Pojednává i o knihovnách a programovacích technikách, které budou využity při implementaci.

Další kapitola popisuje tvorbu aplikace, využívání knihoven, použité algoritmy a způsob implementace jednotlivých měření.

Předposlední kapitola diskutuje naměřené výsledky a jejich vypovídající hodnotu. Srovnává testované počítačové sestavy a vliv jejich parametrů na výsledky testů.

Závěrečná kapitola nastiňuje možnosti dalšího vývoje projektu a zhodnocuje celkový přínos práce.

Kapitola 2

Teorie

2.1 Profilování programů

Profilování je typ dynamické analýzy programů. Na rozdíl od statické analýzy zkoumá dynamická analýza chování programu během jeho provádění. Této analýzy lze využít při optimalizování programů, obvykle pro hledání kritických míst, kde program stráví nejvíce času. Tato místa můžeme hledat ručně, např. měřením stráveného času pomocí funkce `time()` v jazyce C. Další možností je specializovaný profilovací program, např. gprof nebo valgrind. Při psaní programu pro měření výkonnosti je žádoucí, aby tento program běžel co nejefektivněji, proto pravděpodobně některou techniku profilování využiji.

2.2 Měření výkonnosti

Měření výkonnosti počítačového hardwaru je stále velmi aktuální problém, protože podle jeho technické specifikace nelze vždy dobře určit skutečný výkon. Nemůžeme kupříkladu srovnávat různé typy procesorů na základě jejich pracovní frekvence nebo grafické karty podle velikosti jejich paměti. Proto vznikla celá řada testů (benchmarků) pro porovnávání různých počítačových komponent a architektur.

2.2.1 Způsoby měření

Při měření výkonnosti můžeme zjišťovat, jak dlouho trvá určitý úkon, nebo naopak kolik operací se provede za jednotku času. V prvním případě například zkoumáme, kolik času zabere výpočet několika řádů Ludolfova čísla, komprese souboru o určité velikosti, výpočet několika iterací rychlé Fourierovy transformace apod. U druhé možnosti se často měří počet operací s čísly s plovoucí desetinnou čárkou (FLOPS), počet provedených procesorových instrukcí (MIPS), počet vykreslených snímků (FPS) atd.

Jiný způsob dělení benchmarků je podle typu prováděného programu. Můžeme využít existující reálný program, nebo vytvořit umělý (syntetický) test. Výhodou reálných programů je, že měří přímo efektivitu provádění programů, které využívá při své práci uživatel. Příkladem těchto testů mohou být archivační programy, šifrování a dešifrování dat nebo kódování videa. Naproti tomu syntetické testy se mohou přímo zaměřit na často používané nebo výkonově kritické operace. Určení, které operace je vhodné testovat, závisí na konkrétní komponentě a účelu testování.

2.3 Vlastnosti současných grafických akcelerátorů

Základním úkolem grafického adaptéru je převést geometrický popis nějaké trojrozměrné scény na dvojrozměrný obraz. Tento proces je rozdělen na několik samostatných částí, takže může grafický adaptér fungovat jako proudový procesor. Tomuto způsobu zpracování se říká grafická pipeline. Dále se budu zabývat tím, jak grafickou pipeline definuje model OpenGL. Informace z následující podkapitoly jsem získal v [3].

2.3.1 Grafická pipeline

Vstupní data

Na vstupu grafické pipeline jsou geometrická a obrazová data. Geometrická data se skládají z bodů zadaných souřadnicemi v prostoru a typu primitiv, které popisují (bod, úsečka, polygon). Dále mohou obsahovat i další informace, jako třeba barvu, normálový vektor, souřadnice textury atd. Obrazová data jsou reprezentována barvou pixelu a případně hodnotou alfa, která se uplatňuje při různých technikách míchání barev. Tyto data mohou být čtena z paměti procesoru, z frame-bufferu nebo z paměti textur.

Operace s vrcholy

V této fázi jsou souřadnice vrcholů násobeny modelačními a pohledovými maticemi, jejich normálové vektory jsou násobeny inverzními maticemi. Dále jsou pro každý vrchol provedeny výpočty osvětlení na základě normály, materiálu a zdrojů světel. Těmito výpočty jsou vygenerovány nové barvy vrcholů. Poté jsou vrcholy transformovány projekční maticí z 3D modelovacího prostoru na 2D výstupní plátno. Po transformaci proběhne ořezání geometrických primitiv pohledovým objemem. Body, které leží mimo ořezávací roviny, jsou zahozeny. Pokud jsou ořezávány úsečky a polygony, jsou přidány nové body na průsečících s ořezovou rovinou. Nakonec probíhá proces rasterizace. Během tohoto procesu se určí, které pixely zobrazovaného okna jsou obsazeny daným primitivem. Vznikají tak fragmenty.

Operace s fragmenty

V této fázi je každému fragmentu přiřazena konečná barva. Probíhá několik testů, které mohou fragment vyřadit ze zobrazování – alfa test, stencil test, test hloubky. Při zapnutém texturování je vzorkováním vypočten jeden bod v textuře (texel), který je aplikován na barvu fragmentu. Dále může dojít k míchání barvy fragmentu s barvou již existujícího fragmentu. Na architekturách s malým počtem barev se provede dithering. Nakonec je fragment zkopírován do zobrazovací paměti (frame-bufferu) a stává se z něj pixel na obrazovce.

Implementace v HW

Fyzická pipeline je rozdělena stejně jako teoretická na dvě části, kde se vykonávají operace s vrcholy a operace s fragmenty. Poslední fázi renderování má na starosti jednotka ROP – render output unit, která kopíruje výsledný fragment do výstupního bufferu. Počet renderovacích jednotek tedy ovlivňuje teoretický objem pixelů, který je schopna grafická karta za určitý čas vykreslit.

2.3.2 Programovatelná pipeline

Grafická pipeline popsaná v předchozí podkapitole se nazývá fixní. Všechny funkce fixní pipeline jsou pevně dané, používají se předdefinované (hardcoded) operace. Přidávání nové funkcionality do tohoto modelu je náročné. Funkce se musí přidat do grafického API (OpenGL) jako rozšíření nebo jako součást nové revize. Krom toho se však musí nová vlastnost implementovat i v grafickém hardwaru, což může trvat poněkud delší dobu. Vývoj nových zobrazovacích technologií byl proto značně pomalý. OpenGL verze 2.0 tento problém řeší rozšířením pipeline o programovatelný přístup, kde můžeme část fixních funkcí nahradit vlastními programy zvanými shadery. Tyto programy se píší ve speciálním programovacím jazyce GLSL (OpenGL Shading Language). Rozlišujeme tři různé typy shaderů. V původní specifikaci byly vertex a fragment (pixel) shader, od verze OpenGL 3.2 se k nim přidal geometry shader.

Vertex shader

Vertex shader nahrazuje část operací prováděných s vrcholy ve fixní pipeline, konkrétně maticové transformace (modelační, pohledová, projekční), automatické generování souřadnic textur a výpočty osvětlení. Mění se tedy souřadnice vrcholu, jeho barva, normálový vektor atd. Ořezávání se ovšem stále provádí fixně, ve vertex shaderu nemůžeme vrcholy odebírat ani přidávat. Jinými slovy vstup i výstup vertex shaderu je jediný vrchol. Výstupní vrchol je pak dále zpracováván v geometry shaderu, pokud je přítomný.

Geometry shader

Vstupem geometry shaderu je celé grafické primitivum s vrcholy upravenými vertex shaderem. V této fázi se mohou některé vrcholy vyřadit ze zpracování, nebo se naopak mohou generovat nová grafická primitiva. Využití této funkčnosti je například při generování úrovně detailu podle vzdálenosti objektu nebo při teselaci polygonů.

Fragment shader

Fragment shader nahrazuje část operací prováděných s fragmenty ve fixní pipeline, konkrétně získávání texelů z textur a jejich aplikaci, kombinaci primární a sekundární barvy, aplikaci mlhy. Tento shader mění barvu fragmentu a případně obsah dalších bufferů (depth-buffer, stencil-buffer), může i vyřadit fragment ze zpracování. Lze zde aplikovat efekty jako průhlednost, stíny, odlesky, různé druhy nanášení textur (bump mapping).

Implementace v HW

Při programování sice rozlišujeme tři různé typy shaderů (vertex, fragment a geometry), ale v dnešních grafických akcelerátorech jsou všechny shadery sjednoceny. Od řady grafických karet Nvidia 8 (viz [4]) používají všechny shadery Unified shader architecture. To znamená, že shadery sdílí výpočetní jednotky. Ty se mohou přidělovat na základě aktuální vytíženosti jednotlivých částí. Ve starších grafických kartách byl pevný počet výpočetních jednotek pro vertex shadery a pro pixel shadery, jejich výpočetní výkon se tedy mohl lišit.

2.3.3 Texturování

Texturování je proces, kdy se na povrch 3D objektu nanáší 2D obraz zvaný textura. Objektu tak lze dodat větší detailnost zobrazení. Při mapování textury se přiřadí jednotlivým vrcholům objektu souřadnice textury, které se zadávají buď přímo při definici geometrického objektu, nebo se vypočítají automaticky ve vertex shaderu, případně v odpovídající části fixní pipeline. Poté se musí přiřadit každému pixelu odpovídající hodnota z textury. Jednotlivé texely textury málokdy přesně korespondují s jednotlivými pixely na obrazovce. Výsledná hodnota je tedy potřeba interpolovat. K ulehčení tohoto problému se může používat mipmapování. Při této technice je v paměti uložená jedna textura ve více velikostech, kdy každá další mipmapa má poloviční rozměry oproti předchozí. Pro interpolaci texelů se používá několik filtrů:

- bodové vzorkování – interpolace se neprovádí, použije se texel, jehož souřadnice leží nejblíže středu pixelu.
- bilineární vzorkování – hodnota se vypočítá lineární interpolací ze čtyř texelů, které leží nejblíže středu pixelu.
- trilineární vzorkování – hodnota se počítá jako lineární průměr dvou bilineárně vzorkovaných mipmap.

Každý filtr je výpočetně náročnější než ten předchozí, poskytuje ovšem lepší spojitost a vyhlazování obrazových dat.

Multitexturing

Multitexturování umožňuje najednou aplikovat několik textur na jeden polygon. V OpenGL je dostupné od verze 1.3. Jednotlivé textury se mohou při nanášení sčítat, průměrovat a různě kombinovat. Lze tak využívat například techniku bump mapping, která simuluje drobné nerovnosti povrchu.

Implementace v HW

Mapování textur provádí specializovaná jednotka, která se označuje jako TMU - texture mapping unit. V programovatelné pipeline může fragment shader nahradit některé výpočty prováděné touto jednotkou, ta je však stále zodpovědná za dodávání texturových dat do shaderu. Počet texturovacích jednotek tedy ovlivňuje maximální teoretický objem zpracovávaných texturových dat.

2.4 Existující řešení měření výkonnosti grafických akcelerátorů

Srovnávání výkonnosti různých grafických akcelerátorů je dnes populární záležitost. Existuje několik nástrojů, kterými se dá výkon měřit. Našel jsem dvě komerční řešení, *Aquamark* a řada grafických benchmarků od firmy *Futuremark* (dříve MadOnion). Dále existuje celá řada volně dostupných menších projektů, které pravděpodobně vznikly jen z vlastní iniciativy autorů. Patří mezi ně například *GLMark*, jenž je narozdíl od obou zmíněných komerčních řešení naprogramován v OpenGL a je šiřitelný ve formě zdrojových souborů.

Oblíbený způsob, jak měřit grafický výkon počítačové sestavy, je pomocí počítačových her. Mnoho herních titulů má výkonnostní test zabudovaný přímo v sobě, jako příklad uvedu World in Conflict. Jiné hry nabízejí volně dostupný benchmark, aby si uživatel mohl vyzkoušet, zda mu hra poběží plynule. Tuto službu poskytují např. hry Far Cry 2 nebo The Last Remnant. Nemusíme se však spoléhat pouze na speciální řešení, výkon se dá měřit i ručně. Pro měření počtu vykreslených snímků ve hře lze použít nástroje jako *Fraps*. Pokud to hra umožňuje, je vhodné nahrát část jejího běhu (např. timedemo v Half Life 2) a poté měřit výkon pouze na tomto úseku. Tím docílíme toho, že je vždy vykreslována stejná scéna.

Nyní rozeberu výše zmíněné komerční benchmarky.

2.4.1 Aquamark

Aquamark je benchmark pro grafické karty od firmy Masive. Verze 3 tohoto programu vyšla v roce 2003. Poté byl projekt pravděpodobně ukončen, protože jeho internetová doména aquamark3.com patří již jinému majiteli. Informace o Aquamarku jsem nalezl v různých internetových článcích (např. [11]), nebo v dokumentaci [10], která je přiložená k volně stáhnutelné verzi programu.

Všechny obsažené testy běží na grafickém enginu Krass, který je použit ve hrách AquaNox i AquaNox 2. Engine využívá možnosti grafického API DirectX 9, ale je zpětně kompatibilní i s verzemi DirectX 8 a 7. Jak již napovídá použitý engine, Aquamark nepoužívá syntetické testy. Vykreslovány jsou komplexní scény, které jsou podobné skutečným počítačovým hrám. Přesto je každý test zaměřený na určité grafické výpočty, např. počítání osvětlení, mlhy, operace s velkým množstvím polygonů, přetěžování pixel shaderů apod.

Dále Aquamark nabízí speciální funkce, které jsou určeny pro náročné uživatele a nadšence. První z nich po nastavených intervalech automaticky ukládá vykreslené obrázky, což je užitečné při porovnávání kvality obrazu. Další technika graficky znázorňuje, kolik výpočetních operací bylo s daným pixelem provedeno. Pro zobrazení byly využity podobné barvy jako pro teplotní diagramy, kde modrá představuje nízkou teplotu – malý počet operací a červená vysokou teplotu – velký počet operací. Na obrázku 2.1 vidíme praktickou ukázku této techniky. Červeně zobrazený je kouř, který je simulovaný velkým počtem malých částic a potřebuje tedy velký počet výpočetních operací. Podobně si můžeme nechat barevně odlišit, jaká verze pixel shaderu (verze DirectX) byla pro daný pixel použita.

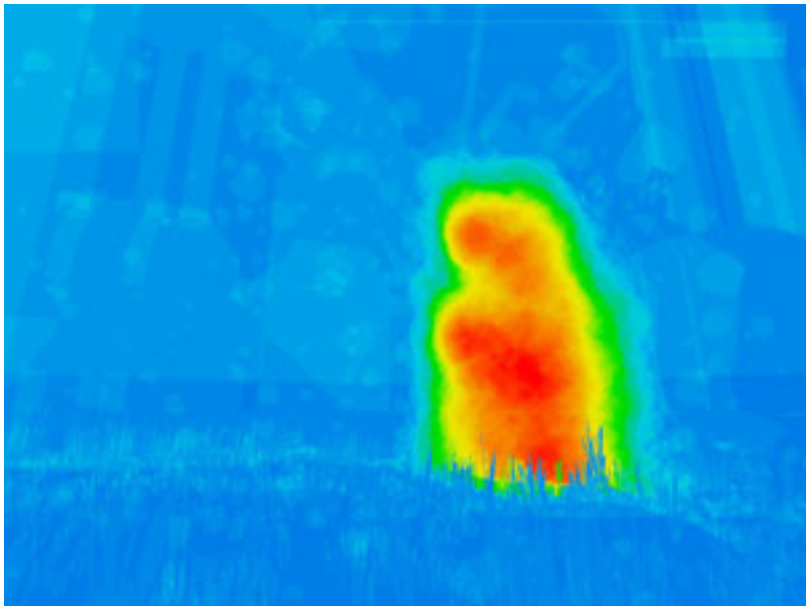
Aquamark byl dle mého názoru slibný a zajímavý grafický benchmark, který mohl konkurovat s řešením od firmy Futuremark. Bohužel se projekt přestal dále vyvíjet, pravděpodobně kvůli malé podpoře uživatelů.

2.4.2 Futuremark a 3DMark

Firma Futuremark byla založena v roce 1997 ve Finsku a zabývala se především programováním 3D grafických aplikací. Mezi významné produkty firmy dnes patří 3DMark a PCMark. PCMark testuje výkon celého počítače, konkrétně procesor, operační paměť, pevný disk a zčásti i grafický akcelerátor. Naproti tomu 3DMark se zabývá pouze grafickým akcelerátorem, proto ho v následující části detailněji rozeberu.

Vývoj verzí

Verze 3DMarku jsou číslovány podle kalendářního roku, kdy měla daná verze působit na trhu. Program vycházel vždy na začátku tohoto roku, nebo s několikaměsíčním předstihem.



Obrázek 2.1: Počet operací s pixelem v Aquamarku (obrázek z [11])

První verze 3DMark99 používala DirectX 6. Dalších několik vydání 3DMarku koresponduje s aktualizacemi rozhraní DirectX. 3DMark2000 běží nad DirectX7, 3DMark2001 používá DirectX8 a 3DMark03 již potřebuje DirectX9. Ve všech těchto benchmarcích se však vyskytují testy, které poběží i bez poslední verze DirectX. To se změnilo v následujících dvou verzích 3DMark05 a 3DMark06. Tyto vyžadují pro svůj běh grafickou kartu s plnou podporou DirectX9. Navíc přidávají pokročilé efekty díky novým verzím pixel shaderů.

Poslední 3DMark porušuje číslování podle roků. Je pojmenovaný Vantage. Oproti svým předchůdcům je velmi odlišný. Požaduje DirectX10 a operační systém Windows Vista nebo Windows 7. Změnil se i licenční systém a volně dostupná verze je silně omezená co se funkčnosti týče.

Struktura testů

Ve všech generacích benchmarků 3DMark je podobná struktura testů. Hlavní jádro vždy tvoří několik *Game* testů. Ty mají za úkol simulovat chování současných počítačových her. Každý test je pak zaměřen na určitý typ hry (first-person shooter, letecký simulátor apod.) a odpovídající vlastnosti grafické karty. Další skupinou jsou syntetické testy, které měří konkrétní vlastnosti grafického akcelerátoru. V každé verzi je *Fill rate* test, jenž měří propustnost texturovacích jednotek v počtu texelů za sekundu. Ve starších verzích se používaly testy rychlosti mapování a renderování textur a výpočtů s velkým počtem vrcholů. Pozdější verze přinesly testy shaderů. U vertex shaderů se zjišťuje počet zpracovaných vrcholů za sekundu, u pixel shaderů se měří jen počet vykreslených snímků za sekundu. Poslední skupinou testů jsou takzvané *Feature* testy. Jak již název napovídá, zkoumá se výkon při použití nejnovějších technologií v dané verzi DirectX. Tyto testy jsou pochopitelně v každé verzi velmi odlišné.

Kapitola 3

Návrh testování

3.1 Výkonově kritické operace

3.1.1 Programovatelné shadery

V dnešních grafických aplikacích a počítačových hrách jsou hojně využívány programovatelné shadery. Většina nově představených technologií s nimi souvisí nebo je na nich přímo závislá. Důležité efekty jako osvětlování, stínování, zrcadlení, pohyb objektů nebo vlastnosti povrchů jsou počítány právě v programovatelných shaderech. Ve většině aplikací tedy platí, že výpočetní výkon programovatelných shaderů představuje úzké hrdlo a z velké části určuje celkový grafický výkon počítače. V grafickém benchmarku by proto neměl chybět test zaměřený právě na programovatelné shadery.

3.1.2 Texturování

Další důležitou technikou je texturování. Nejedná se však jen o pouhé nanášení obrazových dat na povrch objektu, jak tomu bylo v dřívějších dobách. Kromě klasických textur se dnes používá celá řada speciálních map, např. normálové mapy, spekulární, difúzní, kubické, odrazové, lomové a další. Tyto textury se pak pomocí multitexturování nanáší na objekt (více viz 2.3.3). Výpočetní operace při kombinování těchto textur se obvykle provádí v programovatelných shaderech, texturovací jednotky ovšem musí stíhat dodávat shaderům texturovací data. Proto může kvalita a počet texturovacích jednotek ovlivnit celkový výkon.

3.2 Návrh testování

3.2.1 Fill rate

Nejdříve implementuji klasické testy propustností renderovacích a texturovacích jednotek. Budu měřit počet pixelů přenesených na obrazovku u renderovacích jednotek a počet přenesených texelů u texturovacích jednotek. Díky tomu nebude výsledek měření závislý na rozlišení obrazovky, na rozdíl od počítání vykreslených snímků za sekundu. Je možné, že při jednoduchém texturování bude maximální naměřená hodnota ovlivněna i renderovacími jednotkami. Proto u jednoho texel fill rate testu využiji techniku multitexturování a snížím tak zatížení renderovacích jednotek.

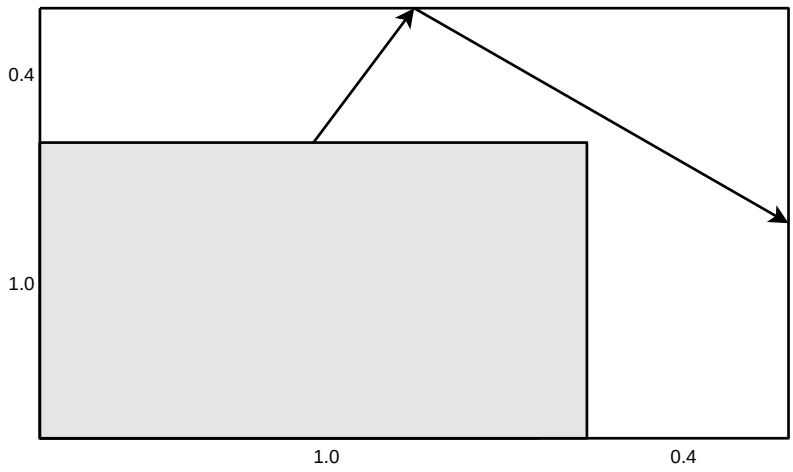
Pixel rate

U testování propustnosti vykreslovacích jednotek budu potřebovat při každém překreslení obrazovky přistoupit ke všem pixelům a změnit jejich hodnotu. Nejjednodušší způsob, jak toho dosáhnout, je zobrazit přes celou obrazovku jeden čtyřúhelník a po každém vykreslení měnit jeho barvu. Dá se očekávat vysoký počet snímků za sekundu a kdybych měnil barvu od bílé po černou nebo přes jiné kontrastní barvy, obrazovka by nepříjemně blikala. Proto zvolím pouze několik barev v odstínech šedé, abych zabránil rychlému blikání monitoru.

Texel rate

V tomto testu budu na každý pixel obrazovky nanášet data z textury. Obsah textury nelze měnit stejně jednoduše jako barvu pixelu, proto nemohu využít princip z pixel rate testu. Místo toho nanesu texturovací data na čtyřúhelník, který přesahuje hranice obrazovky. Na základě testování jsem zvolil čtyřúhelník 1.4 krát větší než rozměry obrazovky. Po každém překreslení scény budu pohybovat s pohledovým oknem po tomto čtyřúhelníku. Na začátku testu je pohledové okno na souřadnicích [0.0, 0.0] (pravý spodní roh). S oknem budu posunovat směrem nahoru a doprava až do chvíle, kdy narazím na horní okraj čtyřúhelníku. V tomto bodě změním směr posunu dolů. Podobně budu měnit směr pohybu doleva na pravém okraji čtyřúhelníku. Pohledové okno se takto bude „odrážet“ od okrajů čtyřúhelníka podobně jako spořiče obrazovek v systému Microsoft Windows.

Situaci znázorňuje obrázek 3.1. Šedě vybarvený obdélník představuje obrazovku (pohledové okno) na začátku testu, tedy v pravém dolním rohu. Šipky ukazují směr pohybu pohledového okna a jeho změnu po dosažení horní hranice bílého čtyřúhelníku. Čísla u hran obou čtyřúhelníků vyjadřují jejich poměr velikostí.



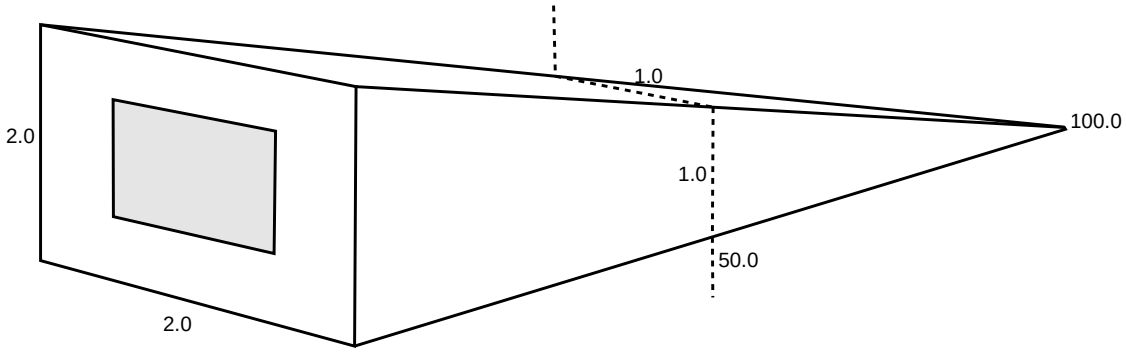
Obrázek 3.1: Posun zobrazované části scény

3.2.2 Mapování textur

V tomto testu se zaměřím na rychlost mapování textur, jejich filtrování a renderování. Test se spustí několikrát, přičemž se při každé iteraci zvětší velikost nanášené textury. Pro zachování kompatibility se staršími grafickými kartami budu používat texturu s rozměry rovnými mocnině dvou (libovolné rozměry textur jsou podporovány až od OpenGL 2.0).

Textura bude také čtvercová, aby se dala snáze algoritmicky zvětšovat na požadovaný rozměr. Nemohu používat techniku mipmapování, protože by se používala menší textura a její zvětšování by tak nemělo smysl. Z interpolačních filtrů vyberu bilineární vzorkování (viz 2.3.3). Pro měření výkonnosti nemám jinou možnost než zvolit počítání vykreslených snímků za sekundu i za cenu toho, že výsledky budou závislé na použitém rozlišení.

Textury budu nanášet na objekt jehlanu, kterým budu simulovat průlet tunelem. Jehlan bude mít základnu v rovině obrazovky a vrchol v souřadnici 100.0 na ose z. Textury nanesu na stěny jehlanu a celým objektem budu posunovat po ose z ve směru k pozorovateli. Rozměry podstavy jehlanu budou 2 krát větší než pohledové okno. Při posunování objektu směrem k pozorovateli pak přesně v polovině (50.0 na ose z) přesáhne pohledové okno rozměry jehlanu. V tomto bodě budu muset test ukončit. Šedý čtyřúhelník na obrázku 3.2 představuje pozici pohledového okna na začátku testu, přerušovaná čára jeho polohu při ukončení testu.



Obrázek 3.2: Schéma jehlanu pro nanášení textur

Návod, jak používat textury v OpenGL i s ukázkovým příkladem, je na [1]. Nejdříve pomocí funkce `glGenTextures` získáme volný identifikátor textury, tuto texturu pak označíme pro další práci funkcí `glBindTexture`. Poté nastavíme způsob nanášení textury funkcí `glTexEnvf`, na výběr máme např. `GL_REPLACE` nebo `GL_MODULATE`. Další parametry textury jako filtrování a opakování textury mimo její rozsah nastavuje funkce `glTexParameterf`. Podrobnosti o nastavování textur lze nalézt v [3]. Vlastní obrazová data pak do textury nahrajeme příkazem `glTexImage2D`. Při samotném kreslení objektů pak můžeme každému vrcholu přiřadit souřadnice textury pomocí funkce `glTexCoord3d`. Na daný bod (polygon) se pak mapuje textura vybraná funkcí `glBindTexture`.

3.2.3 Programovatelné shadery

Nakonec budu zkoumat výpočetní výkon programovatelných shaderů. V dnešních grafických kartách se sice všechny shadery provádějí na společných výpočetních jednotkách (viz 2.3.2), přesto budu zvlášť testovat vertex a pixel shadery (geometry shadery budu ignorovat). Naměřené výsledky by se mohly lišit. Navíc se stále používají grafické akcelerátory, které mají jednotky pro pixel a vertex shadery oddělené. Na těchto kartách pak obvykle mají pixel shadery výkonově navrch. Výkon shaderů budu měřit v počtu operací (sčítání a násobení) s čísly s plovoucí desetinnou čárkou za sekundu, očekávané výsledky jsou v jednotkách GFLOPS. Tyto výsledky jsou nezávislé na rozlišení obrazovky a měly by být přímo srovnávatelné pro různé grafické akcelerátory.

Fragment shader

Při programování testu na fragment (pixel) shader využiji podobného principu jako u testu pixel rate (viz 3.2.1), tzn. velký jednobarevný čtyřúhelník přes celou obrazovku. Po každém vykreslení změním barvu čtyřúhelníku, ta však bude ještě modifikována zátěžovými výpočty ve fragment shaderu. Pro každý pixel se bude muset vykonat několik set až tisíc operací s čísly s plovoucí desetinnou čárkou, protože dnešní grafické karty disponují výkonem i přes jeden TFLOPS.

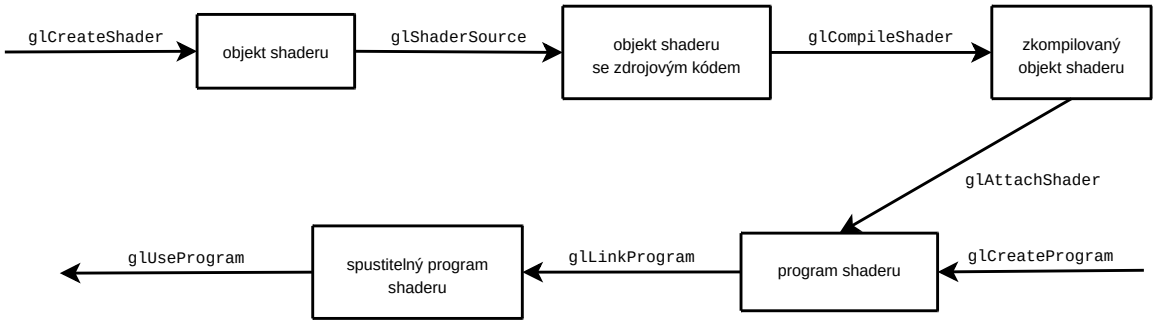
Vertex shader

Ve vertex shaderu budu modifikovat barvu vrcholu stejným způsobem jako u fragment shaderu, aby byly naměřené výsledky porovnávatelné. Navíc však ve vertex shaderu musí být určena pozice vrcholu. Dále budu muset změnit vykreslovanou scénu, aby v ní bylo srovnatelné množství vertexů jako pixelů. Na každý pixel na obrazovce tedy umístím jeden bod. Při výpočtech pozicí bodu však mohou vzniknout chyby, proto přes celou obrazovku zobrazím ještě čtyřúhelník. Ve scéně tedy bude o 4 body více než je počet pixelů na obrazovce.

Programování shaderů v OpenGL

Shadery se v OpenGL programují pomocí jazyka GLSL. Zdrojový kód v tomto jazyce se musí v OpenGL aplikaci zkompilovat a slinkovat do programu shaderu. Ten se pak teprve může využít pro zpracování vrcholů a fragmentů. Detailní popis, jak používat shadery v OpenGL, je v [3] nebo v [2]. V následujícím odstavci stručně shrnu postup vytváření shaderů.

Nejdříve vytvoříme objekt shaderu pomocí funkce `glCreateShader`, které předáme jako parametr konstantu pro typ shaderu (`GL_VERTEX_SHADER` nebo `GL_FRAGMENT_SHADER`). Dále musíme do tohoto objektu vložit zdrojový kód shaderu. K tomu slouží `glShaderSource`. Zdrojový kód předáme funkci jako pole znakových řetězců (datový typ `char**`). Objekt shaderu pak zkompilujeme funkcí `glCompileShader`. Po vytvoření všech vertex a fragment shaderů je musíme propojit do programu shaderu. Program vytvoříme funkcí `glCreateProgram`. Jednotlivé objekty shaderů do něj vložíme pomocí funkce `glAttachShader`. Nakonec celý program slinkujeme funkcí `glLinkProgram`. Nyní je program shaderu připravený k používání. Zpracovávat vrcholy a fragmenty však začne až po zavolání funkce `glUseProgram`. Celý tento postup je znázorněn na obrázku 3.3.



Obrázek 3.3: Vytváření shaderu (podle [3])

3.3 Návrh implementace

3.3.1 Grafické API a knihovny

Pro implementaci navržených testů jsem si vybral grafické API OpenGL. Velkou předností tohoto rozhraní je, že není závislé na operačním systému. Většina současných grafických benchmarků je určena pouze pro platformu Microsoft Windows. Pro operační systém Linux žádné komplexní řešení měření výkonu grafického akcelerátoru neexistuje, dostupných je jen několik menších projektů (např. GLMark). Budu se proto snažit, aby byl můj program multiplatformní a běžel alespoň na operačních systémech MS Windows a Linux.

GLUT

Prvním problémem, který musím řešit při naplňování výše uvedeného cíle, je vytváření oken a manipulace s nimi. Samotné OpenGL neobsahuje žádné funkce pro práci s okny, aby byla dodržena nezávislost na operačním systému. Budu tedy muset použít některé rozšíření OpenGL pro zpřístupnění systému oken. Jako vhodné řešení pro mou aplikaci se mi jevila knihovna GLUT. Poskytuje sice jen základní funkčnost a není určena pro rozsáhlé OpenGL aplikace, pro mé potřeby však plně dostačuje.

GLUT je zkratka pro OpenGL Utility Toolkit. Tato knihovna kromě funkcí pro správu oken přidává i podporu zpracování událostí, vstupu z klávesnice a myši, kreslení 3D objektů a další užitečná rozšíření.

V programu se GLUT nejdříve inicializuje funkcí `glutInit`, které bychom měli předat parametry příkazové řádky. Dále nastavíme funkcí `glutInitDisplayMode` zobrazovací mód, např. `GLUT_DOUBLE` nastaví používání dvojitého framebufferu. Po inicializaci můžeme funkcí `glutCreateWindow` vytvořit okno. V mé aplikaci však raději použiji `glutEnterGameMode`, což je optimalizovaný fullscreen mód, který navíc umožňuje změnit rozlišení. Parametry vytvořeného okna se nastavují funkcí `glutGameModeString`, např. rozlišení 800 na 600 a 32 bitovou barevnou hloubku nastavíme řetězcem „800x600:32“. Po vytvoření okna musíme nastavit funkce zpětného volání. Několik příkazů pro registrování těchto funkcí je v následujícím seznamu (viz [3]):

- `glutDisplayFunc` – překresluje obsah okna
- `glutReshapeFunc` – volá se při změně velikosti nebo posunutí okna
- `glutKeyboardFunc` – zpracovává vstup z klávesnice
- `glutMouseFunc` – zpracovává vstup z myši
- `glutIdleFunc` – volá se při nečinnosti programu, když nejsou žádné jiné události. Využívá se pro řízení procesu na pozadí.

Nakonec pomocí příkazu `glutMainLoop` uvedeme program do nekonečné smyčky, kdy se zpracovávají události.

GLEW

Další nepříjemností při vývoji OpenGL aplikace je fakt, že systém Microsoft Windows nativně podporuje pouze OpenGL verze 1.1 (viz [5]). Ve své aplikaci budu používat i některé pokročilé funkce (multitexturování, programovatelné shadery), které jsou dostupné pouze ve vyšších verzích OpenGL. Pro jejich zpřístupnění budu muset použít některou přídavnou

knihovnu. Vybral jsem si knihovnu GLEW (OpenGL Extension Wrangler Library). Ta zpřístupňuje funkčnost až do verze OpenGL 4.0 a byla testována na operačních systémech Windows, Linux, Mac OS X, FreeBSD, Irix a Solaris (viz [6]).

Pro používání knihovny v aplikaci stačí zavolat funkci `glewInit`, která knihovnu inicializuje. Ověření, jakou verzi OpenGL máme k dispozici, provedeme testováním definovaných konstant. Např. pro kontrolu dostupnosti verze 2.0 použijeme konstantu `GLEW_VERSION_2_0`. Podobně můžeme ověřovat přímo jednotlivá rozšíření, třeba podporu multitexturování reprezentuje konstanta `GLEW_ARB_multitexture`.

3.3.2 Optimalizace

Při tvorbě programů, které mají měřit výkonové vlastnosti počítače, hraje důležitou roli i efektivnost a optimalizace. V grafickém benchmarku se například musí vypnout vertikální synchronizace. Pokud je totiž zapnutá, synchronizují se vykreslené snímky s frekvencí monitoru. Dnešní monitory obvykle fungují na frekvencích od 60 do 100 Hz. S výkonnými grafickými akcelerátory je pravděpodobné, že se dosáhne při testech více než 100 snímků za sekundu. Zapnutá vertikální synchronizace by tedy omezovala naměřený výkon.

Dále je potřeba omezit používání příkazu `glClear`. Tento příkaz vymaže dané buffery, neboli nastaví všechny prvky bufferu na jejich základní hodnotu (vynulování hloubkového bufferu, nastavení všech pixelů frame bufferu na černou barvu apod.). Tyto buffery jsou obvykle velmi rozsáhlé, protože korespondují s rozlišením obrazovky. Např. pro rozlišení 1280x1024 má každý buffer více než milión bodů. Operace mazání sice bývají dobře optimalizované, přesto však mohou trvat nezanedbatelný čas a jejich přílišné používání také snižuje naměřený výkon (viz [3]).

Display list

OpenGL obsahuje techniku pro optimalizaci opakovaného provádění série příkazů. Příkazy uložíme do zobrazovacího seznamu (display list), kde jsou pak připraveny k opětovnému provedení. Zvýšení výkonu můžeme dosáhnout, pokud jsou v zobrazovacím seznamu např. složité maticové operace, nastavování materiálů nebo pokud složitě počítáme souřadnice objektů nebo textur. Tyto výpočty jsou provedeny pouze při zadávání a do seznamu se uloží jen jejich výsledky. Používáme-li ovšem příliš krátké a jednoduché zobrazovací seznamy, může se projevit zpoždění spojené se skoky do těchto seznamů.

Každý zobrazovací seznam je odkazován svým celočíselným indexem. Při vytváření zobrazovacího seznamu vygenerujeme volný index funkcí `glGenLists`. Funkce očekává jako svůj parametr kladné číslo, které určuje, kolik volných indexů chceme vygenerovat. Nový seznam pak vytvoříme funkcí `glNewList`. Jako první parametr funkci předáme vygenerovaný index, druhý parametr pak bude jedna z konstant `GL_COMPILE` nebo `GL_COMPILE_AND_EXECUTE`. Po tomto příkazu následuje samotná definice obsahu zobrazovacího seznamu. Zadávání seznamu ukončíme příkazem `glEndList`. Samotné zavolání uloženého seznamu pak realizuje funkce `glCallList`.

Schéma kódu pro používání zobrazovacího seznamu:

```
jmenoSeznamu = glGenLists(1);
glNewList(jmenoSeznamu, GL_COMPILE);
   \\definice grafickych objektu
   ...
glEndList();
...
\\v zobrazovaci funkci
glCallList(jmenoSeznamu);
```

Více o zobrazovacích seznamech nalezneme v [3].

Kapitola 4

Implementace

Aplikaci budu programovat v jazyce C++. Pro implementaci jsem zvolil rozhraní OpenGL a knihovny GLUT a GLEW (viz 3.3.1). V následujících podkapitolách uvedu jejich konkrétní nastavení a popis implementace jednotlivých testů.

4.1 Nastavení OpenGL a knihoven

Ve funkci `main` inicializuji knihovnu GLUT a registruji funkce zpětného volání. Jako zobrazovací funkci nastavím funkci `display`. Ta obsahuje kód pro vykreslení jednotlivých testů rozdělených pomocí příkazu `switch`. Podobně je rozdělena i funkce `idle`, která se volá při nečinnosti programu. Má na starosti animaci jednotlivých testů, měření času, počítání vykreslených snímků a zpracování výsledků testů. Vstup z klávesnice zpracovává funkce `keyboard`, která při stisku klávesy escape ukončí program. Při změně velikosti okna se volá funkce `reshape`, kde se změní pohledové okno pomocí `glViewport` a uloží se rozlišení obrazovky pro pozdější zpracování výsledků testů.

Po inicializaci knihovny GLUT se volá funkce `init`. Ta nastavuje barvu pro mazání framebufferu na černou a také projekční transformaci. Většina testů poběží ve 2D zobrazení, proto projekční matici nastavím pomocí:

```
glOrtho (0.0, 1.0, 0.0, 1.0, -1.0, 1.0)
```

Ostatní matice nechám jednotkové. Zobrazování kurzoru myši zakáži příkazem:

`glutSetCursor (GLUT_CURSOR_NONE)`.

Nakonec ještě musím zaručit vypnutí vertikální synchronizace. V Linuxu toho mohu dosáhnout funkcí `glXSwapIntervalSGI` definovanou v hlavičkovém souboru `GL/glxew.h`, ve Windows stejnou funkčnost poskytuje `wglSwapIntervalEXT` definovaná v `GL/wglew.h`. Připojování hlavičkových souborů i samotné volání těchto funkcí budu muset ohraničit direktivami pro preprocesor – `#ifdef _WIN32`, respektive `#ifdef __linux__`.

4.2 Fill rate

4.2.1 Pixel rate

Ve vykreslovací funkci se u tohoto testu pouze nastavuje barva příkazem `glColor3f` a vykresluje se čtyřúhelník s rohy v souřadnicích [0,0], [1,0], [1,1] a [0,1]. Aktuální barva se mění ve funkci `idle` v intervalu 0.49 až 0.5 pro všechny složky (červená, zelená, modrá). Takto získám tak 5 odstínů šedé (pokud je na každou barvu vyhrazeno 8 bitů). Jednou za 100 snímků kontroluji čas, pokud přesáhl 3 sekundy, test ukončím. Počet vykreslených

pixelů spočítám vynásobením rozlišení obrazovky s počtem vykreslených snímků. Rychlost vykreslování za sekundu pak získám podělením tohoto čísla naměřeným časem.

4.2.2 Texel rate

V tomto testu musím nejdříve načíst obrázek textury ze souboru. Vytvoření textury implementuje funkce `LoadTextureBMP` v souboru `tex_bmp.cpp`. Opakování textury nastavím na `GL_MIRRORED_REPEAT`, což zaručí jednolitost textury bez výrazných přechodů na jejích okrajích. Některé hodně staré grafické karty tuto volbu nepodporují, ale automaticky zobrazují texturu v módu `GL_REPEAT`. Načtenou texturu pak mapuji na čtyřúhelník tak, aby každý bod textury odpovídal pixelu na obrazovce. Grafický akcelerátor tak nebude zatížen výpočty interpolací textury (viz 3.2.2). Kolikrát se má textura na čtyřúhelníku s daným rozlišením opakovat zjistím podělením rozměrů texturovaného čtyřúhelníka rozměry textury.

Pohledovým oknem pak musím pohybovat po otexturovaném čtyřúhelníku, jak je popsáno v 3.2.1. Dosáhnu toho příkazem:

```
glOrtho(0.0+move_x, 1.0+move_x, 0.0+move_y, 1.0+move_y, -1.0, 1.0)
```

Proměnné `move_x` a `move_y` měním ve funkci `idle` od 0.0 do 0.4 po malých krocích odpovídajících rozlišení obrazovky. Abych snížil rychlost pohybování okna, mění se při sudém průchodu pouze směr nahoru/dolů a při lichém doleva/doprava. Dále jsem využil metodu „dva kroky vpřed, jeden krok vzad“, což dále snižuje rychlost pohybu. Ukončení testu a výpočet rychlosti vykreslování probíhá stejně jako u testu pixel rate.

Test s použitím multitexturování je implementován stejně, na každý pixel je však aplikováno až 8 textur. První textura je stejná jako u jednoduchého texturování, ostatní jsou jednobarevné textury, které obraz zesvětlují a zintenzivňují modrou barevnou složku. Přidané textury jsou nanášeny v módu `GL_ADD`, takže se hodnoty barev sčítají. Pokud má grafický akcelerátor méně než 8 texturovacích jednotek, jsou nadbytečné textury ignorovány. Výsledek měření v tomto testu pak musím vynásobit počtem použitých jednotek. Jejich maximální počet zjistím pomocí funkce:

```
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &value);
```

Před spuštěním tohoto testu musím ověřit, zda je multitexturování k dispozici. Mám dvě možnosti, jak to udělat. Mohu kontrolovat, zda je verze OpenGL alespoň 1.3, nebo přímo dostupnost daného rozšíření:

```
if (!GLEW_VERSION_1_3)
    ...
else if (!GLEW_ARB_multitexture)
```

4.3 Texturování

Texturu jsem zvolil stejnou jako v minulém testu, stejný je i způsob jejího opakování. Stěny jehlanu popsaného v části 3.2.2 vytvořím pomocí čtyř trojúhelníků, které mají dva vrcholy postupně v bodech [-2.0, -2.0], [2.0, -2.0], [-2.0, -2.0], [-2.0, 2.0] se souřadnicí $z = -1.0$ a třetí vrchol v bodě [0.0, 0.0] se souřadnicí $z = -101.0$. Podstavu jehlanu vytvářet nepotřebuji, nikdy nebude vidět. Aby se jehlan zobrazil korektně a vytvořil efekt tunelu, je potřeba změnit projekční matici na perspektivní:

```
glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 30.0);
```

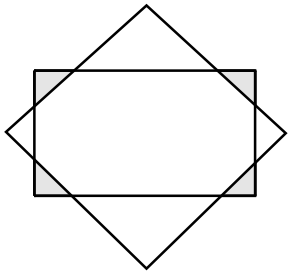
Poslední parametr tohoto volání udává, že budu zobrazovat scénu jen do souřadnice $z = 30.0$. Kdybych nechal zobrazit celý jehlan, byly by v jeho vrcholu textury hodně

deformované kvůli chybám v interpolačních výpočtech. Mohl bych sice využít techniku mipmapování, ale to by pak nemělo význam zvětšovat rozměry textury (viz 3.2.2).

Celý jehlan nakonec pomocí procedury `glRotatef` otočím o 45 stupňů podle osy z. Celý tunel se díky tomu bude jevit symetrický a textury na sebe budou ve hranách jehlanu dobře navazovat.

Po každém překreslení posunu celým jehlanem po ose z směrem k pozorovateli pomocí procedury `glTranslatef`. Dosáhnu tak efektu průletu tunelem, který se na konci postupně zužuje.

Ve funkci `idle` nyní budu muset kromě času hlídat, jestli se pohledové okno nedostalo mimo okraje jehlanu. V části 3.2.2 jsem ukázal, že toto nastane právě v polovině jehlanu v souřadnici $z = 50.0$. Jak ovšem ukazuje náčrtek 4.1, otočení jehlanu o 45 stupňů způsobilo, že se mimo okraje jehlanu dostaneme dříve, přibližně v souřadnici $z = 30.0$.

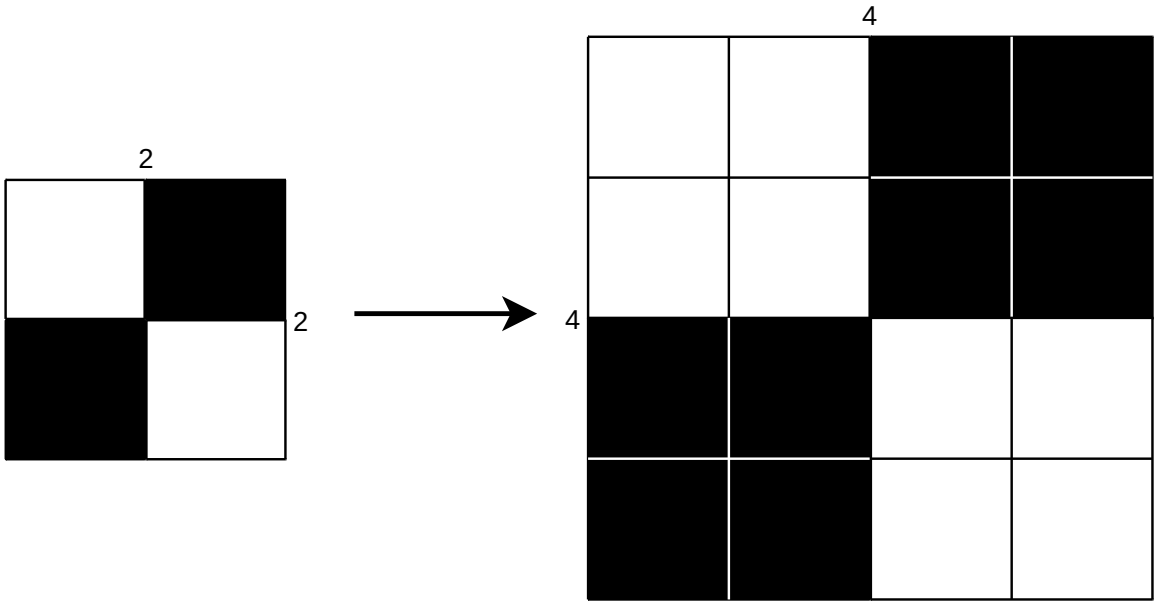


Obrázek 4.1: Posun objektu o 45 stupňů oproti oknu

Při prvním průchodu testem pracuji s texturou o rozměrech 512 pixelů. Po každém průchodu její rozměry zdvojnásobím, abych otestoval závislost velikosti textury na rychlosti zobrazení. Pro zvětšování rozměrů načítané textury je přizpůsobená moje funkce `LoadTextureBMP`, která očekává jako parametr číslo vyjadřující, kolikrát se má rozměr zvětšit. Implementoval jsem jednoduchý algoritmus bodového vzorkování bez interpolačních výpočtů. Každý pixel původní textury se zkopíruje x krát za sebou na řádek před dalším pixelem původní textury. Celý takto roztažený řádek se poté x krát zopakuje. Tento princip ilustruje obrázek 4.2, kde se textura o rozměru 2 rozšíří na rozměr 4. Je vidět, že celková velikost textury se zvětší x^2 krát.

Ve výchozím nastavení se testování ukončí při dosažení rozměru 8192 pixelů, kdy textura v paměti zabírá 256 MB. Ne všechny grafické akcelerátory však podporují takto velké textury. Při každém rozšiřování tedy musím kontrolovat, zda je velikost ještě podporována a zda lze celá textura nahrát do grafické paměti. Maximální rozměr textury zjistím voláním funkce `glGetIntegerv` s parametrem `GL_MAX_TEXTURE_SIZE`. Dostatek grafické paměti mohu ověřit zástupcem textury, kdy se vytvoření textury s danými parametry pouze simuluje. Konkrétní příklad použití zástupce textury:

```
GLint test;
glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA, width, height,
             0, GL_BGR, GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0, GL_TEXTURE_WIDTH, &test);
if (test == 0)
  //textura se nevešla do grafické paměti
else
  //vytvoření textury
```



Obrázek 4.2: Dvojnásobné zvětšení texturovacích dat

4.4 Shadery

Jak vytvářet v OpenGL pomocí jazyka GLSL programovatelné shadery a jak tyto shadery používat pro zpracování vrcholů a fragmentů jsem vysvětlil v části 3.2.3.

U testování fragment shaderu využiji stejné vykreslování i měnění barvy jako u pixel rate, ale na každý pixel aplikuji zátěžový shader program:

```
void main()
{
  vec4 c = gl_Color;
  for(int i = 0; i<300; i++){
    c += vec4(float (i), float (i), float (i), float (i)) *
           vec4(0.0000035, 0.0000024, 0.0000072, 0.0000023);
  }
  gl_FragColor = c;
}
```

Pro každý průchod cyklem se provede 8 výpočetních operací - 4 operace sčítání a 4 operace násobení. Celý cyklus proběhne 300 krát, takže celkový počet operací s čísly s plovoucí desetinnou čárkou je 1200. Program zesvětlí konečnou barvu fragmentu, přičemž nejsilnější bude modrá složka.

U vertex shaderu prohodím jednotlivé barevné složky při výpočtu, aby byly od sebe testy vizuálně odlišné. Navíc do programu musím přidat výpočet pozice bodu, aby se body zobrazily. Výchozí výpočet je vynásobení zadané souřadnice bodu se sjednocenou pohledovou, modelační a projekční maticí:

```
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
```

Celkový počet operací, který vykoná daný shader, zjistím vynásobením rozlišení obrazovky, počtu operací v jednom shader programu a počtu vykreslených snímků. Po vydělení

časem pak dostanu výsledek v GFLOPS. Problémem zjišťování výkonnosti shaderů však je, že první vykreslení scény trvá až 100 krát déle než ostatní průchody. Proto jsem musel začít měřit čas až po prvním vykreslení a při výpočtu pak od počtu snímků jeden odečíst.

Před spuštěním tohoto testu musím ověřit, zda jsou programovatelné shadery k dispozici. Ve specifikaci OpenGL se objevily ve verzi 2.0, což ověřím podmínkou:

`if (!GLEW_VERSION_2_0)`

Dostupnost konkrétního rozšíření pro vertex a fragment shader program zjistím pomocí podmínek:

```
if (!glewGetExtension("GL_ARB_vertex_shader"))
if (!glewGetExtension("GL_ARB_fragment_shader")
```

Kapitola 5

Testování

Testování je nejdůležitější částí tvorby benchmarku. Nejdříve jsem porovnával naměřené výsledky s teoretickými hodnotami, které udává výrobce. Pak jsem testoval vliv procesoru, základní desky a operačního systému na naměřené výsledky. Nakonec jsem srovnával výkon různých grafických akcelerátorů. Vždy jsem používal poslední grafické ovladače na danou kartu a operační systém Windows XP, pokud není uvedeno jinak. Každé měření jsem několikrát opakoval a výslednou hodnotu zprůměroval. To je standartní postup při měření a nebudu ho v této práci více rozebírat.

5.1 Teoretické hodnoty

Parametry grafických akcelerátorů, jejich teoretické rychlosti a podporované verze grafických API jsou pěkně shrnuty na stránkách [7], [8] a [9], nebo se dají dohledat přímo na stránkách výrobce a v manuálech.

Tabulka 5.1 ukazuje naměřené výsledky pro pixel rate, texel rate a výkon shaderů. Ve sloupcích označených „Max“ jsou teoretické (maximální) hodnoty pro dané grafické karty. Vidíme, že se jim naměřené hodnoty blíží a řádově odpovídají. To se dá považovat za úspěch, protože teoretických hodnot nelze nikdy v reálu dosáhnout.

Test / GPU	Pixel rate (GP/s)	Max (GP/s)	Texel rate (GT/s)	Max (GT/s)	Shader (GFLOPS)	Max (GFLOPS)
ATI HD 4770	11.1	12	20.9	24	766	960
ATI HD 3300	2.1	2.8	2.6	2.8	44	56
ATI HD 4890	12.6	13.6	27.3	34	1110	1360
Nvidia G210M	1.1	2.5	2.3	5	31	72

Tabulka 5.1: Srovnání naměřených hodnot s teoretickými rychlostmi

5.2 Operační systém

Naimplementoval jsem aplikaci, která funguje pod více operačními systémy. Mohu proto testovat, jaký má operační systém vliv na grafický výkon. Vybral jsem tři grafické akcelerátory: ATI Radeon 4770 HD, ATI Radeon 3300 HD a Intel X3100. Měření jsem provedl

pod 32 bitovými operačními systémy Windows XP a Ubuntu Linux. Naměřené výsledky jsou v tabulkách 5.2 a 5.3.

Podle očekávání dopadl lépe systém Windows, ovladače grafické karty nebývají v Linuxu tak dobře optimalizované. Zajímavá je však situace při zpracování velkých textur u Radeonu 3300 a Intelu X3100, kdy pod Linuxem je maximální velikost větší než pod Windows. Dalším zajímavým výsledkem je prudký výkonnostní propad při nanášení 256 MB velké textury s kartou ATI 4770. Ve Windows dosahuje rychlosti 190 fps, zatímco v Linuxu pouhých 28 fps. Výsledky u ostatních textur jsou přitom skoro stejné. Toto je pravděpodobně způsobeno tím, že v Linuxu není textura uložena ve speciální texturovací paměti a není rezidentní (viz [3]). V Linuxu s grafickým akcelerátorem Intel navíc nemohl proběhnout test vertex shaderu, pravděpodobně z důvodu chyby v ovladačích (verze ovladačů byla Mesa 7.6). Mohu tedy říct, že akcelerování 3D grafiky v Linuxu funguje, ne však zcela optimálně a bezchybně, záleží velmi na použitých ovladačích.

Test / GPU	Pixel rate (GP/s)	Texel rate (GT/s)	Multi texel (GT/s)	Fragment (GFLOPS)	Vertex (GFLOPS)
ATI 4770 – Windows	11.1	10.1	20.9	766	740
ATI 4770 – Linux	8.2	8.0	17.7	764	738
ATI 3300 – Windows	2.1	1.9	2.6	44	43
ATI 3300 – Linux	1.5	1.4	2.6	46	39
Intel X3100 – Windows	0.34	0.33	0.83	3.5	1.8
Intel X3100 – Linux	0.12	0.10	0.42	2.5	X

Tabulka 5.2: Srovnání výsledků propustností ve Windows/Linux

Velikost textury / GPU	1 MB (fps)	4 MB (fps)	16 MB (fps)	64 MB (fps)	256 MB (fps)
ATI 4770 – Windows	1161	492	269	218	190
ATI 4770 – Linux	1069	490	269	218	28
ATI 3300 – Windows	122	54	31	25	X
ATI 3300 – Linux	88.6	39.6	23.1	19.1	14.8
Intel X3100 – Windows	56.6	29.3	18.2	X	X
Intel X3100 – Linux	29.9	20.0	14.7	12.0	X

Tabulka 5.3: Srovnání výsledků texturování ve Windows/Linux

5.3 Procesor

U grafických benchmarků jako je 3DMark se na konečném výsledku ve velké míře podílí i procesor a rychlost systémové sběrnice a operační paměti. Proto jsem zkoumal, jak tyto parametry ovlivňují naměřené výsledky mého programu. První testovaná sestava byla s grafickou kartou ATI Radeon 3300 HD a s procesorem AMD Phenom II X4. Základní frekvence tohoto procesoru je 3.2 GHz. Budu zkoumat, jak se výsledky liší při podtaktování procesoru na 0.8 GHz. Druhá testovaná grafická karta byla Nvidia GeForce 6600. Tuto kartu jsem nejprve osadil do počítače s procesorem Intel Pentium 4 o frekvenci 2.0 GHz, poté

do sestavy s procesorem Intel Pentium 3 o frekvenci 900 MHz. Tyto dvě sestavy se kromě výkonu procesorů výrazně liší i v rychlosti operačních pamětí. Výsledky testování ukazují tabulky 5.4 a 5.5.

U testu s ATI 3300 a podtaktovaným procesorem AMD Phenom jsou naměřené výsledky v podstatě stejné, drobné rozdíly jsou pravděpodobně způsobeny zaokrouhlovacími chybami nebo procesy na pozadí. Nvidia 6600 dokonce dosáhla nepatrně lepších výsledků na slabším procesoru, který je o generaci starší než jeho protivník. Tento paradox však může být způsoben tím, že na počítači s procesorem Pentium 4 byly dva roky nainstalované Windows XP Home, zatímco počítač s procesorem Pentium 3 běžel pod čerstvě nainstalovanými Windows XP Professional. Každopádně jsem ovšem ukázal, že procesor má na naměřené výsledky mým programem minimální nebo žádný vliv a nemusí se na něj při srovnávání grafických akcelerátorů brát zřetel.

Test / GPU	Pixel rate (GP/s)	Texel rate (GT/s)	Multi texel (GT/s)	Fragment (GFLOPS)	Vertex (GFLOPS)
ATI 3300 – 3.2 GHz	2.10	1.90	2.66	44.1	43.8
ATI 3300 – 0.8 GHz	2.09	1.90	2.66	44.2	43.8
Nvidia 6600 – P4	1.06	0.86	1.10	3.30	1.28
Nvidia 6600 – P3	1.09	0.87	1.11	3.31	1.28

Tabulka 5.4: Srovnání výsledků propustností s různými CPU

Velikost textury / GPU	1 MB (fps)	4 MB (fps)	16 MB (fps)	64 MB (fps)	256 MB (fps)
ATI 3300 – 3.2 GHz	170.0	73.4	43.9	35.9	X
ATI 3300 – 0.8 GHz	170.1	73.4	43.8	35.8	X
Nvidia 6600 – P4	172.0	88.2	55.2	43.6	X
Nvidia 6600 – P3	172.2	89.0	55.4	43.7	X

Tabulka 5.5: Srovnání výsledků texturování s různými CPU

5.4 Srovnání grafických akcelerátorů

Na závěr testování provedu srovnání výsledků několika mnou zvolených grafických akcelerátorů. Všechny testy budu provádět na rozlišení 1280x1024 kvůli porovnatelnosti výsledků testů texturování. Jak jsem ukázal v 5.3, nemusím se při zpracování výsledků zabývat různými procesory v testovaných sestavách. Jako operační systém jsem zvolil MS Windows XP. Testované karty jsou ATI Radeon 4890 HD, ATI Radeon 4770 HD, ATI Radeon 3300 HD, Intel X3100, Intel 4500 HD, Nvidia GeForce 6600, Nvidia GeForce 7600 GT a Nvidia GeForce G210M. Naměřené výsledky jsou v tabulkách 5.6 a 5.7.

Tabulka 5.8 ukazuje pořadí zvolených grafických akcelerátorů v jednotlivých skupinách testů. Na prvních dvou místech ve všech testech se dle očekávání umístily ATI Radeon 4890 HD a ATI Radeon 4770 HD, což jsou v mém výběru jediné dvě plnohodnotné moderní grafické karty. Dále je ve výběru několik integrovaných grafických akcelerátorů (Intel X3100, Intel 4500 HD, ATI 3300 a Nvidia G210M) a dvě starší grafické karty od firmy Nvidia. Je

zajímavé, že starší grafické karty se umístily velice vysoko v testech propustností a texturování, ale propadly ve výkonnosti programovatelných shaderů. Je to způsobeno tendencí ve vývoji grafických akcelerátorů, kdy se velká část výpočtů přesunula právě do programovatelných shaderů a ostatní části grafického akcelerátoru přestaly mít takovou důležitost. U starších grafických karet je také vidět nevyrovnanost ve výkonnosti fragment a vertex shaderů, protože mají ještě oddělené výpočetní jednotky pro oba typy. U novějších karet se sjednocenými shadery je drobný rozdíl ve výkonnosti pravděpodobně způsoben mírně odlišným způsobem měření, kdy u se u vertex shaderu zobrazuje mnohem více vrcholů (viz 3.2.3). Porovnání výsledků výkonu shaderů u jednotlivých grafických akcelerátorů je v grafu na obrázku 5.1. Musel jsem v něm použít logaritmické měřítko, protože mezi testovanými akcelerátory jsou velké rozdíly.

Test / GPU	Pixel rate (GP/s)	Texel rate (GT/s)	Multi texel (GT/s)	Fragment (GFLOPS)	Vertex (GFLOPS)
ATI 4890 HD	12.9	12.5	27.3	1110	993
ATI 4770 HD	11.1	10.1	20.9	766	740
ATI 3300 HD	2.1	1.9	2.7	44.2	43.8
Intel X3100	0.34	0.32	0.83	3.52	1.82
Intel 4500 HD	0.36	0.33	0.84	6.99	5.75
Nvidia 6600	1.06	0.86	1.10	3.30	1.28
Nvidia 7600 GT	4.68	2.17	3.21	10.2	3.23
Nvidia G210M	0.88	0.75	1.78	30.1	29.7

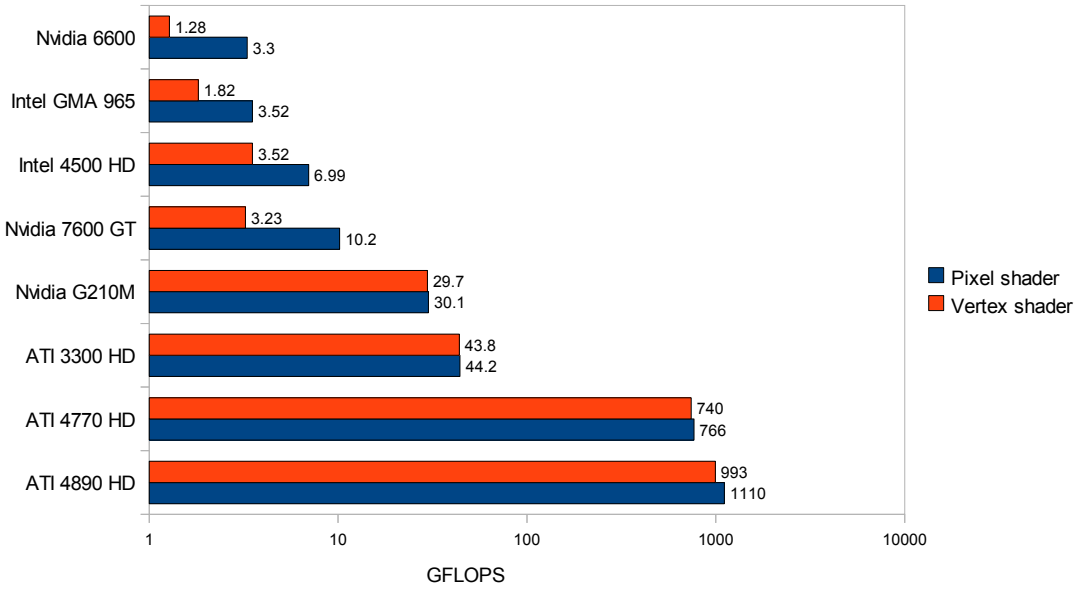
Tabulka 5.6: Srovnání výkonu různých GPU

Velikost textury / GPU	1 MB (fps)	4 MB (fps)	16 MB (fps)	64 MB (fps)	256 MB (fps)
ATI 4890 HD	3676	1626	906	694	632
ATI 4770 HD	1633	697	373	297	250
ATI 3300 HD	170	73.7	43.4	35.2	X
Intel X3100	78.9	45.6	24.9	X	X
Intel 4500 HD	121	67.8	39.6	10.1	X
Nvidia 6600	172	89.0	55.2	40.6	X
Nvidia 7600 GT	384	190	114	83.9	X
Nvidia G210M	256	138	83.6	44.0	28.9

Tabulka 5.7: Srovnání výkonu různých GPU

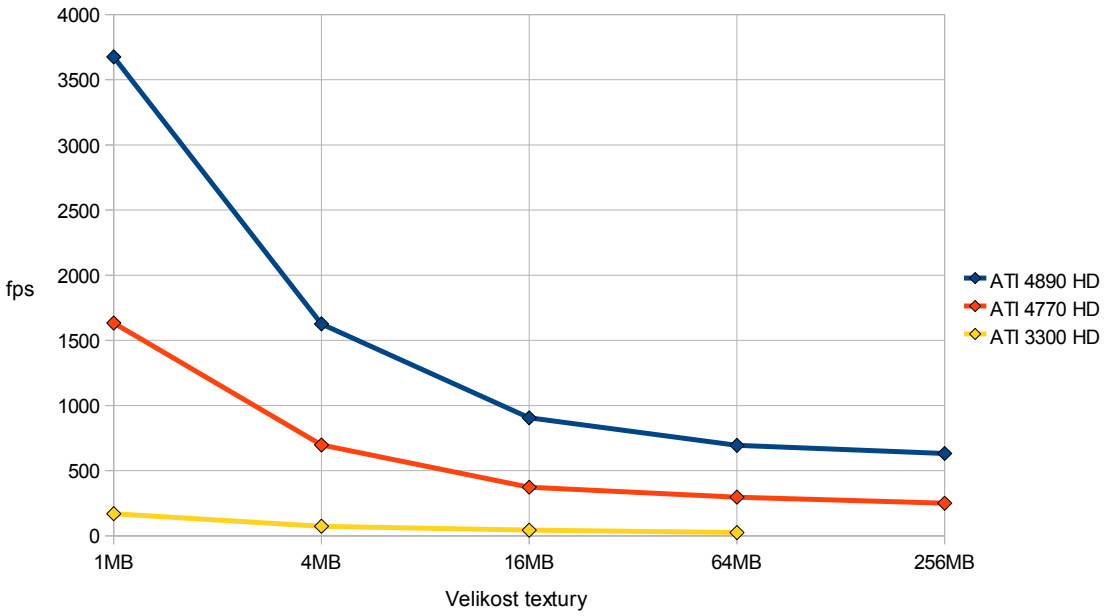
Pořadí/test	Fill rate	Texturování	Shadery
1	ATI 4890	ATI 4890	ATI 4890
2	ATI 4770	ATI 4770	ATI 4770
3	Nvidia 7600	Nvidia 7600	ATI 3300 HD
4	ATI 3300	Nvidia G210M	Nvidia G210M
5	Nvidia 6600	Nvidia 6600	Nvidia 7600
6	Nvidia G210M	ATI 3300	Intel 4500
7	Intel 4500	Intel 4500	Intel X3100
8	Intel X3100	Intel X3100	Nvidia 6600

Tabulka 5.8: Pořadí GPU podle jednotlivých testů

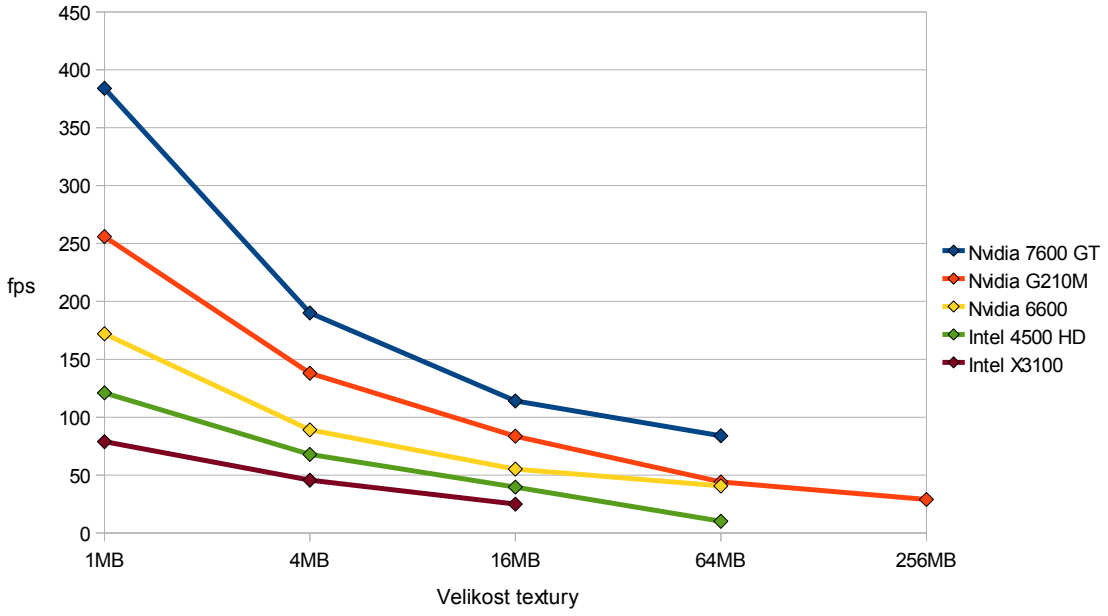


Obrázek 5.1: Porovnání výkonu pixel a vertex shaderů

Na grafech 5.2 a 5.3 je vidět, jak klesá počet snímků vykreslených za sekundu v závislosti na velikosti použité textury. Kvůli velkým výkonnostním rozdílům mezi testovanými grafickými akcelerátory jsem nemohl nanést všechny do jednoho grafu. Velikosti použité textury tvoří geometrickou řadu, proto je osa x v logaritmickém měřítku. Přesto je vidět, že rychlost vykreslování se zvětšující se texturou prudce klesá.



Obrázek 5.2: Závislost rychlosti vykreslování na velikosti textury



Obrázek 5.3: Závislost rychlosti vykreslování na velikosti textury

Kapitola 6

Závěr

Cílem této práce bylo vytvořit aplikaci, která bude měřit výkonnost dnešních grafických akcelerátorů. Dále jsem stanovil požadavek, aby tato aplikace byla multiplatformní a fungovala alespoň na systémech MS Windows a Linux.

Tyto cíle se podařilo splnit, díky použitým knihovnám GLUT a GLEW není aplikace závislá na operačním systému. Co se týče měření výkonnosti, aplikace nezjišťuje dostupnost posledních technologií a rychlost jejich zpracování. Zaměřil jsem se na testování základních částí grafické karty, jako jsou texturovací jednotky a programovatelné shadery. Díky tomuto přístupu nejsou naměřené hodnoty ovlivněny dalšími parametry počítače, jako je např. procesor. Výkon programovatelných shaderů je vyjádřen ve zpopularizovaných jednotkách GFLOPS a dá se přímo použít ke srovnávání grafických akcelerátorů, nezávisle na použitém rozlišení při testu. Celé měření netrvá déle než jednu minutu.

V této aplikaci je hodně prostoru pro vylepšování a další vývoj. Dalo by se přidat grafické uživatelské rozhraní pro nastavení parametrů a výběru požadovaných testů. Dále by se mohl implementovat výstup naměřených výsledků v podobě např. html stránek, případně vytvořit systém pro online sbírání výsledků od různých uživatelů. Bylo by možné i rozšířit počet testů s využitím některých možností novějších verzí OpenGL a pokročilých technik v programovatelných shaderech.

Díky zpracovávání této práce jsem se dozvěděl spoustu nových informací o moderních grafických akcelerátorech, získal přehled o jejich současné nabídce a lépe se orientuji v rozdílech ve výkonnosti jednotlivých typů. Naučil jsem se základy programování v OpenGL a zjistil, jak se jeho možnosti mohou rozšiřovat pomocí přídavných knihoven.

Literatura

[1] HODGE, B.: OpenGL Texture Tutorial [online]. [rev. 2001-08-25], [cit. 2010-05-02]. URL http://www.nullterminator.net/gltexture.html

[2] RADIL, P.: *Programovatelné shadery v OpenGL*. Bakalářská práce, FIT VUT v Brně, Brno, 2009.

[3] SHREINER, D.; aj.: *OpenGL Průvodce programátora*. Brno: Computer Press, 2006, ISBN 80-251-1275-6, přeložil: Jiří Fadrný.

[4] WWW stránky: GeForce 8800 – Technical Specifications [online]. [cit. 2010-05-01]. URL http://www.nvidia.com/page/8800_tech_specs.html

[5] WWW stránky: Getting started [online]. [rev. 2010-04-13], [cit. 2010-04-30]. URL http://www.opengl.org/wiki/Getting_started

[6] WWW stránky: GLEW [online]. [rev. 2010-04-27], [cit. 2010-05-02]. URL http://glew.sourceforge.net/

[7] WWW stránky: Comparison of AMD graphics processing units [online]. [rev. 2010-05-06], [cit. 2010-05-07]. URL http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units

[8] WWW stránky: Comparison of Nvidia graphics processing units [online]. [rev. 2010-05-06], [cit. 2010-05-07]. URL http://en.wikipedia.org/wiki/Comparison_of_Nvidia_graphics_processing_units

[9] WWW stránky: Intel GMA [online]. [rev. 2010-05-06], [cit. 2010-05-07]. URL http://en.wikipedia.org/wiki/Intel_GMA

[10] *Aquamark The Reality Benchmark – Documentation*. Massive Development a JoWooD Productions, 2003.

[11] ŠTĚPÁNEK, M.: Aquamark 3: tvrdý útok na 3D Mark [online]. [rev. 2003-09-18], [cit. 2010-04-30]. URL http://www.svethardware.cz/art_doc-22F7D996704C2961C1256DA400323C55.html

Příloha A

Obsah CD

Na přiloženém CD jsou zdrojové soubory mé aplikace a přeložený program i s potřebnými knihovnami GLUT a GLEW pro spuštění pod systémem Microsoft Windows. Dále CD obsahuje i text této práce, zdrojové soubory v LaTeXu a prezentační plakát v nekomprimovaném formátu. Adresářový strom je následující:

`<ROOT>`

- `<bin>` – přeložený program a knihovny GLUT a GLEW
  - `<tex>` – použité textury
  - `config.cfg` – konfigurační soubor
  - `readme.txt` – návod k aplikaci
- `<doc>` – zdrojové soubory tohoto dokumentu v LaTeXu
  - `<fig>` – obrázky použité v tomto dokumentu
- `<src>` – zdrojové soubory aplikace
  - `<tex>` – použité textury
  - `makefile` – pro přeložení pod Microwoft Windows a Linux
- `ibp10.pdf` – tento dokument
- `poster.png` – prezentační plakát ve vysokém rozlišení

Příloha B

Manuál

Program se přeloží příkazem `make` v adresáři `src`. V daném operačním systému přitom musí být nainstalovány knihovny GLUT a GLEW. Ve Windows se při překladu linkuje přímo knihovna `glew32.dll`, která je v adresáři `src` obsažena.

Aplikace se dá spustit buď příkazem `make run` v adresáři `src`, nebo přímo spuštěním binárního souboru `pure.exe` ve Windows, respektive `pure` v Linuxu. Soubor s texturou `256.bmp` se musí nacházet v podadresáři `tex`.

Po spuštění provede program sérii fill rate testů, poté test rychlosti texturování s postupně se zvětšující texturou a nakonec test výkonnosti vertex a pixel shaderů. Testy, které nejsou podporovány daným grafickým akcelerátorem, jsou automaticky přeskočeny. Naměřené výsledky a případná chybová hlášení se vypíšou na standardní výstup a do souboru `results.txt`. Do souboru se navíc uloží i čas testu, typ grafického akcelerátoru, verze OpenGL, verze GLEW a rozlišení obrazovky. Pokud se nepodaří otevřít soubor pro zápis, výsledky jsou vypsány pouze v terminálu a program před ukončením čeká na stisk klávesy *enter*, aby si uživatel stihl přečíst výsledky z konzole.

V základním nastavení poběží aplikace v rozlišení 1280 na 1024 bodů. To můžeme změnit v konfiguračním souboru `config.cfg`, kde program hledá řádek začínající textem „resolution=“. Za znakem '=' se očekává požadované rozlišení ve formátu *šířka*'x'*výška*, např. tedy „resolution=800x600“. Případné bílé znaky v řetězci se ignorují.

Program lze kdykoliv ukončit stiskem klávesy *escape*.

Příloha C

Plakát


Zjistěte čistý výpočetní výkon
vašeho GPU v GFLOPS!
Pure Benchmark
Funkční pod windows i linux
Test nezávislý na vašem procesoru
za méně než jednu minutu !