VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

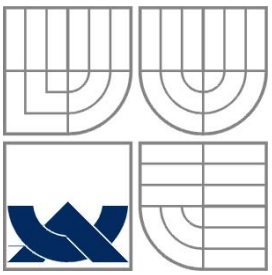
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZPRACOVÁNÍ OBRAZU V REÁLNÉM ČASE

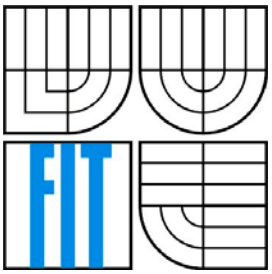
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Polok Lukáš
AUTHOR

BRNO 2007



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

ZPRACOVÁNÍ OBRAZU V REÁLNÉM ČASE

REAL-TIME IMAGE PROCESSING

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — Polok Lukáš
AUTHOR

VEDOUCÍ PRÁCE — Zemčík Pavel, doc. Dr. Ing.
SUPERVISOR

BRNO 2007

Zadání bakalářské práce/2289/2006/xpolok00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Polok Lukáš**
Obor: Informační technologie
Téma: **Zpracování obrazu v reálném čase**
Kategorie: Zpracování obrazu

Pokyny:

1. Prostudujte literaturu na téma algoritmy zpracování obrazu se zaměřením na časově efektivní výpočty
2. Navrhněte vhodnou sadu algoritmů pro zvolenou úlohu zpracování obrazu (nutno konzultovat)
3. Navrženou sadu operací implementujte v jazyce C
4. Vyhodnoťte dosažené výsledky

Literatura:

- Žára, J. a kol.: Počítačová grafika principy a algoritmy, Grada, Praha 1992
- Žára, J. Beneš B. Felker P.: Moderní počítačová grafika, Computer Press, 1998
- dále dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Prostudujte literaturu na téma algoritmy zpracování obrazu se zaměřením na časově efektivní výpočty
- Navrhněte vhodnou sadu algoritmů pro zvolenou úlohu zpracování obrazu (nutno konzultovat)

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zemčík Pavel, doc. Dr. Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Pavel Zemčík
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Lukáš Polok**
Id studenta: 84186
Bytem: Makov 1, 570 01 Litomyšl
Narozen: 15. 05. 1985, Brno - Veveří
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Zpracování obrazu v reálném čase
Vedoucí/školitel VŠKP: Zemčík Pavel, doc. Dr. Ing.
Ústav: Ústav počítačové grafiky a multimédií
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Tato práce se zabývá urychlením operací nad dvojrozměrným obrazem pomocí programovatelných grafických akcelerátorů (také GPU). Sada implementovaných operací zahrnuje některé základní, většinou atomické, operace nad obrazem. Knihovna je navržena tak aby mohly být lehce přidány i nové funkce.

Implementace grafických algoritmů na GPU profituje z jejich vysokého výpočetního výkonu a rychlého vývojového cyklu. Knihovna však obsahuje i referenční implementaci algoritmů v jazyce C, poskytující stejnou funkcionalitu v případech kdy cílový hardware neodpovídá daným požadavkům.

Klíčová slova

Zpracování obrazu, GPU, OpenGL, programovatelný grafický hardware, počítačová grafika.

Abstract

This work is concerned with acceleration of two-dimensional image processing using programmable graphical hardware (or GPU‘s). Set of implemented operations includes some basic, mostly atomic, image processing functions. However, library is designed in such a manner so it can be easily extended by adding new functions.

Implementation of graphical algorithms using GPU’s profits from their high computational power and fast development cycles. For situations where target hardware doesn’t support programmable shading, required by the library, reference „C“ language implementation is available.

Keywords

Image processing, GPU, OpenGL, programmable graphic hardware, computer graphics.

Citace

Lukáš Polok: Zpracování obrazu v reálném čase, bakalářská práce, Brno, FIT VUT v Brně, 2007

Zpracování obrazu v reálném čase

Prohlášení

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

……
Jméno Příjmení
Datum

Poděkování

Chtěl bych použít této příležitosti abych poděkoval svému vedoucímu, panu Zemčíkovi, za odbornou pomoc, zaměstnancům firmy NVidia za jejich vynikající práci a publikace, Marku Segalovi a Kurtovi Akeleymu za OpenGL a v neposlední řadě Bjärne Stroustrupovi za programovací jazyk C++.

Obsah

Úvod

Zpracování obrazu je v podstatě jakékoliv zpracování informace kde vstupem je obraz, například fotografie, nebo snímky videa. Výstupem nemusí být nutně obraz, může to být například popis nějakých vlastností vstupního obrazu. Většina algoritmů zpracovávajících obraz interpretuje obraz jednoduše jako dvojrozměrný signál, na něž aplikuje postupy, běžné pro zpracování signálů.

Mezi typické problémy zpracování obrazu patří geometrické operace jako zvětšení, rotace, zkosení nebo například nějaká nelineární transformace. Velmi běžné jsou barevné transformace jako úprava jasu, kontrastu, gamma korekce, kvantizace, nebo konverze do jiného barevného systému. Seznam by mohl pokračovat donekonečna, jen námětem kombinace obrazů, registrace obrazů, segmentace nebo zlepšení kvality obrazu jako doostření nebo redukce šumu.

1 Akcelerace operací nad obrazem

Operace nad obrazem lze akcelerovat (tedy urychlovat) mnoha způsoby. Prvním krokem by mělo být nalezení optimálního algoritmu, jež najde požadované řešení s přípustnou chybou v minimálním čase. Pokud vyžadujeme větší rychlost, můžeme se ubírat dvěma směry. Buďto můžeme zvětšit přípustnou chybu a tím dovolit zjednodušení výpočtů, potřebných pro provedení dané operace, nebo můžeme použít hardware s větší výpočetní silou. Vzhledem k situaci návrhu grafické knihovny je nemyslitelné snižovat přesnost výpočtů, jelikož není předvídatelné jak přesné výpočty budou uživatelé vyžadovat. Tudíž je potřeba prozkoumat možnosti urychlení výpočtů použitím větší výpočetní síly.

1.1 Paralelní zpracování

Jednoduchý, technologicky nenáročný způsob na zrychlení výpočetní síly je použití paralelního zpracování. Úloha je nějakým způsobem vykonávána několika výpočetními jednotkami, na každé z nichž běží výpočetní proces. Prakticky to může být provedeno použitím počítače či serveru, obsahujícího větší počet procesorů, popřípadě použité procesory mohou obsahovat více „jader“, v nichž se paralelně provádí program. Paralelní zpracování obrazu může být také distribuované, běží-li více spolupracujících programů na separátních počítačích, komunikujících po síti. S paralelním zpracováním vyvstává otázka jak přerozdělit danou úlohu na více paralelně běžících procesů.

1.1.1 Rozdělení obrazu na části

Na první pohled jednoduché řešení je nechat každý proces zpracovávat část obrazu, například rozdělit obraz na horní polovinu a dolní polovinu. Po zpracování obou částí se obraz znova složí dohromady. Tento přístup má tu nevýhodu že obraz může v jedné z polovin obsahovat data, jež vyžadují delší zpracování, tudíž některé procesy budou více vytížené než jiné. To jde do jisté míry vyřešit zpětnou vazbou z času zpracování na přidělenou plochu obrazu. Je tedy cílem přerozdělit obraz na různé části, tak velké aby všechny procesy byly vytížené rovnoměrně. Tím se o něco zvětší režie zpracování.

1.1.2 Prokládané zpracování snímků

Pro potřeby zpracování proudu videa je možné rozdělit práci jednoduše po snímcích. Každý proces zpracovává pouze každý *n*-tý snímek. Tento přístup má výhodu jednoduchosti, avšak nevýhod je hned několik. Urychlení zpracování může být libovolné, jednoduše přidáváním dalších výpočetních jednotek se dá zvětšovat donekonečna. Avšak zpoždění je konstantní.

Pokud například zpracování snímku trvá jednu vteřinu a je požadováno zpracování proudu videa se dvaceti pěti snímky za vteřinu, můžeme nechat pracovat dvacet pět výpočetních jednotek, které potom práci stihnou, aniž by nějaké snímky byly ztraceny. Ale zpoždění takového systému bude stále jedna vteřina, což v některých aplikacích může být překážkou.

Další potíže přibývají, pokud jsou ke zpracování snímku potřebná nějaká data z předchozího či předchozích snímků. Je stále možné použít tuto metodu, ale je nutno počítat s nutností přenášet data mezi jednotkami a tím pádem vyšší režií.

1.1.3 Rozdělení algoritmu na oddělené části

Obě předchozí metody počítaly pouze s rozdělením zdrojových dat. Je také možné přerozdělit jednotlivé fáze algoritmu a nechat každou výpočetní jednotku vykonávat určitou fázi algoritmu. To ale není zcela univerzální přístup jelikož ne každý algoritmus lze takto rozdělit a ještě méně algoritmů lze rozdělit takovým způsobem, aby mohly být spouštěny na libovolném počtu výpočetních jednotek.

1.2 Funkční platformy

Jednotlivé platformy mohou mít různé účely a tudíž mohou být různě výkonné z hlediska řešeného problému. Výkon se dá tedy zdánlivě zvýšit i výběrem vhodné platformy. Dnes se pro zpracování obrazu běžně používají osobní počítače nebo servery, programovatelné pole FPGA, nebo grafické akcelerátory (GPU), jež jsou dnes obvyklou součástí osobních počítačů.

Knihovna je určena pro použití na osobních počítačích, známe tedy cílovou architekturu a můžeme se pokusit napsat kód jež by na ní běžel co nejoptimálněji. Dnešní procesory (CPU)

disponují řadou rozšiřujících instrukčních sad, jako MMX, SSE (2, 3). Ty jsou někdy označovány jako „multimediální“ instrukční sady protože oproti klasické instrukční sadě procesorů typu 80x86 obsahují instrukce pro operace s vektory. Běžné obrazy jsou složeny ze tří nebo čtyř barevných složek (červená, modrá, zelená a v některých případech „alpha“) a instrukce, schopné pracovat se všemi čtyřmi složkami obrazu najednou mohou některé operace výrazně urychlit, jak ostatně ukázala MMX - nejmladší ze jmenovaných rozšíření.

1.2.1 FPGA

Programovatelné hradlové pole, FPGA je polovodičové zařízení, obsahující matici programovatelných logických komponentů a síť jejich propojení. Logické komponenty můžou být naprogramovány tak, aby realizovaly určitou logickou funkci (jako AND, OR, XOR) nebo složitější kombinační funkce jako dekodéry nebo základní matematické funkce. Většina typů FPGA obsahuje rovněž paměťové obvody, buďto jako klopné obvody nebo celé paměťové bloky.

Propojující síť umožňuje s určitým stupněm svobody naprogramovat propojení mezi logickými komponentami a tím realizovat nějaký logický obvod. Tudíž oproti klasickým procesorům FPGA neprovádí instrukce programu, ale realizuje cílenou funkci. Klíčovou vlastností FPGA je možnost kdykoliv přeprogramovat (část) plochy čipu a tím změnit realizovanou funkci podle potřeby.

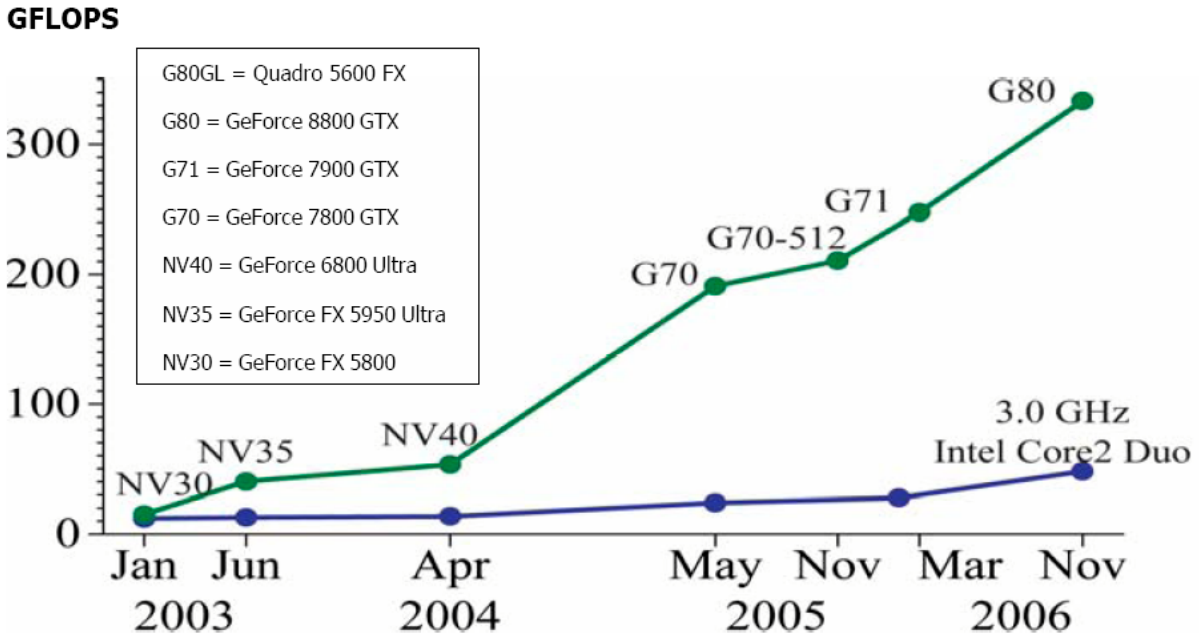
FPGA umožňují opravdové paralelní zpracování, obvod může být navržen tak aby data proudila v několika paralelních cestách. Nevýhodou je potřeba nějakého rozhraní pro použití v PC, zpravidla kartu do PCI slotu nesoucí potřebné rozhraní. Na druhou stranu je možné vytvořit samostatné zařízení, obsahující pouze FPGA a nějakou řídící logiku což může být v některých (průmyslových) aplikacích vhodnější než použití počítače.

1.2.2 Grafické akcelerátory

Relativně novou platformou pro zpracování obrazu jsou grafické akcelerátory. Pojem grafický procesor je známý již dlouho, okolo roku 1980 bylo obvyklé používat univerzální procesory pro zpracování grafiky. Počítače tehdy obsahovaly centrální procesor a procesor, dedikovaný pro zpracování grafiky. (tzv. grafický procesor) Začínaly se vyrábět procesory se specializovaným řadičem paměti pro obrazovou paměť. Postupem času se zvyšovala výkonnost a složitost procesorů, avšak jejich možnosti byly vždy limitované, jelikož obsahovaly pouze výrobcem dané funkce.

S rokem 2000 přišel pojem „programovatelné stínování“ (angl. „programmable shading“), grafické procesory obsahují pevně daný řetězec navazujících bloků, určených pro kreslení trojrozměrného obrazu (angl. „fixed-function pipeline“) z nichž některé části šly nahradit uživatelsky definovanými programy. Každý pixel tedy mohl být zpracován krátkým programem ve speciálním assembleru jež vzal jako vstupy textury a interpolované souřadnice vrcholů geometrie a vyprodukoval

výslednou barvu. („pixel shader“) Stejně tak každý vrchol kreslené geometrie mohl být zpracován programem, zodpovědným za transformace souřadnic („vertex shader“).



Obrázek 1: Graf výkonnosti grafických akcelerátorů [6]

I když grafické akcelerátory stanovují řadu omezení, mají kratší vývojový cyklus než procesory (jejich výkon stoupá rychleji) a dnes mají mnohanásobně větší výpočetní výkon než klasické procesory. Proto se začínají používat i v oblastech, pro které původně nebyly určeny. Anglicky je to označováno jako GPGPU - „general purpose computing on graphical processor unit“, tedy univerzální výpočty na grafickém procesoru. Mezi aplikace patří rychlé matematické funkce, prostorové databáze, hledání ropy, předpověď cen akcií nebo dokonce vyhledávání na internetu.

1.3 Použitelné rozhraní pro GPU

V následující kapitole zhodnotíme dnes používaná grafická aplikační rozhraní a zhodnotíme nejlepší variantu.

1.3.1 DirectX

Mezi často používané rozhraní patří DirectX, vyvíjené společností Microsoft. Je to relativně mladé rozhraní, které vzniklo v roce 1995 jako alternativa k OpenGL, které tehdy bylo podporováno jejich systémem Windows NT. Záměrem bylo zcela vytlačit OpenGL z trhu, což se naštěstí nepovedlo (námětem můžeme připomenout vzorové příklady pro programování OpenGL pod Windows, jež byly napsány tak aby ukázaly nízkou výkonnost). Dnes jeho existenci považujeme za samozřejmou,

ale díky špatnému návrhu bylo ze začátku velice neoblíbené. Můžeme zmínit dopis známého vývojáře Johna Carmacka, navrhující společnosti Microsoft zastavení vývoje a podpory tohoto rozhraní.

DirectX je sice zdarma, ale bohužel je implementováno pouze pod operačním systémem Windows, což může být pro implementaci grafické knihovny značná nevýhoda jelikož to zužuje okruh potenciálních uživatelů. Existuje sice jakási unixová emulace, ale protože zdrojové kódy DirectX nejsou volně přístupné, jde pouze o API s podobným chováním vytvořené metodami reverzního inženýrství.

Pro uživatelské programy je použit jazyk HLSL, „high-level shading language“ tedy vysokoúrovňový jazyk pro stínování. Jeho syntax je podobná jazyku C, nebo stínovacímu jazyku CG, vyvinutého společností NVidia.

1.3.2 OpenGL

Další, velmi oblíbené rozhraní je OpenGL, vyvíjené společností SGI. Na rozdíl od DirectX, jež odkrývá funkce grafického hardware, OpenGL je specifikace grafických funkcí a jejich chování. OpenGL doslova skrývá rozhraní a možnosti grafického hardware, tedy pokud hardware nepodporuje nějakou funkci, OpenGL použije softwarovou emulaci. Vlastní implementace je přitom vytvářená výrobci grafických karet.

Existují implementace OpenGL pro Mac OS, Windows, Linux a hodně dalších unixových operačních systémů. Existuje také několik implementací, poskytujících funkce OpenGL na platformách kde hardwarová akcelerace není dostupná, nejznámější z nich je Mesa.
OpenGL je neustále vyvíjeno, ačkoliv už ne firmou SGI. Od roku 2006 se o vývoj stará Khronos Group. Je ustanovena rada pro údržbu architektury OpenGL (ARB), jež sestává z výrobců grafických karet a dalších firem jež se podílejí na vývoji. Microsoft Původně podal návrh na její založení, ale opustil ji roku 1993. OpenGL nemá tolik verzí jako DirectX (poslední verze OpenGL ke 29.4.2007 je 2.1, verzí DirectX je už 10 - nepočítaje obměsíční aktualizace devátá verze) nová funkcionalita je přístupná jako rozšíření, přičemž nové funkce jsou linkované dynamicky (vstupní bod funkce vrácen dle jména funkce) a původní API zůstává stále stejné.

Co se týče programovatelného stínování, OpenGL nabízí hned tři jazyky, první a nejstarší je podobný assembleru. Běžně se mu říká nízkoúrovňový stínovací jazyk (angl. „low-level shading language“). Druhý, vysokoúrovňový stínovací jazyk je velmi podobný jazyku C. Nakonec je tu třetí alternativa firmy NVidia, programovací jazyk CG (C pro grafiku). CG má vynikající kompilátor, jež umí přeskupit operace podle obsazené grafické karty tak, aby se maximálně využily její vlastnosti (přeskupení výpočetních instrukcí a instrukcí pro přístup do paměti, využití nových výpočetních instrukcí). Na grafických kartách firmy NVidia je tento kompilátor použit i pro překlad vysokoúrovňového stohovacího jazyka OpenGL. Vzhledem k rychlosti vývoje grafických akcelerátorů dává větší smysl použití vyššího jazyka, jelikož v nových generacích grafických karet

pravděpodobně poběží optimálněji než assembler, laděný na kartách, dostupných v době vývoje aplikace.

1.3.3 CUDA

Opravdu novým rozhraním je CUDA. Toto rozhraní je vyvíjeno firmou NVidia a vzniklo letos (2007). CUDA je zkratka pro „compute unified device architecture“, tedy „unifikovaná architektura výpočetního zařízení“. Je to nová hardwarová a softwarová architektura pro management a spouštění programů na GPU bez podpory jakéhokoliv grafického rozhraní. GPU je přitom modelováno jako paralelní výpočetní jednotka. CUDA odstraňuje několik nevýhod, do nynějška spojených s používáním GPU pro obecné výpočty:

- GPU mohlo být používáno pouze skrze grafické API, což vyžaduje jeho znalost
- Grafické API většině negrafických výpočtů nevyhovuje.
- Klasické shadery, běžící na GPU mohou číst z jakékoliv adresy paměti, ale mohou zapisovat pouze na jedinou, předem danou adresu výsledného pixelu.
- Některé aplikace vyžadují větší paměťovou prostupnost, než je na GPU v současnosti dosažitelná.

Většina problémů je odstraněna použitím zvláštních paměťových modifikátorů, jež umožňují výrazně snížit potřebný počet přístupů do paměti a optimálně využít všechny dostupné registry a cache. Počítá se s paralelním programováním s velmi vysokým počtem paralelně běžících vláken.

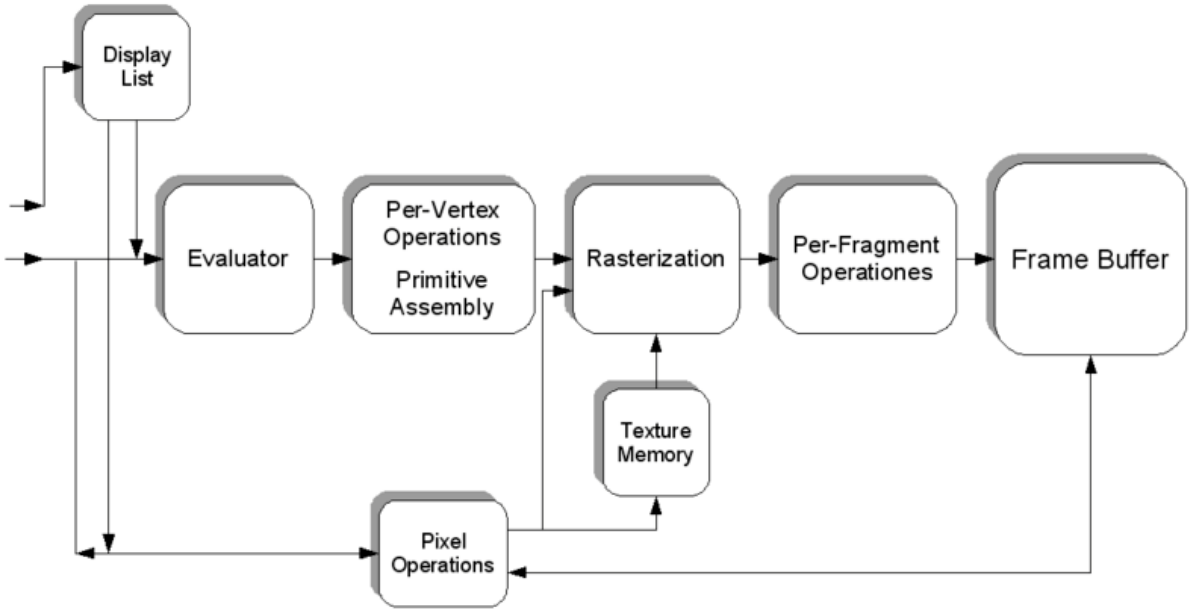
CUDA je minimálním rozšířením jazyka C, jedná se o čtyři hlavní změny:

- kvalifikátory typu funkce pro rozlišení cíle spuštění funkce (CPU nebo GPU)
- paměťové kvalifikátory pro specifikaci umístění proměnných v paměti
- direktivy pro ovládání způsobu spouštění programu
- vestavěné proměnné, obsahující identifikátor spouštěného vlákna a informace o něm

Hardwarové požadavky jsou rozumné, podporovány jsou grafické karty jež jsou dnes běžné (GeForce 8800, Quadro FX 4600 a novější; grafické karty od ATI samozřejmě podporovány nejsou, nehledě k tomu že ATI ještě nemá žádný model na stejné technologické úrovni). Naneštěstí jsem v době na začátku vývoje knihovny měl k dispozici pouze GeForce 7950, tudíž jsem nemohl toto rozhraní použít.

1.4 Programovatelné GPU a OpenGL

Jak již bylo řečeno výše, OpenGL má k dispozici programovací jazyka nízké i vysoké úrovně a rozšiřující jazyk CG společnosti NVidia. K implementaci grafických algoritmů byl využit vysoko-úrovňový programovací jazyk, GLSL (angl. „GL shading language“). Někdy se mu též říká „glslang“. Na následujícím obrázku je blokový diagram OpenGL:



Obrázek 2: blokové schéma OpenGL (převzato z [3])

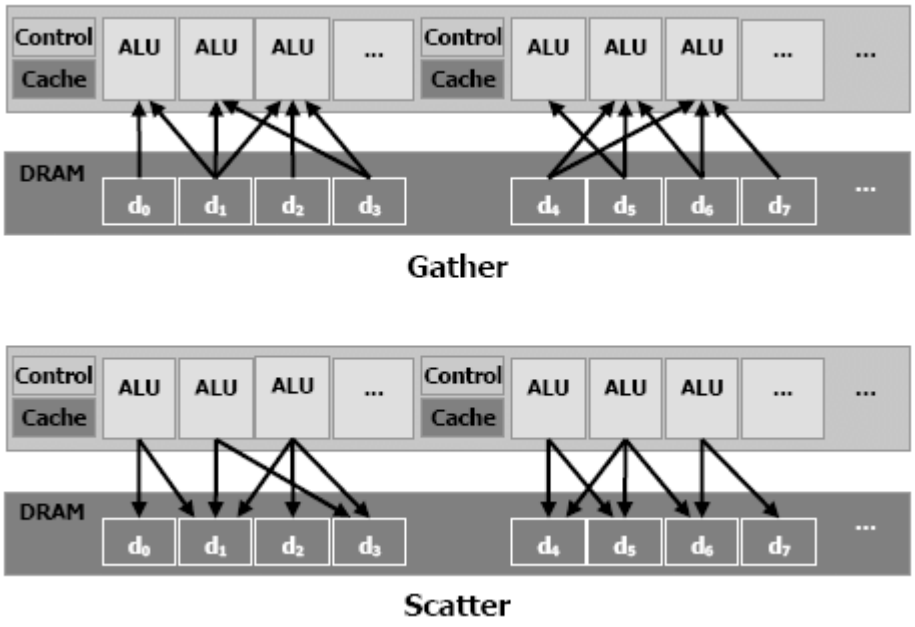
Zleva přichází primitiva k vykreslení, ty mohou být uloženy jako a duplikovány z display listu, následuje vyhodnocení, operace s vrcholy, skládání primitiv, rasterizace primitiv a konečně výsledky operací nad fragmenty jsou zapsány do obrazové paměti.

Programovatelné části jsou operace s vrcholy (vertex shader), skládání primitiv (geometry shader) a operace nad fragmenty (fragment shader, někdy též pixel shader (terminologie DirectX)).

Pro potřeby implementace většiny grafických operací by stačil fragment shader, avšak z hlediska rychlosti provádění je vhodné předpočítat si některé části rovnice ve vertex shaderu a nechat si jejich hodnoty (ne)lineárně interpolovat pro každý pixel kdykoliv je to možné.

Jak si lze z diagramu povšimnout, zdrojová obrazová data pro zpracování shadery jsou umístěna v paměti textur. Shader může obsahovat speciální globální proměnné, jejichž hodnoty lze před započetím kreslení nastavit voláním příslušné funkce. Další parametry lze zadávat buď jako souřadnice vrcholů kreslené geometrie, nebo jako stavové proměnné OpenGL, jež jsou shaderům zpřístupněny pro čtení. Sem patří všechny parametry světel, parametry mlhy (fog), transformační matice, klipovací roviny, parametry generování souřadnic textury, parametry materiálu a parametry bodu.

Zřejmou nevýhodou je nemožnost zapisovat do textury, jež je použita jako vstupní data. Říkáme že lze data shromažďovat (angl. „gather“), ale nelze je rozptylovat (angl. „scatter“). Operace znázorňuje následující obrázek. Z toho plynou jistá omezení pro implementované algoritmy, většinou však lze algoritmus přeskupit tak, aby mu stačilo data shromažďovat.



Obrázek 3: Znázornění operací gather a scatter (převzato z [6])

Programovací jazyk GLSL je podobný jazyku C, navíc obsahuje vektorové a maticové datové typy do rozměru čtyři. Matice musely být původně čtvercové, ale v druhé verzi GLSL již toto omezení neplatí. Jako reference jazyka byla použita [4].

Použití standardních operátorů na vektory způsobí provedení paralelní operace s odpovídajícími složkami vektorů. Je tedy možné provádět takové operace jen mezi vektory stejných velikostí. Pokud se v operaci ocitne vektor a skalár, skalár je interpretován jako vektor o potřebném rozměru se všemi složkami s hodnotou původního skaláru. Stejné pravidlo platí i pro operace mezi maticí a skalárem.

Mezi vektorem a maticí je definována pouze operace násobení, jež odpovídá algebraickému násobení vektoru a matice. Počet komponent vektoru tedy musí být stejný jako počet řádek matice.

Pro všechny vektorové operace zpravidla ve výsledku vystačí jediná SIMD instrukce. Pro násobení vektoru s maticí je instrukcí zpravidla tolik, kolik je řádků matice.

Oproti inicializačním konstrukcím jazyka C obsahuje GLSL „chytré“ konstruktory. Konstruktor pro n-rozměrný vektor tak může obsahovat n skalárních hodnot, potom každá složka vektoru bude naplněna odpovídající skalární hodnotou. Stejně tak může obsahovat n-1 rozměrný vektor a jednu skalární hodnotu, které budou tímto způsobem spojeny. Může obsahovat i n+1 rozměrný vektor, který bude tímto způsobem připraven o poslední složku. Obdobná pravidla platí i pro matice. Při inicializaci vektoru jedinou skalární hodnotou jsou všechny složky vektoru nastaveny na tuto hodnotu, při inicializaci matice jedinou skalární hodnotou jsou prvky matice na hlavní diagonále nastaveny na tuto hodnotu, ostatní jsou nastaveny na nulu.

Konstruktory jsou použity i jako přetypování, tedy nepíše se `(int)x` jako v C, ale `int(x)` jako v C++. Konstruktory mohou být tím pádem použity i ve výrazech.

Podporovány jsou pole i struktury, přístup k prvkům pole je přes standardní operátor [], přístup k prvkům struktury přes operátor tečky.

Zvláštní konstrukce, spojená s možnostmi grafického hardware je tzv. „swizzling“, tedy permutace prvků vektoru. Pro každý vektor lze sestavit masku, jež obsahuje permutaci jeho komponent. Jednotlivé prvky se mohou opakovat nebo se nemusí vyskytnout vůbec, rozměr výsledného vektoru nemusí být sejný jako rozměr zdrojového vektoru. Pro čtení z vektoru jsou povoleny všechny kombinace, pro zápis do vektoru jsou povoleny jen kombinace kde se prvky neopakují a jsou ve vzestupném pořadí. Masky jsou sestaveny z malých písmen, první verze GLSL umožňovala sestavit masky z písmen {x, y, z, w}, dnešní verze přidává {r, g, b, a} a {s, t, p, q}. Jejich chování v programu se nějak neliší, jsou určené pouze k logickému rozlišení obsahu vektoru (pozice, barva, souřadnice textury) pro lepší přehlednost programu. Nejlepší vlastností je, že permutace spolu s případnou negací nevyžaduje žádné instrukce navíc, tudíž lze je používat v neomezené míře, kdykoliv se to hodí. Následuje několik příkladů permutací:

```
vec4 v1, v2; // dvě proměnné typu vektor v R4
v1.x = 4; // nastaveni první složky v1 na hodnotu 4
v1.r = 4; // ekvivalentní zápis
v1.xzw = vec3(1, 2, 3); // nastavení x, z a w na 1, 2 a 3
v1 = v2.zxww;
v1.yzw = -v2.zyx; // další příklady možných permutací
```

GLSL obsahuje řadu vestavěných funkcí pro práci s vektory, maticemi, matematických funkcí a podobně. V neposlední řadě obsahuje funkce pro vzorkování textur.

Shader, napsaný v GLSL sestává z funkce `main()` a možná několika globálních proměnných, z nichž některé mohou být označené jako parametry a jejich hodnoty jsou inicializovány volajícím programem. Shader má určitou množinu vstupních proměnných ze kterých může číst a nějaké výstupní proměnné do kterých by měl zapsat (vertex shader by měl zapsat výslednou pozici vrcholu, fragment shader by měl zapsat výslednou barvu pixelu). Shader může definovat své vlastní funkce a volat je, úroveň maximálního zanoření je však nízká (shadery mají k dispozici různé aplikační limity přístupné jako globální proměnné).

Vertex shader může předávat své výpočty fragment shaderu, a to buď přes vestavěné proměnné, jako souřadnice textur, barva, normála nebo přes globální proměnnou, která je deklarovaná jako interpolovaná. Jednotlivé instance shaderů, zpracovávající pixely vedle sebe si nemohou předávat žádné hodnoty.

Jedno nepříjemné omezení je také na délku cyklů, která je limitovaná výrobcem a typem grafické karty. Zpravidla je to 256 průchodů. Cykly však mohou být vnořené, čímž se omezení dá většinou vcelku dobře obejít.

2 Realizace knihovny

Všechna obrazová data jsou uložena jako textury. Rozšíření OpenGL podporuje textury s rozměry jež nejsou mocniny dvou, což bylo dříve omezením. Toto rozšíření by mělo být přístupné všude, kde je přístupné programovatelné stínování. Textury jsou uloženy pouze bez mip-map, ty nejsou ve většině případů vůbec potřeba a jejich výpočet by jen zdržoval.

Dostupné barevné formáty textur by měly bohatě stačit našim potřebám, v OpenGL jsou dostupné jedno- až čtyřsložkové barvy (neuvažuje se použití indexovaných barev, které je dnes řídké). Co se týče datového typu, je možné použít celočíselné typy o šířce 4, 8, 12 a 16 bitů na kanál nebo nově i typy s plovoucí desetinnou čárkou o šířce 16 nebo 32 bitů na kanál. Celočíselné typy jsou vnitřně interpretovány jako hodnoty v rozmezí 0 až 1, při zápisu do textury s celočíselným typem je zapisovaná hodnota se saturací omezena do tohoto rozsahu. Typy s plovoucí desetinnou čárkou nejsou takto omezovány.

Všechny výpočty v shaderech jsou prováděny v 32-bitové nebo 64-bitové přesnosti, za použití čísel s plovoucí řádovou čárkou. Přesnost záleží na typu grafické karty, OpenGL ji výslovně nespecifikuje. Tudíž i přestože zdrojová data můžou mít nízkou přesnost celých čísel s nízkou šířkou, výpočet bude prováděn v nejvyšší možné přesnosti.

Čtení hodnot z textur zajišťují tzv. „samplery“, tedy „vzorkovače“. Vnitřní implementace je zcela závislá na výrobci a typu grafické karty. Program může pouze ovlivnit zda chce hodnotu nejbližšího pixelu, nebo bilineárně či trilineárně filtrovanou hodnotu. Globálně je možné aktivovat anizotropní filtrování, jež má za úkol lépe rozložit váhy pixelů, z nichž sestává vzorek na základě tvaru matice v níž vzorky leží. Pokud vzorky leží v pravidelné čtvercové matici, není zde rozdíl oproti bilineárnímu filtrování. Ale jakmile by vzorky vytvářely obdélníkové či kosodélníkové oblastí, váhy zdrojových pixelů se upraví tak, aby výsledná hodnota co nejvíce odpovídala vážené sumě barev pod čtyřúhelníkem vzorku. Tím je dosaženo lepší obrazové kvality při potahování prostorových objektů texturami.

2.1 Uložení obrazů do textur

Textury leží v paměti na desce grafické karty. Zdrojová data z obrázků či videa na pevném disku, nebo z proudu videa přicházejícího z kamery však budou uložena v systémové paměti počítače. K tomu knihovna musí obsahovat funkce pro rychlé přenosy dat mezi systémovou pamětí a pamětí grafické karty.

Dříve byly přenosy synchronní, tudíž systémový procesor musel čekat na dokončení přenosu, čímž vznikaly významné ztráty. Při zpracování objemných obrazových dat také vyvstává problém dlouhé doby přenosu a tím zpomalení výpočtu, dost možná nad dobu, jakou by výpočet zabral jsa spouštěn na systémovém procesoru. Pak by použití GPU zcela ztratilo smysl. Naštěstí OpenGL pomocí jistých rozšíření umožňuje realizovat asynchronní přenosy mezi paměťmi a tím pádem po dobu přenosu uvolní procesor pro další činnost. V některých případech může i samotný grafický akcelerátor provádět přenos na pozadí a zároveň se věnovat operacím v příkazové frontě. Konec přenosu se dá zjistit pomocí volání funkce, popřípadě se na něj dá počkat.

Bohužel, společnost ATI je dost pozadu s implementací rozšíření OpenGL do svých grafických karet, výsledkem je sice možnost asynchronních přenosů, ale bez možnosti zjistit kdy přenos skončil. Něco takového ovšem ztrácí pro praktické použití smysl.

Knihovna udržuje seznam textur a jejich asociace na obrazová data v systémové paměti. Je možné nastavit správu přenosů na automatickou, takže knihovna přenáší data kdykoliv je to potřeba, což může být vcelku neoptimální, pokud je voláno mnoho jednoduchých operací nad obrazem (proto knihovna přidává možnost vyhodnocovat výrazy nad obrazem, kdy přenosy optimalizuje vynecháním zbytečných přenosů a použitím asynchronních přenosů, kdykoliv je to vhodné). Automatická správa přenosů může být v jednom nebo v obou směrech přenosu zastavena a uživatel rozhodne sám, kdy je potřeba aktualizovat textury, nebo naopak jejich kopie v systémové paměti. Také sám rozhoduje, kdy použít přenos synchronní a kdy asynchronní.

2.2 Mapování grafické do systémové paměti

Knihovna podporuje ještě druhý způsob uložení obrázků, a sice textury mapované do systémového paměťového prostoru. Potom odpadá veškeré kopírování textur mezi systémovou a grafickou pamětí, v některých případech je zařídí grafický ovladač, v jiných případech je možné kopírování zcela vypustit a přistupovat ze systémového procesoru přímo do paměťového prostoru grafické karty. Takový způsob je jistě výkonnější a měl by být favoritem.

Tyto obrázky však mají tu nevýhodu že před použitím textury jako zdrojových dat, nebo jako cíl grafické operace, je třeba zrušit její paměťové mapování. Po dokončení operace jej lze samozřejmě obnovit, ale specifikace OpenGL neurčuje zda musí mapovaný obraz ležet ve stejné paměti jako předtím, tudíž pokud si programátor v C někam uloží adresu nějakých pixelů obrazu, po provedení operace s daným obrazem může taková adresa být neplatná. To není velký problém, pokud se s takovým chováním při používání knihovny počítá ale jelikož knihovna je rychlejší verzí již existující knihovny, může být použití tohoto typu obrazů při použití již existujícího kódu, využívajícího starší verzi knihovny, někdy problematické.

2.3 Realizace výpočtu

Výpočet operace nad obrazem je prováděn ve většině případů shaderem (pro některé operace jako sčítání, násobení nebo kombinace obrazů postačuje pevná funkcionalita). Všechny zdrojové obrazy jsou nastaveny do texturovacích jednotek, z nichž potom fragment shader bude moci vzorkovat pixely. Výstup je směřován do cílové textury, nebo do více cílových textur. Existuje rozšíření, umožňující zápis výstupních hodnot shaderů do více frame-bufferů. Tedy ne zápis kopie jediné výstupní hodnoty do více cílů, ale výstup několika hodnot, z nichž každá je zapsána do jednoho cíle. To může někdy významně urychlit výpočet, jež by jinak vyžadoval více průchodů.

Vlastní výpočet je spuštěn jednoduše nakreslením obdélníku, potaženého zdrojovými texturami. Vertex shader je zodpovědný za transformace souřadnic vrcholů obdélníka, může k nim dopočítat i nějaké hodnoty pro fragment shader. Ty potom budou, stejně jako například souřadnice textury, perspektivně korektně interpolovány po povrchu obdélníka. Fragment shader je poté spuštěn pro každý vykreslovaný pixel. Může číst připravené souřadnice, vzorkovat textury na libovolných souřadnicích, provádět výpočty a nakonec buďto zapsat hodnotu výsledného pixelu, nebo výsledek zahodit a ponechat původní hodnotu pixelu. Jak již bylo řečeno, fragment shader nemůže ovlivnit pozici kam bude výsledný pixel zapsán.

Po dokončení výpočtu je spuštěno kopírování výsledku z textury do systémové paměti, pokud je to potřeba a pokud je to kýžené chování (automatické přenosy nebyly ve směru z grafické do systémové paměti zakázány).

Existuje jedno omezení, a sice není možné použít jednu texturu jako zdroj dat a zároveň jako cíl kreslení. K tomu je potřeba vytvořit novou dočasnou texturu, což však není velká režie protože po dokončení operace není potřeba obsahy textur kopírovat, stačí textury přehodit v tabulce asociací s obrazy, uloženými v systémové paměti.

2.4 Struktura knihovny

Knihovna má dvě úrovně zpracování obrazu. První, nižší je sada operací nad obrazy z nichž většina jsou operace atomické.

Druhá, vyšší úroveň je funkce pro zpracovávání výrazů s obrazy. Ta je jednoduchá na použití, přeložené výrazy jsou ukládány do zásobníku a pokud je již přeložený výraz v zásobníku nalezen, znovu se nepřekládá. Navíc si sama optimalizuje paměťové přesuny obrázků, takže je rychlá a zároveň jednoduchá na použití. Tato úroveň je pouze nadstavba, volá funkce, poskytované nižší úrovní.

Knihovna se dělí na fasádu, tedy funkční rozhranní, implementaci algoritmů v C a systém volání shaderů v OpenGL. Fasáda jsou jen malé funkce, jež se rozhodují zda zavolat OpenGL implementaci nebo referenční implementaci v C.

Implementace algoritmů v C je jednoduchá, avšak každá funkce musí obsahovat několik verzí pro různé barevné typy obrázků, většinou nepodporuje všechny možné kombinace typů ale jen některé, které lze běžně očekávat. Oproti GPU implementaci má omezení na velikost obrazu, a sice velikost vstupních i výstupních obrazů musí být stejná, jinak funkce skončí aniž by se provedla. GPU implementace používá vzorkování textur, kde je nativně bilineární filtrování zdarma. Všechny vstupní obrazy jsou tedy automaticky převzorkovány na velikost výstupního obrazu, pokud se jejich velikosti liší.

Implementace algoritmů na GPU zahrnuje množství kódu, jež slouží jako rozhraní mezi fasádou knihovny a OpenGL API. Pro optimalizaci používá zásobník stavů OpenGL (zamezuje zbytečnému volání příkazů jako `glEnable` a jemu podobných). Dále obsahuje kód pro vlastní inicializaci OpenGL a pro nalezení vstupních bodů funkcí použitých rozšíření. Následují moduly pro zobecnění implementace kreslení do textury a pro kompilování, management a spouštění shaderů. Tato implementace kvůli některým omezením OpenGL, které v C implementaci nevyvstaly potřebuje relativně velké množství dočasných objektů, aby se zamezilo jejich opakovanému alokování a uvolňování, je zde několik resource managerů kteří se starají o vytváření, vracení a znovupoužití takových objektů. Patří sem dočasné textury pro uchování mezivýsledků nebo pro případy kdy je výstupní obraz zároveň vstupním a tím pádem do něj nelze kreslit přímo. Pro kreslení do textury jsou potřeba tzv. framebuffer objekty (FBO), které se také ukládají. A nakonec, pro minimalizaci inicializačního času knihovny a jejích paměťových nároků, jednotlivé shadery se nahrávají a kompilují podle potřeby, takže je také potřeba držet seznam shaderů v paměti, což velice dobře obstará stejný manager.

2.4.1 ImageStruct

ImageStruct je struktura pro ukládání obrazových informací, původně definovaná knihovnou DigiLib. DigiLib jako taková definuje rozhraní pro vytváření a správu obrázků, ale neobsahuje vlastní grafické operace. Moduly s grafickými operacemi byly sice napsány ale většina z nich není volně dostupná. Cílem této práce bylo vytvořit rychlý modul s operacemi. Zdrojový kód struktury je uveden viz. příloha 1.

ImageStruct umožňuje uložit v paměti obrázek o libovolné velikosti, s několika datovými typy pro uložení pixelu a s volitelným zarovnáním pixelů a volitelným posuvem obrazových řádků. Tím lze definovat obrázky s prokládáním, tedy takové jež mají mezi hodnotami jednotlivých pixelů mezery, stejně tak i mezi daty jednotlivých obrazových řádků. Obrazové řádky navíc mohou následovat v pořadí odshora dolů, nebo opačně. OpenGL však vyžaduje pixely, zarovnané na jeden až čtyři byte, přičemž směr obrazových řádků je seshora dolů. Maximální velikost obrázků je definována implementací OpenGL, v řadě GeForce FX je obvyklá hodnota 4096 × 4096 pixelů, GeForce 8800 zvyšuje hranici na 8192 × 8192 pixelů.

Knihovna s podobnými situacemi sice počítá, avšak pro optimální výkon by obrázky měly být bez prokládání, jinak přesuny obrazových dat z a do grafické paměti můžou trvat neúměrně dlouho.

Obrázky, mapované do paměti grafické karty jsou vytvářené funkcí knihovny, jejíž vstupy jsou pouze formát pixelů obrázku a jeho rozlišení. Zarovnání pixelů a obrazových řádek je potom zvoleno tak, aby souhlasilo s formátem OpenGL. Výstupem je prázdný obrázek, jehož ukazatel `Raster` ukazuje do mapované grafické paměti.

Podporované typy obrázků jsou `Image8`, `Image16`, `Image32` (avšak v grafické paměti jsou uložené jako obrázky s plovoucí řádovou čárkou, takže může docházet k nepřesnostem, jejich přenosy také trvají déle, neboť je nutné provádět konverzi dat), `ImageFloat`, `ImageRGB` a `ImageComplex`. Výpočty s komplexními obrázky jsou udělané tak, aby odpovídaly algebraickému komplexnímu počtu.

2.4.2 Konverze obrazových typů

OpenGL při vzorkování textury automaticky převádí hodnotu pixelu v daném typu textury. U jednokomponentových pixelových typů (neboli stupně šedi) se barva převádí na šeď s alpha kanálem nastaveným na 1.

U typu ImageRGB, který však obsahuje i alpha složku, se počítá s platným obsahem alpha kanálu. Při konverzi z barevného obrazu do stupňů šedi je použit pouze červený kanál, ostatní kanály jsou odříznuty.

Typ ImageComplex je vždy ošetřován jako komplexní číslo, při konverzi z komplexního čísla do RGB nebo do stupňů šedi je imaginární část odříznuta a výstupní hodnota odpovídá šedi s jasem reálné hodnoty.

2.4.3 Omezení referenční „C“ implementace

Referenční implementace knihovny je udělaná tak, aby co nejvěrněji odpovídala chování knihovny, běžící na GPU. Jediným rozdílem je nemožnost použití tak komplikovaného vzorkovacího schématu jako na GPU, proto jsou obrázky vzorkovány způsobem „nejbližší pixel“ takže pokud velikost vstupních a výstupního obrázku nesouhlasí, výstup bude převzorkován poněkud nekvalitně.

Další omezení plyne z náročnosti napsat v jazyce C efektivní konverzi pixelových hodnot ve stylu OpenGL. Každá funkce pro zpracování obrazu by musela obsahovat velký počet větvení, pro všechny možné kombinace vstupního a výstupních typů obrazů. Proto jsou implementovány „rychlé“ varianty pro očekávané typy pixelů, ostatní varianty jsou potom ošetřeny pomocí poněkud pomalejší konverze. Tím se počet větví programu v každé funkci značně sníží.

3 Sada implementovaných operací

Následuje popis implementovaných grafických operací, spolu s jejich omezeními a problémy realizace pomocí jazyka GLSL.

3.1 Aritmetické operace

Aritmetické operace jsou ekvivalentem klasické algebry, jen místo s čísly pracují s obrazy. Obrazy jsou zde nahlíženy jako sady hodnot (bodů), operace jsou potom prováděny nad jednotlivými hodnotami. Pokud se jedná o binární operaci, jako například součet obrazů, sčítány jsou hodnoty které by při položení obrazů na sebe ležely na stejném místě.

3.1.1 Součet dvou obrazů

Funkce sčítání obrazů `ImageImageAdd` provede jednoduchý součet hodnot pixelů. Pokud pixely obsahují větší počet komponent, provede se součet mezi odpovídajícími komponenty.

Při součtu hodnoty pixelu ve formátu čísla s plovoucí čárkou s hodnotou typu celé číslo je první interpretováno jako jeho hodnota, celé číslo je interpretováno jako hodnota v rozsahu 0 až 1, kde 0 odpovídá hodnotě nula a 1 odpovídá maximální hodnotě celého čísla, plynoucí z jeho šířky.

Pokud je výsledek součtu záporný, nebo naopak větší než jedna a má být zapsán jako celé číslo, je omezen do intervalu <0, 1> a vynásoben maximální hodnotou celého čísla. Pokud má být zapsán jako reálné číslo, není podroben žádným úpravám.

3.1.2 Rozdíl, součin a podíl dvou obrazů

Pro rozdíl obrazů je zde funkce `ImageImageSub`, pro součin `ImageImageMult` a pro podíl `ImageImageDiv`. Platí zde stejná pravidla jako při sčítání.

Pokud je při operaci dělení děleno nulou a výsledek je zapisován jako reálné číslo, výsledkem může být buďto ± nekonečno (dělenec byl nenulový), nebo hodnota NaN (angl. „not a numer“, není číslo). Pokud je hodnota zapisována do celého čísla, je nekonečno zapsáno jako maximální hodnota (což ostatně vychází z politiky ořezávání výsledků do intervalu 0 - 1) a NaN je zapsáno jako nula.

3.1.3 Vážený součet obrazů

Pro realizaci váženého součtu obrazů (angl. „blending“) existují dvě funkce, jež se liší zdrojem poměru. První funkce `ImageImageBlend` má váhy obrazů zadané jako dva parametry, výsledek odpovídá součtu obrazových hodnot, vynásobených odpovídajícími vahami.

Druhá funkce ImageImageDecall bere váhu z alpha kanálu druhého obrazu a sčítá obrazy v poměru:

$$y = x_1(1-\alpha_2)+x_2\alpha_2 \tag{1}$$

Je tedy zřejmé že druhý obraz by měl obsahovat alfa kanál. Pokud jej neobsahuje, konverze doplní hodnotu alpha = 1, takže výsledkem bude druhý obraz.

3.1.4 Inverze hodnot obrazu

Inverze hodnot obrazu je jednoduchá operace, při níž se odečtou hodnoty jednotlivých pixelů od jedné. Funkce pro inverzi obrazu se jmenuje `ImageInvert`.

3.1.5 Minimum a maximum ze dvou obrazů

Jde o funkce, vybírající do výsledného obrazu maximální (minimální) hodnoty z jednotlivých obrazů. Pokud mají pixely více složek (barevné nebo komplexní obrázky), je extrémní hodnota vybírána pro každou složku zvlášť. Funkce se jmenují `ImageImageMin` a `ImageImageMax`.

3.2 Bodové operátory

Bodové operátory jsou funkce, pracující s hodnotami jednotlivých bodů obrazu. Na rozdíl od aritmetických operací však jde o unární funkce.

3.2.1 Operátor prahování

Prahování je prováděno funkcí `ImageThresh`, výsledkem jsou hodnoty 0 pokud je hodnota pixelu pod prahem nebo 1 pokud je hodnota větší nebo rovna nastavenému prahu. Práh je zadáván jako reálné číslo, jeho hodnota by měla sledovat stejnou sémantiku jako při konverzi hodnot pixelů. Tedy například pro prahování na polovinu jasu je hodnota vždy 0.5. Hodnoty větší než 1 mají smysl pouze jedná-li se o prahování obrazu s hodnotami v plovoucí desetinné čárce.

3.2.2 Operátory nelineárního průběhu

Do této skupiny patří několik operátorů, jež provádějí nad každou složkou obrazových hodnot nějakou funkci s obecně nelineárním průběhem.

Mezi základní funkce patří logaritmický a exponenciální průběh prováděný funkcemi `ImageLog` respektive `ImageExp`. Jedná se o přirozené funkce, jejich základem je tedy Eulerovo číslo.

Další užitečné funkce jsou odmocnina a obecná mocnina. Jejich názvy jsou `ImageSqrt` a `ImagePow`. Při odmocňování záporného čísla vzniká hodnota NaN jež se, jak již bylo řečeno dříve, konvertuje na nulu. Komplexní čísla jsou odmocňovány bez vzniku chybových hodnot.

3.2.3 Operátor absolutní hodnoty

Výpočet absolutní hodnoty se spouští funkcí `ImageAbs`.

3.2.4 Jas a kontrast

Jas a kontrast je v podstatě lineární transformace obrazových hodnot. Lineární transformací se rozumí posun obrazových hodnot po přímce. Rovnice vypadá takto:

$$y = a + bx \tag{2}$$

Kde x je hodnota vstupního pixelu, y je výsledná hodnota a a, b jsou parametry. Odpovídající funkce se jmenuje `ImageScale`, parametry rovnice jsou opět zadávány jako reálná čísla.

3.2.5 Operátor transformace barev maticí

Tento operátor bere za vstup obrázek a klasickou transformační matici čtvrtého řádu. Vezme každý pixel obrázku, převede jej na vektor čtyř hodnot, vynásobí jej danou transformační maticí a výsledkem je jedna výstupní hodnota.

Transformace barev maticí je vhodná při převodu do jiného barevného systému, například pro převod z barevného systému RGB do systému YC_BC_R (použito v televizním systému PAL, v JPEG kompresi) lze použít následující matici:

$$M = \begin{bmatrix} 0.299 & 0.587 & 0.114 & 0 \\ -0.168736 & -0.331264 & 0.5 & 0.5 \\ 0.5 & -0.418688 & -0.081312 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Sloupce jsou násobeny hodnotami *R, G, B* a *A*, výsledkem jednotlivých řádků je *Y, C_B, C_R* a 1. Počítá se hodnotou složky $A = 1$. Odpovídající funkce nese název `ImageColorMatrix`.

3.3 Geometrické operace

Mezi geometrické operace patří zvětšování, rotace, zrcadlení a posun obrazu. Zde bylo použití bilineárního filtrování zváženo jako nedostatečné a bylo implementováno bikubické filtrování, jež dává ostřejší obraz který při větším zvětšení vypadá méně zubatý.

Bikubický filtr vzorkuje šestnáct vzorků, mezi nimiž interpoluje dvěma funkcemi:

$$\begin{aligned} w_1(x) &= (A+2)x^3 - (A+3)x^2 + 1 \\ w_2(x) &= Ax^3 - 5Ax^2 + 8Ax - 4A \end{aligned} \tag{4}$$

Parametr funkcí x je subpixelová odchylka z níž se počítají jednotlivé váhy. Koeficient A ovlivňuje ostření filtru. Bikubický filtr má pevně nastaveno $A = 0.75$. Vzorkované pixely jsou váženě sečteny ve čtyřech řádcích napřed horizontálně, čímž vzniknou čtyři hodnoty které jsou následně váženě sečteny vertikálně a výsledkem je barva vzorku. Váhy pro čtyři pixely se spočítají jako:

$$w = \{w_2(1+x), w_1(x), w_1(1-x), w_2(2-x)\} \tag{5}$$

Kde w je výsledný vektor vah, x je subpixelová odchylka souřadnice pixelu (pokud je v prostoru souřadnic obrázku 1 velikost pixelu, subpixelová odchylka je její desetinná část).

Pro geometrické operace jsou definovány čtyři funkce. Počínaje zvětšováním obrázku, `ImageResize`, jež změní velikost zdrojového obrázku tak, aby přesně vyplnil cílový. Není tedy zadáván žádný parametr poměru zvětšení.

Druhou funkcí je rotace obrázku. Parametrem je úhel v radiánech a způsob rotace. Obrázek může být rotován buďto tak, že velikost není změněna a obrázek bude mít oříznuté rohy nebo naopak,

velikost je přizpůsobena aby výsledný obraz obsahoval celý otočený obdélník vstupního obrazu. Pokud má výstupní obrázek jiné rozlišení než vstupní, je vstupní obrázek zvětšen v odpovídajícím poměru. Jméno funkce je `ImageRotate`.

Třetí, obecnější funkcí je `ImageTransform`. Té jsou zadány relativní 2D souřadnice čtyř rohů vstupního obrazu v prostoru výstupního obrazu, je vykreslen čtyřúhelník, obsahující interpolovaný vstupní obraz.

Poslední funkcí pro geometrické transformace je obecnější varianta předchozí funkce, jež provádí afinní transformaci obrazu. Název funkce je `ImageAffineTransform`, jejími parametry jsou opět čtyři souřadnice rohů vstupního obrazu, avšak tentokrát obsahují i třetí dimenzi. Čtyřúhelník je potom vstupním obrazem potažen perspektivně korektně.

Pro potřeby referenční implementace v jazyce C byl implementován jednoduchý rasterizér, jehož omezením je konvexnost vykreslovaných útvarů, stejně jako v OpenGL. Vzorkovací funkce textury je tentokrát identická s OpenGL, kvalita by tedy neměla být horší. Kvůli časové náročnosti výpočtů byl použit inline assembler, urychlující výpočet instrukční sadou MMX.

Obrázek 4: Porovnání bikubického filtru (vlevo) a bilineárního filtru (vpravo).

3.4 Morfologické operátory

Morfologické operace obvykle pracují nad binárními obrázky, avšak stejné operátory jsou navrženy i pro šedotónové obrázky. Implementace knihovny se k obrázkům chová jako k binárním, šedotónová hodnota je prahována s hodnotou jedna polovina. Při morfologických operacích nad vícekanálovými obrazy (barevné a komplexní) je každý kanál ošetřen jako individuální vrstva obrazu.

3.4.1 Eroze

Eroze je jednou ze dvou základních morfologických operací, druhou je dilatace. Operace eroze si bere za své vstupy zdrojový obraz a strukturní element. Strukturní element je binární maska, při výpočtu eroze je okno se strukturní maskou posouváno nad všemi možnými pozicemi ve zdrojovém obraze a výstup pod středem okna je vysoká hodnota právě tehdy, pokud pod každým pixelem strukturního elementu s vysokou hodnotou leží pixel zdrojového obrazu s taktéž vysokou hodnotou. Jinak je výstupem nízká hodnota (pixel erodoval).

Obecným efektem eroze je zmenšování oblastí s vysokými hodnotami a růst případných děr uvnitř. Dá se využít například při počítání objektů, kdy je potřeba oddělit slité tvary počítaných objektů.

Implementované operace eroze počítají se dvěma tvary strukturních elementů, buďto obdélníkový strukturní element s rozměrem $m \times n$, nebo element ve tvaru diamantu, vepsaného do čtverce o rozměru $m \times m$. Účelem diamantového strukturního elementu je připomínat kruh. Poslední variantou je strukturní element, zadaný jako obraz. Funkce realizující operaci eroze jsou `ImageSquareErode`, `ImageDiamondErode` a `ImageImageErode`.

3.4.2 Dilatace

Dilatace je operace, opačná k erozi, avšak aplikace eroze a poté dilatace, nebo naopak, nevrátí původní hodnoty - operace jsou de facto ztrátové. Na kombinaci obou operací jsou založeny doplňkové morfologické operace. Dilatace opět pracuje s obrazem a strukturním elementem, strukturní element je opět posouván nad vstupním obrazem, výstupem je vysoká hodnota v případě že pod libovolným pixelem strukturního elementu s vysokou hodnotou leží pixel vstupního obrazu s taktéž vysokou hodnotou.

Obecným efektem dilatace je rozšiřování ploch s vysokými hodnotami, uzavírání děr v objektech.

Implementované operace dilatace podporují opět dva základní tvary strukturních elementů, buďto obdélníkový, s rozměrem $m \times n$, nebo element ve tvaru diamantu, vepsaného do čtverce o rozměru $m \times m$. Opět lze použít obraz jako strukturní element. Názvy funkcí pro dilataci jsou `ImageSquareDilate`, `ImageDiamondDilate` a `ImageImageDilate`.

3.4.3 Otevření

Otevření je definováno jako doplňková operace, jednoduše jako eroze následovaná dilatací. Operace musí být nutně prováděna ve dvou průchodech, avšak při implementaci na GPU je rychlejší volat funkci otevření něž volat po sobě obě operace jelikož to vyžaduje méně přepínání stavů. Dočasný obraz si knihovna alokuje automaticky, jeho typ bude stejný jako typ výstupního obrazu.

Funkce, pro otevření obrazu se jmenují `ImageSquareOpen`, `ImageDiamondOpen` a `ImageImageOpen`.

3.4.4 Uzavření

Uzavření je, stejně jako otevření, kombinací dilatace a následné eroze (tedy v opačném pořadí než otevření). Názvy funkcí jsou analogicky k funkcím pro operaci otevření `ImageSquareClose`, `ImageDiamondClose` a `ImageImageClose`.

3.4.5 Obecná morfologická transformace

Anglicky „hit-and-miss transform“, je jakýmsi zobecněním morfologických operací, jelikož pomocí ní mohou být realizovány. Strukturní element při erozi a dilataci obsahoval vysoké hodnoty, nízké byly interpretovány jako hodnoty s nulovým významem. Nyní strukturní element může obsahovat vysoké i nízké hodnoty, musí ale obsahovat i hodnoty s nulovým významem (ty jsou v implementaci realizovány jako hodnoty blízké polovině jasu). Strukturní element se opět posouvá nad všemi pozicemi v obraze, vysoká hodnota je výstupem v případě že se všechny významné hodnoty strukturního elementu rovnají obrazovým hodnotám pod nimi.

Zde nemá smysl implementovat normované typy strukturních elementů, k dispozici je tudíž pouze `ImageImageHitMiss`.

3.5 Obrazové filtry

Obrazové filtry jsou používány zejména jako pásmové propusti, tedy k odstranění vysokých frekvencí z obrazu (rozmazání), nebo k odstranění nízkých frekvencí z obrazu (hledání hran). Obraz může být filtrován ve frekvenčním nebo obrazovém prostoru. Filtrace ve frekvenčním prostoru odpovídá násobení, filtraci v obrazovém prostoru odpovídá operace konvoluce. Jelikož přepočet obrazu do frekvenčního spektra a zpět je časově náročná operace, jsou implementované operace obecně konvolucemi. U některých operací je maximální velikost konvolučního jádra omezena, u některých je neomezená.

3.5.1 Průměr hodnot pod oknem

Aritmetický průměr obrazových hodnot pod oknem obrazu je jednoduchým rozostřovacím filtrem, je implementován jako separovatelný dvouprůchodový filtr. Obraz je tedy napřed filtrován v horizontálním smyslu, výsledek je poté filtrován ve vertikálním smyslu. Tím se výrazně sníží náročnost na paměťové přenosy, jelikož při velikosti filtrovacího okna $m \times n$ je potřeba jen $m + n$ obrazových vzorků, zatímco při naivní implementaci by bylo potřeba $m \times n$ vzorků. Tím efektivně snížíme náročnost z $O(n^2)$ na $O(2n)$.

Funkce pro výpočet průměru obrazu se jmenuje `ImageMean`, jejími parametry jsou vstupní obraz a šířka a výška okna filtru. Maximální rozměr okna filtru je omezen na 256, což by mělo být pro naprostou většinu potřeb dostatečné. Větší rozměr okna lze napodobit více průchody operace.

Průměr může být použit pro odstranění vysokých frekvencí, tedy podle míry buď odstranění jemného šumu až po rozmazání obrazu.

3.5.2 Medián hodnot pod oknem

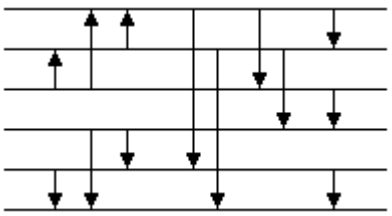
Nevýhoda průměrového filtru je neschopnost odstranit šum typu pepř a sůl, tedy řídce rozprostřené hodnoty s vysokými odchylkami od obsahu obrazu. Tento šum je lépe odstraněn mediánovým filtrem.

Podstatou filtru je nalezení střední hodnoty pod oknem filtru. Tím se vyloučí extrémní hodnoty jako již zmiňovaný šum pepř a sůl. Naproti průměrovému filtru nedochází k tak výraznému rozmazání, při větších velikostech okna však dochází k mírným deformacím.

Mediánový filtr je opět implementován jako separovatelný, avšak jeho efektivní implementace na GPU není tak snadná, jak by se mohlo zdát. Shader, běžící na GPU nemá k dispozici žádnou dynamicky alokovatelnou paměť, má jen relativně velký počet registrů, jež jsou indexovatelné. Pro implementaci tohoto filtru však nebyla zvolena metoda řazení hodnot v poli pomocí klasického řadícího algoritmu jako merge-sort nebo quick-sort, ale tzv. řadící sítě.

3.5.2.1 Řadící sítě

Řadící síť je abstraktní model sítě, sestávající z datových linek, propojených komparátory jež je použitá pro seřazení vstupní sekvence [8]. Síť je přitom navržena tak, aby co největší počet komparátorů mohl pracovat paralelně a síť tedy byla „co nejkratší“. Každý komparátor vezme hodnoty z obou vstupních linek a pokud rozhodne že jejich velikosti jsou obráceně než by měly, hodnoty navzájem vymění. Hlavní rozdíl mezi řadící sítí a řadícím algoritmem je fakt že řadící síť je stanovená předem a pořadí komparací je nezávislé na vstupních datech a výsledcích předchozích komparací.



Obrázek 5: Řadící síť pro šest prvků (směr šipek znázorňuje smysl komparace)

Fyzická řadící síť může být navržena pro jakýkoliv počet vstupních prvků, doba potřebná k seřazení je proporcionální k délce sítě (a je tedy pro různé vstupní sekvence konstantní) a využívá

paralelismu. V ideálním stavu kdy paralelní komparace opravdu probíhají paralelně dokáže seřadit vstupní sekvenci exponenciálně rychleji než běžný řadící algoritmus.

Řadící algoritmy, jež jsou implementovatelné jako řadící síť zahrnují bubble sort, sudé-liché transposition-sort, sudý-lichý merge-sort, bitonic-sort, a shell-sort. Knihovna používá řadící síť, odvozenou z algoritmu bitonic-sort [7]. Tento řadící algoritmus dává neklesající (nebo nestoupající) monotónní sekvenci, vstupem je „bitonní" sekvence, skládající se ze dvou subsekvencí, jedna z nich je monotónně neklesající, druhá z nich je monotónně nestoupající. Sekvence na první úrovni jsou jednotlivé vstupní hodnoty, dál se jejich délky zvětšují až vznikne seřazená sekvence původní délky.

3.5.3 Medián hodnot pod oknem (pokračování)

Jelikož řadící síť musí být navržena před vlastním výpočtem, aby mohla být zkompilována, je navrženo jen několik řadících sítí pro různé délky, takže mediánový filtr je k dispozici jen pro velikosti oken 3, 5, 7 a 15 pixelů.

Bylo dokázáno že bitonní řadící síť, implementovaná pomocí klasického procesoru je oproti klasickým řadícím algoritmům neefektivní jelikož komparátory nepracují paralelně. Pro n elementů taková řadící síť obsahuje $n \cdot log(n)^2$ komparátorů, zatímco merge-sort dosahuje minimální hranice $n \cdot log(n)$ porovnání. Spoléhá se tedy na fakt, že GPU bude umět párovat dostatečné množství operací.

Další fakt, mluvící pro řadící sítě je řazení více polí paralelně. To nastává v případě kdy jsou řazeny vektorové hodnoty (barevné a komplexní obrázky) jelikož jednotlivé barevné hodnoty jsou řazeny nezávisle na sobě. Díky své architektuře dokáží GPU takto řadit vektory o čtyřech komponentech pomocí dvou instrukcí (jedna instrukce „min" a jedna „max"). Tím se řazení značně zefektivní. Pro největší síť, obsahující 15 pixelů bude muset merge-sort provést $15log_2(15) = 60$ porovnání. Řadící síť bude muset provést čtyřikrát více. Avšak GPU spolehlivě párují operace s nezávislými operandy, takže rychlost řazení je v podstatě poloviční. Pokud k tomu přidáme fakt, že řazení probíhá na čtyřech polích současně, což pomocí merge-sortu není možné jinak než provést řazení čtyřikrát po sobě, je GPU implementace na počet potřebných komparací dvojnásobně rychlejší, nemluvě o výkonnostní převaze samotného grafického procesoru.

Funkce pro filtraci mediánovým filtrem se jmenuje `ImageMedian`, parametr vstupního obrazu doprovází ještě velikost okna filtru, jejíž povolené hodnoty jsou již zmíněné 3, 5, 7 a 15.

3.5.4 Konvoluční filtry

Jak již bylo řečeno, většina obrazových filtrů se dá realizovat pomocí operace konvoluce. Konvoluční filtr bere jako vstup obraz a konvoluční jádro (kernel). Jádro se potom posouvá po obraze a pod něj

se zapisuje hodnota součtu obrazových pixelů, vážených odpovídajícími hodnotami konvolučního jádra. Diskrétní konvoluce [9] je definována jako:

$$(f * g)(m) = \sum_{n} f(n)g(m-n) \tag{6}$$

Kde *f* je vstupní funkce (obrázek), *g* je konvoluční jádro, *m* je rozměr jádra, *n* je rozměr obrázku. Zde jde o jednorozměrnou konvoluce, dvojrozměrná konvoluce je velmi podobná, jediným rozdílem je počet parametrů funkcí *f* a *g*.

Konvoluční jádro může být dvojrozměrné, nebo může být rozloženo na dvě separovatelné funkce, kdy se operace konvoluce rozpadne na dvě po sobě jdoucí konvoluce, jedna v horizontálním směru a druhá nad jejím výsledkem ve směru vertikálním. Separovatelná konvoluce o stejném efektivním rozměru jádra je časově méně náročná než ekvivalentní dvojrozměrná konvoluce.

Konvoluce se na GPU dá provádět buďto pomocí shaderu, nebo pomocí tzv. „imaging subset“, tedy sady funkcí pro zpracování obrazu. Imaging subset má tu nevýhodu že jádro konvoluce je specifikováno jako pole hodnot, nikoliv jako obrázek (textura). Na druhou stranu, shader má omezení co do počtu průchodů cyklem takže pro vytvoření konvolučního filtru s neomezeným rozměrem jádra je zapotřebí překonat toto omezení. Pro implementaci konvoluce byl nakonec zvolen shader z důvodů nemožnosti implementace správné komplexní konvoluce.

Podporované operace konvoluce jsou separovatelná konvoluce a 2D konvoluce. Odpovídající funkce knihovny nesou následující názvy: `ImageConvolveSeparable` a `ImageConvolve2D`. Jejich vstupem je obraz, jež má být filtrován a obraz (u separovatelné konvoluce dva obrazy), jež bude použit jako jádro. Pro rozměry jádra není žádné omezení, krom omezení maximální velikosti textury. Je podporována i komplexní konvoluce (komplexní obraz, jádro nebo obojí).

3.5.5 Filtry hledání lokálních extrémů

Do této skupiny filtrů patří dva, filtr pro lokální minimum a maximum. Filtry hledají minimální, respektive maximální obrazové hodnoty v okně. Oba opět pracují ve stylu separovatelné konvoluce. Pro barevné nebo komplexní obrázky jsou maxima nalezena pro každý kanál zvlášť.

Funkce se jmenuji `ImageLocalMax`, `ImageLocalMin` a jejich parametrem je vstupní obraz, spolu s velikostí okna filtru. Maximální velikost je 256, což by opět mělo být více než dostačující.

3.6 Obrazové transformace

Předchozí operace vždy pracovaly ve smyslu kdy byla obrazová informace nějakým způsobem změněna nebo ovlivněna. Výstupy obrazové transformace však jsou často zcela jiné povahy a typu

informační hodnoty, než byl vstupní obraz. Na první pohled mezi vstupem a výstupem nemusí být žádná geometrická nebo jiná podobnost.

3.6.1 Fourierova transformace

Fourierova transformace [9] převádí obraz z prostorové oblasti do frekvenční oblasti, existuje i inverzní fourierova transformace, pracující opačným směrem. Diskrétní fourierova transformace je definována jako:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi k \frac{n}{N}} \qquad k = 0,1,...,N-1 \tag{7}$$

Inverzní fourierova transformace vypadá velice podobně:

$$x_n = \frac{1}{N} \sum_{n=0}^{N-1} X_k e^{-j2\pi k \frac{n}{N}} \qquad n = 0,1,...,N-1 \tag{8}$$

Dvojrozměrná fourierova transformace se dá odvodit, ale vzhledem ke složitosti se častěji využívá vlastnosti separovatelnosti transformace, tudíž dvojrozměrná transformace se dá rozložit na sadu jednorozměrných transformací řádků a poté sloupců. S inverzní transformací je to samozřejmě stejné.

Pro šedotónový obrázek je výsledkem obrázek o stejném rozměru, obsahující komplexní hodnoty. Vstupem může být i komplexní obrázek. Vstupem inverzní fourierovy transformace je komplexní obrázek, výstupem je opět šedotónový obrázek. Nebyl by velký problém spočítat fourierovu transformaci barevného obrázku, ale DigiLib ani OpenGL nedefinuje žádný datový typ, jež by mohl obsahovat tři komplexní hodnoty v jediném pixelu.

Oblíbenou variantou fourierovy transformace je FFT (angl. „fast fourier transform“), tedy rychlá fourierova transformace. Ta dává stejné výsledky jako diskrétní fourierova transformace, avšak její výpočet byl algebraicky upraven tak, aby neobsahoval žádné goniometrické operace a aby celkově proběhl co nejrychleji. K výpočtu rychlé fourierovy transformace je zapotřebí přepočítaná tabulka váhových faktorů, knihovna si takovou tabulku spočítá automaticky pokaždé když je potřeba a uschová ji v paměti pro pozdější potřebu.

Omezením rychlé fourierovy transformace je nutnost mít velikost vstupního (a výstupního) obrazu rovnu libovolné mocnině dvou. Existují i varianty, které dokáží rychle vypočítat i rychlou fourierovu transformaci obrazů s velikostí prvočíselných faktorů. Implementace však vyžaduje velikost mocnin dvou.

Funkce pro rychlou fourierovu transformaci se jmenují `ImageFFT` a `ImageIFFT`.

4 Zpracování výrazů s obrazy

Druhá vrstva knihovny obsahuje jedinou funkci, jež má tři parametry, prvním je výstupní obraz, následuje výraz a poslední je seznam obrazů, na něž se výraz odkazuje. Funkce se jmenuje `ImageExpression`, následuje popis gramatiky jež přijímá:

```
Obraz-id ::= %číslo
Bin-Op ::= + | - | * | /
Morph-Func ::= erode | dilate | open | close
Filter-Func ::= avg | median | min | max
Unary-Func ::= fft | ifft | invert | abs | log | exp | sqrt
Binary-Func ::= min | max
Obraz ::= Obraz-id
        | (Obraz)
        | Obraz Bin-Op Obraz
        | Unary-Func(Obraz)
        | Binary-Func(Obraz, Obraz)
        | Filter-Func(Obraz, velikost okna)
        | Morph-Func(Obraz, Obraz)
        | Morph-Func(Obraz, [square|diamond], velikost)
        | hit_miss(Obraz, Obraz)
        | pow(Obraz, mocnina)
        | scale(Obraz, bias, factor)
        | matrix(Obraz, m11, m12, m13, m14, m21, … , m34, m44)
        | resize(Obraz, šířka, výška)
        | rotate(Obraz, úhel, mód)
        | ilerp(Obraz, x1, y1, x2, y2, x3, y3, x4, y4)
        | ilerpa(Obraz, x1, y1, z1, x2, y2, z2,
                       x3, y3, z3, x4, y4, z4)
```

Funkce textově porovná výraz se dříve zpracovanými výrazy pro případ že už by existoval v podobě grafu. Pokud neexistuje, výraz je rozložen na tokeny a je sestaven graf. Graf je optimalizován a posléze proveden. Výsledek výrazu musí být typu `Obraz`.

5 Výsledky

V této kapitole jsou zhodnoceny dosažené výsledky, zejména z hlediska rychlosti výpočtů, jakožto jednoho z hlavních hledisek při zpracování obrazu v reálném čase.

5.1 Měření rychlosti výpočtů

Při provádění výpočtů na CPU rychlost odpovídá počtu pixelů, jež je procesor schopný zpracovat za sekundu, anglicky se tento údaj označuje termínem „fillrate“. Z takového údaje můžeme říci jak dlouho bude trvat zpracování obrazu o daném rozlišení. Na druhou stranu, u GPU existuje ještě několik dalších faktorů ovlivňujících rychlost výpočtu, jež nás při zpracování na CPU netrápí.

Jedním důležitým parametrem je rychlost přenosu ze systémové paměti do grafické paměti a zpět. Při zpracování proudu obrazu z kamery by ideálně byl přenos zdrojového snímku směrem tam a výsledku směrem zpět. Někdy ale výpočet potřebuje několik vstupních obrazů, nebo některé funkce na GPU nejsou dostupné a je potřeba používat referenční implementaci, pro kterou se obrázek musí stáhnout do systémové paměti, pokud tam ještě není, a poté opět nahrát zpět do grafické paměti. Obecně se snažíme vyhnout přenosům za každou cenou, neboť by byly zbytečnou penaltou rychlým funkcím zpracování obrazu. Měřením byly experimentálně zjištěny následující hodnoty:

Slot grafické karty	**AGP 8×**	**PCI-X 16×**
Rychlost synchronního přenosu do grafické paměti	92.65 MB/s	576.27 MB/s
Rychlost synchronního přenosu do systémové paměti	170.78 MB/s	753.42 MB/s
Rychlost asynchronního přenosu do grafické paměti	103.44 MB/s	1026.34 MB/s
Rychlost asynchronního přenosu do systémové paměti	273.39 MB/s	431.82 MB/s

Tabulka 1: Přenosové rychlosti grafických slotů AGP a PCI-X, NV6600 a NV7950

Údaje byly naměřeny na dvou různých počítačích, první byl osazen grafickou kartou NVidia 6600 v AGP 8× slotu, druhý obsahoval kartu NVidia 7950 ve slotu PCI-X 16×. Zajímavé je, že u novějšího PCI-X je asynchronní přenos z grafické paměti do systémové pomalejší, než synchronní. Výhodou však zůstává že CPU může po dobu přenosu vykonávat nějakou další činnost.

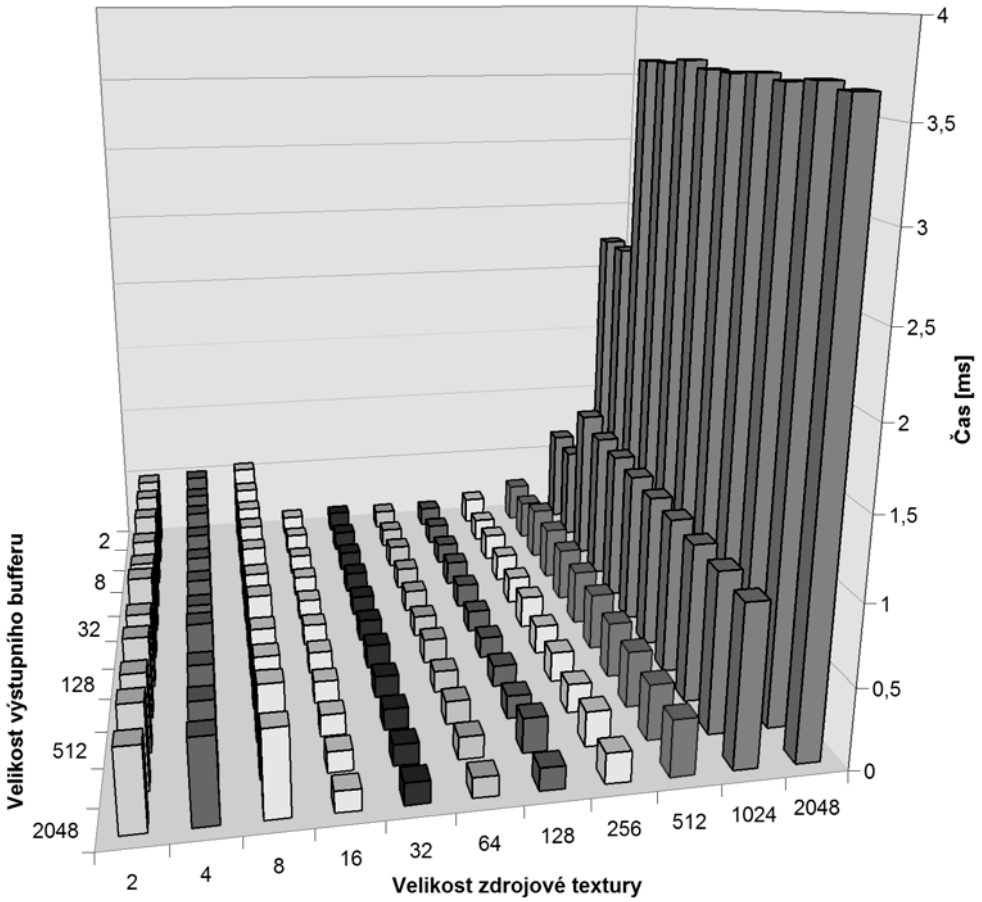
Knihovna alokuje své prostředky v okamžiku, kdy je potřebuje. Proto je dobré vědět jak dlouho trvá alokace používaných typů OpenGL objektů Objekty se nealokují pokaždé znova, knihovna má managery pro efektivní používání již vytvořených objektů. Knihovna používá textury pro uložení obrázků, framebuffer objekty pro kreslení do textury a shadery jako jednotlivé operace. Následující tabulka obsahuje průměrné hodnoty, naměřené na počítači s kartou NVidia 6600 v AGP slotu a procesorem Athlon FX 1.7GHz (na frekvenci procesoru záleží doba kompilace shaderů):

Akce	Průměrný čas	Komentář
Alokace textury	56 ns	256 × 256, 8 bit, RGBA
Nastavení aktivní textury	11 ns	
Vytvoření FBO	4.9 ms	256 × 256, 8 bit, RGBA
Nastavení FBO pro kreslení	4.1 ms	včetně nastavení cílové textury
Uvolnění FBO	0.12 ms	včetně uvolnění cílové textury
Kompilace shaderu v GLSL	84 ms	jednoduchý vertex a fragment shader (1.2 kB)
Nastavení aktivního shaderu	11 ns	
Nastavení paramteru shaderu	4 ns	Celočíselný parametr (sampler) nebo float.

Tabulka 2: Přibližné časy základních operací s OpenGL, NV6600

Dalším, již méně podstatným faktorem zpracování obrazu na GPU je čas nastavení kontextů před započetím vlastní operace. Ten se pohybuje v jednotkách milisekund.

Posledním parametrem, stejně jako při zpracování na CPU je množství pixelů, zpracovaných za sekundu. U GPU se to zřejmě bude mírně měnit s rozměrem použitých textur, větší textury způsobí horší efektivitu cache a tím pádem i delší čas zpracování. Následující graf zobrazuje závislost času kreslení (tzn. fillrate^{-1}) na rozlišení textury a na rozlišení cílového obrazu:



Obrázek 6: Závislost času kreslení na rozlišení textury a rozlišení výstupu, NV6600

Je zřetelně vidět že rychlost kreslení téměř nezáleží na velikosti výstupního bufferu, i když GPU muselo vykreslit větší obrazovou plochu, naopak záleží na rozlišení textury. Zvláštní ovšem jsou větší časy u velmi malých textur (2×2, 4×4 a 8×8).

5.2 Rychlost jednotlivých operací

Jako příloha 2 je uvedena tabulka, obsahující rychlosti jednotlivých operací, prováděných na GPU a CPU. Protože fillrate není příliš srozumitelná hodnota, je navíc dopočítána doba, jakou by trvalo zpracovat obrázek o velikosti 256x256 pixelů.

6 Závěr

Při realizaci této práce jsem využil znalosti rozhraní OpenGL a programování shaderů, podrobně jsem se seznámil s vnitřní strukturou grafických karet NVidia a optimalizacemi shaderů. Seznámil jsem se i s dalšími možnostmi práce s grafickými kartami (DirectX, CUDA). Vytvořil jsem funkční knihovnu pro zpracování obrazu v reálném čase, spolu s několika testovacími aplikacemi, pomocí nichž byly naměřeny hodnoty v sekci výsledky. Knihovna, včetně operací pro zpracování obrazu je napsaná v jazyce C, pro zpracování na GPU používá mnou již dříve vyvíjený ÜberLame framework.

Knihovna obsahuje funkce, srovnatelné s funkcemi knihovny GPUCV, zpracovávající obraz na GPU, jež je klonem knihovny OpenCV firmy Intel. Stejně tak i rychlost výpočtů je srovnatelná. Mezi výhody oproti GPUCV patří druhá vrstva knihovny, určená ke zpracování algebraických výrazů s obrazy, automatický ústup z GPU na CPU, pokud na daném cílovém stroji nejsou podporované potřebné funkce pro zpracování obrazu na GPU, nezávislost na operačním systému Windows nebo podpora asynchronních přenosů obrazů mezi grafickou a systémovou pamětí. Poslední výhodou je rozhraní, spolupracující s knihovnou DigiLib, jež je již používána. Zbývá poznamenat že knihovna GPUCV nebyla použita žádným způsobem jako reference, její existence byla zjištěna až při hledání srovnatelné knihovny pro porovnání výsledků.

Při porovnání výsledků mezi CPU a GPU je v naprosté většině GPU mnohem rychlejší, avšak je zde časová penalizace způsobená přenosem obrazu mezi systémovou a grafickou pamětí. Rychlost zpracování je dostatečná aby ji vyvážila, pokud jsou zpracovávána dostatečně velké objemy dat, popřípadě jsou použity složité funkce. Například osamocená operace sečtení dvou obrazů, prováděná nad malými obrazy, bude zřejmě vždy rychleji provedena na CPU.

Práce nebyla doprovázena mnoha problémy, snad jediným problémem byla nedostupnost novějšího hardware se kterým by knihovna mohla použít novější technologie, vhodnější pro některé grafické operace, tedy nové rozhraní CUDA. Tímto směrem by se mohl ubírat další vývoj knihovny, protože toto rozhraní lépe využívá vlastnosti GPU a celkově je vhodnějším nástrojem pro realizaci univerzálních výpočtů. Dalším námětem by mohla být knihovna, generující na základě algebraického výrazu s obrazy zdrojové kódy shaderů jež by provedly požadované operace s co nejmenším počtem operací. Zajímavé by taktéž mohlo být rozšířit stávající knihovnu o různé matematické operace jako maticovou algebru pro matice s velkým rozměrem, výpočty nad prostorovými daty, mnoharozměrnými integrály a podobně.

Literatura

[1] Žára, J. a kol.: Počítačová grafika principy a algoritmy. Grada, Praha, 1992.

[2] Žára, J. Beneš, B. Felker, P.: Moderní počítačová grafika, Computer Press, 1998

[3] Segal, M. Akeley, K.: The OpenGL® Graphics System: A Specification (Version 2.1), 2006

[4] Kessenich, J. Baldwin, D. Rost, R.: The OpenGL® Shading Language, 2006

[5] Nvidia Corporation: GPU Programming Guide, 2005

[6] Nvidia Corporation: NVIDIA CUDA Compute Unified Device Architecture, 2007

[7] Batcher, K. E.: Sorting Networks and their Applications. Proceedings AFIPS, 1968

[8] Knuth, D. E.: The Art of Computer Programming, Vol. 3. Addison-Wesley, 1973

[9] Bartsch, H. J.: Matematické vzorce, Mladá fronta Praha, 1996

(pozn.: u titulů kde není uveden nakladatel se jedná o jejich elektronickou podobu.)

Seznam příloh

Seznam použitých zkratek a symbolů

GPU - z anglického graphics processing unit, grafický procesor

FPGA - programovatelné hradlové pole, elektronická komponenta

CPU - z anglického central proceesing unit, systémový procesor počítače

GL, OpenGL - populární grafické api, z anglického open graphics library, otevřená grafická knihovna

GLSL - programovací jazyk, z anglického GL shading language, kde GL je referencí na OpenGL

CUDA - rozhrani pro ovládání grafických akcelerátorů

NV - NVidia, zkrácený název společnosti

FBO - anglicky framebuffer objekt, cíl kreslení v OpenGL

AGP - akcelerovaný grafický port, grafická sběrnice

PCI-X - novější grafická sběrnice, nástupce AGP

(I)FFT - (inverzní) rychlá fourierova transformace

RGB(A) - barevný systém, červená modrá zelená (alpha)

YC_BC_R – jiný barevný systém, rozkládající barvu na luminanci (Y) a chrominanci (C_B, C_R)

2D / 3D - označuje dvoudimenzionální nebo třídimenzionální prostor

Příloha 1. Zdrojové kódy ImageStruct

```
typedef struct ImageStruct {
  unsigned char * Raster;
  char * Info;
  void * BinInfo;
  int BinInfoSize;
  void *ExtraInfo;
  int XSize;
  int YSize;
  int XOffset;
  int YOffset;
  int InfoSize;
  short PixelType;
  short ImageType;
  int References;
  union {
    struct {
      void (* Free)(void *);
    } Regular;
    struct {
      void (* Free)(void *);
      struct ImageStruct * Referenced;
    } Reference;
    struct {
      void (* Free)(void *);
      void (* FreeData)(void *);
      void * Data;
    } External;
  } Optional;
} ImageStruct;
```

Převzato z knihovny DigiLib.