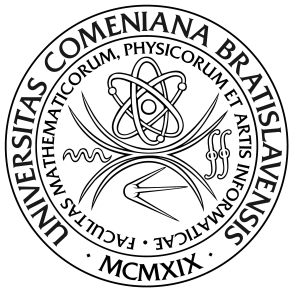
Negrafické výpočty na GPU v prostredí OpenGL

Dizertačná práca

RNDr. Michal Červeňanský



Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Katedra aplikovanej informatiky

Školiteľ: Prof. Ing. Miloš Šrámek, PhD.

Komisia pre obhajoby: 9.2.1. Informatika

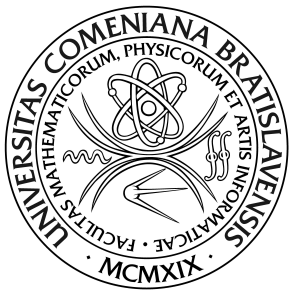
Stupeň odbornej kvalifikácie: doktor filozofie (PhD.)

Bratislava 2010

Non-graphics computations on GPU in the OpenGL environment

Dissertation Thesis

RNDr. Michal Červeňanský



Comenius University in Bratislava

Faculty of Mathematics, Physics and Informatics

Department of Applied Informatics

Supervisor: Prof. Ing. Miloš Šrámek, PhD.

Branch: 9.2.1. Informatics

Academic degree: Doctor of Philosophy (PhD.)

Bratislava 2010

Čestné vyhlásenie:

Čestne vyhlasujem, že dizertačnú prácu som vypracoval samostatne, na základe vlastných vedomostí a použitej literatúry.

. . . . . .
Michal Červeňanský

Poďakovanie

Úprimne ďakujem svojmu školitelovi Prof. Ing. Milošovi Šrámekovi, PhD. za jeho odborné vedenie, cenné pripomienky a motiváciu pri tvorbe tejto práce.

Vďaka patrí aj mojej rodine a priateľke za ich pochopenie a podporu, ktorú mi poskytli.

Michal Červeňanský

Abstrakt

ČERVEŇANSKÝ Michal. Negrafické výpočty na GPU v prostredí OpenGL [dizertačná práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra aplikovanej informatiky. Školiteľ: Prof. Ing. Miloš Šrámek PhD. Komisia pre obhajoby: 9.2.1. Informatika. Stupeň odbornej kvalifikácie: doktor filozofie (PhD.). Bratislava: FMFI UK, 2010. 91 s.

V tejto práci sa zaoberáme využitím moderných grafických akcelerátorov (GPU) v rôznych oblastiach spracovania obrazových a objemových dát. Dnešné GPU ponúkajú oproti bežným CPU vysoký výkon a poskytujú aj rozsiahlu funkcionalitu a programovateľnosť v OpenGL prostredí. Vďaka tomu sa tieto vlastnosti dajú využiť aj mimo hlavného určenia GPU—počítačovej grafiky.

GPU využívame v oblasti spracovania objemových dát prúdovým prístupom, ktorý efektívne využíva dostupné pamäťové prostriedky počítača. Zameriavame sa hlavne na spracovanie po rezoch. Predstavíme všeobecné techniky, ktoré je možné využiť pri spracovaní objemových dát na GPU. Na ich demonštráciu sme implementovali viaceré lokálne filtre spracovávajúce objemové dáta. V syntetických a reálnych testoch sme dosiahli výrazné urýchlenia oproti bežným výpočtovým na CPU.

Ďalšou oblasťou využitia GPU sú paralelné dátovo závislé triangulácie. Výpočet dátovo závislých triangulácií je časovo náročný optimalizačný proces, a tak sme pre potreby využitia GPU navrhli paralelný algoritmus, ktorý vyhovuje obmedzeniam GPU. Výsledné dátovo závislé triangulácie využívame pri rekonštrukcii obrazu, konkrétne pri zväčšovaní. Predstavíme rôzne modifikácie základného algoritmu, ktoré zvyšujú výslednú kvalitu rekonštrukcie s nízkym dopadom na výpočtový čas. Z výsledkov a meraní vyplýva, že využitie dátovo závislých triangulácii je vhodnou voľbou pri rekonštrukcii obrazu. S využitím paralelného algoritmu na GPU je tento výpočet časovo výhodnejší, pričom sme dosiahli 8-násobné urýchlenie oproti neparalelnému algoritmu na CPU.

KĽÚČOVÉ SLOVÁ: GPGPU, dátovo závislé triangulácie, prúdové spracovanie objemových dát

Abstract

ČERVEŇANSKÝ Michal. Non-graphics computations on GPU in the OpenGL environment [dissertation thesis]. Comenius University in Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Applied Informatics. Supervisor: Prof. Ing. Miloš Šrámek, PhD. Branch: 9.2.1. Informatics. Academic degree: Doctor of Philosophy (PhD.). Bratislava: FMFI UK, 2010. 91 p.

In this thesis we describe the usage of modern graphics accelerators (GPU) in various scientific fields. Today's GPUs offer high performance compared to a standard CPU and provide extensive functionality and programmability in the OpenGL environment. These features can be used also outside of rendering, which is the main domain of GPUs.

We use the modern GPUs in volume data processing based on streaming which efficiently utilizes the available computer memory resources. Here, we focus on the slice-based streamed processing. We present a general technique that can be used with volumetric data on the GPU in various applications. To demonstrate the technique we implemented several streamed local filters. We achieved a significant speed-up compared to the standard CPU computations in tests with both synthetic and real tomographic data.

The next area of GPU usage is data-dependent triangulation which involves a time-consuming optimization process. For a GPU implementation we proposed a parallel algorithm which satisfies the GPU constraints. The resulting triangulation is used in image reconstruction, particularly in image magnification. We introduce various modifications of the basic algorithm which improve the final quality of the reconstruction with low impact on computation time. The results and measurements show that the use of data-dependent triangulation is a good choice for image reconstruction. The parallel version of the algorithm is 8-times faster in comparison to a non-parallel version of the CPU algorithm.

KEYWORDS: GPGPU, data-dependent triangulations, stream processing of volume data

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam skratiek

ARB Zoskupenie firiem (Architecture Review Board)

Cg Programovací jazyk na programovanie výpočtových jednotiek GPU (C for Graphics)

CPU Centrálny procesor (Central Processing Unit)

CT Počítačová tomografia (Computer Tomography)

CUDA Knižnica na využívanie GPU (Compute Unified Device Architecture)

DDT Dátovo závislá triangulácia (Data-dependent Triangulation)

FBO Časť pamäte na GPU (Frame Buffer Object)

glsl Programovací jazyk na programovanie výpočtových jednotiek GPU (OpenGL shading language)

GPGPU Všeobecné výpočty na grafických procesoroch (General-Purpouse Computation on Graphics Procesing Unit)

GPU Grafický procesor (Graphics Processing Unit)

GSL Knižnica na vedecké výpočty (GNU Scientific Library)

HLSL Programovací jazyk na programovanie výpočtových jednotiek GPU (High Level Shading Language)

ID Identifikátor (Identification)

MRI Magnetická rezonancia (Magnetic Resonsnce Imaging)

MRT Viacnásobné renderovacie oblasti (Multiple Render Targets)

MSE Stredná kvadratická chyba (Mean Square Error)

MWT Triangulácia s minimálnou cenou (Minimal Weight Triangulation)

OpenCL Knižnica na využívanie paralelných systémov (Open Computing Language)

OpenCV Knižnica počítačového videnia (Open Computer Vision)

OpenGL Otvorená grafická knižnica na využitite GPU (Open Graphics Library)

PBO Časť pamäte na GPU (Pixel Buffer Object)

PSNR Špičkový odstup signál-šum (Peak to Signal Noise Ratio)

ROI Región vplyvu (Region of Influence)

SCF Sederbergova cenová funkcia (Sederbergs Cost Function)

SLI Škálovatelné prepojenie grafických kariet (Scalable Link Interface)

SNR Odstup sinál-šum (Signal to Noise Ratio)

SSE Inštrukčná sada CPU (Streaming SIMD Extensions)

SSIM Index štrukturálnej podobnosti (Structural Similarity Index)

UIQI Univerzálny koeficient kvality obrazu (Universal Image Quality Index)

Voxel Element objemových dát (Volume element)

Kapitola 1

Úvod

Moderné GPU sú v dnešnej dobe veľmi rozšírené a vyskytujú sa takmer v každom bežnom počítači. Ich výkon enormne vzrástol a presiahol aj výkon bežných CPU. So vzrastajúcim výkonom sa rozšírila aj funkcionalita GPU a hlavne programovateľnosť. Táto programovateľnosť nie je úplne bez obmedzení ako je to pri CPU, ale je dostatočne flexibilná, aby aj v tomto atribúte konkurovala CPU. Tieto výrazné vylepšenia GPU sú spôsobené neustálym tlakom na výrobcov najmä zo strany vývojárov počítačových hier. Výkonnejšie GPU sú potrebné pre nové hry renderujúce obraz vo vysokej kvalite, niekedy ťažko rozoznateľnej od skutočnej reality. Na druhej strane, konkurenčný boj výrobcov GPU umožňuje udržať cenu na takej úrovni, aby výkonne GPU mohlo byť dostupné širokej verejnosti. S rozširujúcou sa funkcionalitou a výkonom sa GPU začali využívať aj na výpočty rôzneho charakteru, nie len na renderovanie trojrozmerných scén, na čo sú primárne určené. Takéto využitie GPU je veľmi populárne z dôvodu niekoľkonásobného urýchlenia výpočtov oproti CPU, niekedy o dva alebo aj viac rádov. Využívanie výkonu GPU je populárne najmä vo vedeckých aplikáciach kvôli potrebe spracovania enormného množstva dát a možnej paralelizácie problémov. Aplikácie využívajúce výkon GPU sa postupne rozširujú z vedeckých oblastí aj do bežných oblastí, najmä do rôznych softvérov spracovávajúcich obrázky a video.

V tejto dizertačnej práci sa budeme zaoberať dvomi oblasťami vedeckých výpočtov a ich urýchlením s využitím GPU. Prvou oblasťou je spracovanie a predspracovanie objemových dát pre rôzne segmentačné a vizualizačné techniky. Druhou oblasťou sú dátovo závislé triangulácie, ktoré budú ilustrované v metódach spracovania obrazu, konkrétne v ich zväčšovaní.

1.1 Spracovanie objemových dát

Spracovanie objemových dát môžeme začleniť do oblasti vizualizácie objemových dát. Hlavným odvetvím, ktoré sa zaoberá spracovaním a vizualizáciou objemových dát je medicínska vizualizácia, ktorá je aj najväčším producentom takýchto dát. Sú to hlavne

dáta z rôznych meracích zariadení ako sú počítačové tomografy (*computer tomography - CT*), tomografy na báze magnetickej rezonancie (*magnetic resonance - MRI*) a iné. Výhodou takýchto dát je presný popis meraného objektu a hlavne zachytenie jeho vnútornej štruktúry. Na druhej strane sú tieto dáta veľmi rozsiahle, čo vedie k problémom s ich vizualizáciou a spracovaním, ktoré niekedy trvá aj desiatky minút. V súčasnosti existuje množstvo algoritmov využívajúcich aj staršie grafické akcelerátory na vizualizáciu objemových dát. Na druhej strane, spracovanie takýchto dát na starších generáciách grafických kariet nie je príliš rozšírené z dôvodu ich viacerých obmedzení. S novými možnosťami GPU je proces spracovania objemových dát možné efektívnejšie preniesť na GPU a tak ťažiť z vysokého výkonu GPU. Tejto problematike sa venujeme v kapitole 3 „Prúdové spracovanie objemových dát“.

1.2 Dátovo závislé triangulácie

Dátovo závislé triangulácie spadajú pod oblasť výpočtovej geometrie, ktorá využíva výkon GPU len v obmedzenej miere. Obvykle ide o zobrazovanie rôznych geometrických štruktúr, a to najmä vo forme trojuholníkov. Spracovanie geometrie na GPU nie je príliš rozšírené z dôvodu potreby paralelizovať problém na dosiahnutie efektívneho riešenia. Existujú riešenia, ktoré využívajú GPU v geometrii aj na spracovanie vstupných dát, ale väčšinou sú spracované dáta priamo zobrazené a ich ďalšie spracovanie nie je umožnené prípadne v diskrétnom priestore [RT06, ST04]. V kapitole 4 „Paralelné dátovo závisle triangulácie a rekonštrukcia obrazu“ sa venujeme spracovaniu dátovo závislých triangulácii na GPU, pričom výsledok môže byť ďalej spracovávaný inými algoritmami. Výsledky dátovo závislých triangulácií ilustrujeme na zväčšovaní obrazu, pričom celý proces je vykonávaný na GPU.

1.3 Ciele dizertačnej práce

Hlavným cieľom dizertačnej práce je preveriť schopnosti moderných GPU pri všeobecnom spracovaní dát. Konkrétne ide o návrh nových algoritmov, prípadne optimalizáciu existujúcich na aktuálnu funkcionalitu GPU a využiť ju v čo najväčšej miere a tak dosiahnuť urýchlenie výpočtov oproti štandardným algoritmom. Pri spracovaní objemových dát ide hlavne o prúdový prístup k spracovaniu dát, nakoľko reálne dáta je na bežných počítačoch niekedy problém spracovať z dôvodu nedostatku pamäte. Tieto ciele môžeme zhrnúť do nasledujúcich bodov:

- Navrhnúť algoritmy spracovávajúce dáta na GPU pre rôzne oblasti použitia.
- Optimalizovať známe riešenia s využitím aktuálnych možností moderných GPU.
- Prispôsobiť algoritmy pracujúce s objemovými dátami prúdovému spracovaniu.

Kapitola 2

Grafický hardvér

V tejto kapitole si uvedieme základné informácie o grafickom hardvéri, jeho vzniku, využití pri renderovaní 3D scén ako aj na všeobecné výpočty. Predstavíme si hlavné grafické programovacie rozhrania na komunikáciu medzi grafickou kartou a aplikáciou.

2.1 Vývoj grafického hardvéru

Grafické karty sa začali postupne rozširovať do bežných počítačov v 80-tych rokoch minulého storočia. Hlavnými výrobcami v tej dobe boli firmy ATI, CirrusLogic, S3, Matrox. Tieto grafické karty zo začiatku podporovali len monochromatické zobrazovanie. Postupným vývojom sa karty vylepšovali, ponúkali kvalitnejší obraz s podporou viacerých farieb a väčším rozlíšením. Vývoj pokračoval a grafické karty podporovali operácie v dvojrozmernom priestore a neskôr aj v trojrozmernom. V 90-tych rokoch firma 3dfx Interactive predstavila revolučnú skupinu grafických kariet Voodoo. V tomto období vznikla aj firma NVIDIA, súčasný významný výrobca grafických kariet. Mnoho pôvodných výrobcov kariet zaniklo, prípadne boli odkúpení inými firmami. V dnešnej dobe existujú dvaja hlavný výrobcovia grafických kariet, a to firmy AMD ATI a NVIDIA. Grafické karty ponúkajú aj iní výrobcovia, ktorí ale majú malý podiel na trhu, prípadne sú zameraní na profesionálnu oblasť (Matrox). Veľkou firmou, ktorá sa zaoberá výrobou integrovaných grafických kariet je Intel, ktorý však vyrába grafické karty menšieho výkonu, vhodné na kancelárske použitie. Hlavným motivátorom vo vývoji grafických kariet je herný priemysel, ktorý stále kladie vysoké požiadavky na ich výkon. Dnešné grafické karty svojim vysokým výkonom prevyšujú aj výkon bežných procesorov počítača. Spolu s rastúcim výkonom sa rozširovala aj funkcionalita grafických kariet a tento vývoj môžeme rozdeliť do piatich generácií.

Prvá generácia - textúrovanie
Hlavným predstaviteľom tejto triedy bola rodina grafických kariet Voodoo od spoločnosti 3dfx Interactive, uvedená na trh v roku 1995. Tieto grafické karty umožňovali rasterizovať transformované trojuholníky na fragmenty a vypĺňať ich farbou. Výsledná farba bola

interpolovaná z farby definovanej vo vrcholoch, prípadne vypočítaná pomocou osvetľovacieho modelu, ktorý bol nastavený používateľom. Hlavným prínosom tejto generácie bolo použitie textúr, ktoré vniesli novú kvalitu do renderovacieho procesu.

Druhá generácia - transformácie na vrcholoch
Vývoju grafických kariet sa začali venovať viaceré firmy, ktoré uviedli na trh druhú generáciu, a to grafické karty GeForce 256, GeForce 2 od firmy NVIDIA, Radeon 7500 od ATI a Savage3D od firmy S3. Tieto karty mali dva druhy spracovávajúcich jednotiek (tzv. *shaders*), ktoré vykonávali výpočty na fragmentoch—fragmentová jednotka (*fragment shader*) a na vrcholoch—vrcholová jednotka (*vertex shader*). Fragment označuje rozšírenie pixela—farby o dodatočné údaje využívané na grafických kartách ako je priehľadnosť, vzdialenosť od pozorovateľa, a iné. Spracovávajúce jednotky umožňovali nastavovať príslušné stavy výpočtov a podľa nich dosiahnuť požadované výpočty na fragmentoch respektíve na vrcholoch. Toto bol krok k postupnému rozširovaniu programovateľnosti grafických kariet.

Tretia generácia - programovanie vrcholových jednotiek
Táto generácia kariet rozšírila množinu stavov pre výpočet fragmentov. Hlavným prínosom bola jednoduchá možnosť programovania jednotiek spracovávajúcich vrcholy, ktoré umožňovali vyššiu flexibilitu pre programátora ako iba nastavovanie stavov. K takýmto grafickým kartám patria GeForce tretej a štvrtej generácie od NVIDIA a Radeon ôsmej generácie od ATI.

Štvrtá generácia - programovanie fragmentových jednotiek
Štvrtá generácia kariet prináša vyššiu programovateľnosť spracovávajúcich jednotiek. Zavádza novú funkcionalitu a to vetvenie, cykly, umožňuje čítanie textúr aj vo vrcholových programoch. Jedným z prínosov bolo aj vykonávanie výpočtov a podpora textúr s plávajúcou desatinnou čiarkou, čo prinieslo vyššiu presnosť výpočtov a kvalitu renderovania. Predošlé generácie podporovali len celočíselné renderovanie a výpočty. Pri tejto generácii grafických kariet sa začal vo veľkej miere používať pojem *grafický procesor—GPU* (*Graphics Processing Unit*), ktorý bol ale známy už skôr. Aktuálne GPU sa používajú aj na výpočty nie nutne súvisiace s grafikou a renderovaním 2D/3D scény. Takéto výpočty označujeme pojmom *všeobecné výpočty na grafických procesoroch - GPGPU* (*General-Purpouse Computation on Graphics Processing Unit*) [GPG10]. Hlavnými predstaviteľmi tejto generácie GPU boli rady GeForce FX od NVIDIA, Radeon deviatej generácie od ATI a novšie. V tejto dobe sa začalo rozširovať použitie viacerých GPU v jednom počítači, či už ako dve nezávislé karty alebo dva grafické procesory na jednej karte. Táto technológia nebola nová, bola už uvedená skôr firmou 3dfx Interactive spolu s čipom Voodoo, ale sa vtedy v dostatočnej miere neujala. Kúpou firmy 3dfx Interactive firmou NVIDIA, kúpila firma aj práva na túto technológiu, ktorá sa označuje SLI (Scalable Link Interface). Podobnú technológiu podporujú aj karty od firmy ATI s označením CrossFire.

Piata generácia - zjednotené výpočtové jednotky
Predošlé generácie grafických kariet mali fixný počet jednotiek spracovávajúcich fragmenty a vrcholy, pričom počet fragmentových jednotiek bol rádovo väčší ako počet vrcholových jednotiek. Tento pomer zvýhodňoval aplikácie, ktoré boli náročné na spracovanie fragmentov, a to najmä hry. Pre aplikácie, ktoré spracovávali hlavne geometriu bol tento pomer nevýhodný a mnoho fragmentových jednotiek bolo nevyužitých. Piata generácia GPU zjednotila obe spracovateľské jednotky do výpočtových jednotiek, ktoré mohli spracovávať vrcholy aj fragmenty. Týmto krokom sa zvýšila efektívnosť výpočtov. Pri zjednotení jednotiek došlo aj k zjednoteniu obmedzení na rôzny počet inštrukcií a registrov pre vrcholové a fragmentové jednotky. Piata generácia GPU umožňuje všeobecnejšie spracovanie geometrie, nie len jednoduché transformácie, a to za pomoci jednotky spracovávajúcej geometriu (*geometry shader*). Táto jednotka umožňuje generovať nové polygóny priamo na grafickej karte, čo doposiaľ nebolo možné. Má však veľa obmedzení a malý výkon, a tak nie je veľmi využívaná.

Zatriedenie grafických kariet do generácií nie je všeobecne ustanovené a preto sa môžu rôzne triedenia líšiť v závislosti na špecifických atribútoch, ktoré sa berú do úvahy pri rozdelení týchto kariet.

2.2 Programovacie rozhranie grafických kariet

Spolu s vývojom grafických kariet sa vyvíjalo aj programovacie rozhranie na využívanie funkcionality grafických kariet. Medzi takéto programovacie rozhrania môžeme zaradiť OpenGL a Direct3D.

2.2.1 OpenGL

OpenGL (*Open Graphics Library*) [Gro10b] je otvorené rozhranie na využívanie grafických kariet. Bolo vytvorené firmou SGI (Silicon Graphics, Inc.) a pôvodne bolo spravované konzorciom OpenGL ARB (*Architecture Review Board*). V súčasnosti je spravované firmami zastrešenými v konzorciu Khronos Group [Gro10a]. OpenGL je systémovo nezávislé rozhranie, ktoré je podporované na takmer každom operačnom systéme. Môže byť využité vo viacerých programovacích jazykoch. Jeho hlavným rozdielom oproti Direct3D je prístup k novej funkcionalite grafických kariet. V špecifikácii OpenGL je definovaná základná funkcionalita, ktorú musí podporovať grafická karta deklarujúca podporu danej verzie OpenGL. S nástupom nových grafických kariet a ich novou funkcionalitou výrobcovia definujú vlastné rozšírenia (*OpenGL extensions*), funkcie a premenné, ktoré môžu vývojári používať. Takto si každý výrobca definuje vlastné rozšírenia pre vlastné grafické karty. Po čase sa významné funkcionality zoskupujú a vydávajú ako univerzálne rozšírenie (rozšírenia s označením *GL_EXT_názov_rozšírenia*, *GL_ARB_názov_rozšírenia*). Tieto rozšírenia sú potom väčšinou podporované viacerými výrobcami. Významné rozšírenia sa priamo zahrnú do novej špecifikácie OpenGL pri jej

návrhu. Tento postup má výhodu v tom, že nová funkcionalita grafických kariet je dostupná s vydaním nových ovládačov a nie je potrebné čakať na novú špecifikáciu. V súčasnosti je najnovšia špecifikácia OpenGL 4.0 [SA10], ale ešte nie sú vydané nové ovládače pre jej plnú podporu. Tento spôsob rozširovania novej funkcionality má však nevýhodu v potrebe špecifikovať program pre konkrétny typ grafickej karty s danou funkcionalitou.

OpenGL rozhranie je najčastejšie využívané vedeckými a experimentálnymi aplikáciami práve kvôli systémovej nezávislosti.

2.2.2 Direct3D

Toto rozhranie je navrhnuté a udržiavané firmou Microsoft, preto je dostupné len na operačných systémoch Windows firmy Microsoft. Aktualizácia tohto rozhrania je založená na očakávaných požiadavkách, čo znamená, že v nových verziách je zahrnutá taká funkcionalita, ktorá bude v budúcnosti potrebná ale grafické karty ju v súčasnosti nemusia poskytovať. Avšak návrh a nové verzie vznikajú v úzkej spolupráci s výrobcami, a tak je väčšinou pri vydaní novej špecifikácie Direct3D už dostupný aj grafický hardvér, ktorý ju podporuje. V prípade, že grafický hardvér takúto funkcionalitu nepodporuje, môže byť emulovaná softvérovo. Toto rozhranie je súčasťou balíka DirectX, ktorý zastrešuje nie len grafické rozhranie, ale aj multimediálne rozhranie pre prácu so zvukmi a animáciami. V súčasnosti je aktuálna verzia DirectX 11 podporovaná len GPU od firmy ATI. Verzia DirectX 11 [Cor10a] definuje ďalšiu jednotku a to univerzálnu výpočtovú jednotku (*direct compute*), ktorá by mala programovanie a využitie GPU viac priblížiť programovaniu CPU. DirectX je momentálne najpoužívanejšie na vývoj počítačových hier, kde sa OpenGL rozhranie takmer vôbec nepoužíva.

Obdobne existujú aj rozhrania na programovanie výpočotvých jednotiek grafických kariet. Hlavnými zástupcami sú jazyky Cg, HLSL a glsl.

2.2.3 Cg

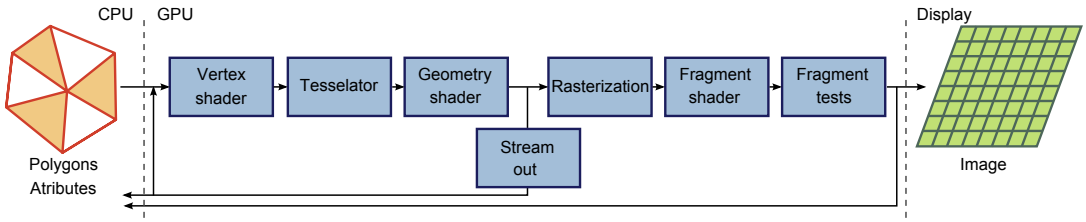
Cg (*C for graphics*) je špecifikácia [FK03, Cor10c] programovacieho jazyka pre výpočtové jednotky grafických kariet založená na jazyku C, z ktorého prebrala syntax. Tento štandard sa skladá z dvoch častí, a to programovacieho rozhrania obsluhujúceho jednotky grafických kariet (inicializácia, nastavovanie, kompilovanie, ...) a z programovacieho jazyka pre tieto jednotky. Štandard Cg bol uvedený firmou NVIDIA nad OpenGL špecifikáciou a preto využíva OpenGL rozšírenia definované hlavne firmou NVIDIA. Grafické karty ATI bolo pôvodne treba programovať priamo v asembleri grafických kariet pre slabú podporu tohto jazyka. Pre rozhranie Direct3D existuje veľmi podobný programovací jazyk HLSL (*High level shading language*) [PM03, Cor10b], ktorý vyvinuli firmy Microsoft a NVIDIA.

2.2.4 glsl

S uvedením verzie OpenGL 2.0 bola uvedená aj špecifikácia programovacieho jazyka pre jednotky grafických kariet glsl (*OpenGL shading language*) [Ros05]. Tento jazyk je priamo zahrnutý v OpenGL špecifikácii a je podporovaný všetkými výrobcami grafických kariet. Syntax je taktiež založená na jazyku C s podobným použitým ako Cg. Hlavným rozdielom medzi Cg a glsl je možnosť pristupovať k nastaveniam (stavom) OpenGL priamo v zdrojovom kóde glsl. Tento prístup Cg jazyk neumožňuje, potrebné nastavenia je nutné importovať z hlavného OpenGL programu.

2.3 Grafické zobrazovanie

Hlavným využívateľom grafických kariet sú počítačové hry, ktoré kladú vysoké požiadavky na výkon a ich nové možnosti. Základným princípom hier je interaktívne zobrazovanie trojrozmerných scén (väčšina hier). Postup tohto zobrazovania je v podstate nemenný, pričom nová funkcionalita GPU tento proces len obohacuje. Takýto proces zobrazovania, renderingu môžeme rozdeliť do troch úrovní: spracovanie geometrie, rasterizácia a fragmentové operácie. Jednotlivé úrovne budú v stručnosti opísané nižšie. Proces zobrazovania 3D scény je totožný pre grafické karty s rôznou architektúrou a je načrtnutý na obrázku 2.1.



Obrázok 2.1: Postup zobrazenia 3D scény na grafickej karte.

Spracovanie geometrie
V prvej časti zobrazovacieho procesu sa spracováva 3D scéna, ktorá je navrhnutá programátorom. Táto scéna vo väčšine prípadov pozostáva z objektov opísaných povrchovou reprezentáciou pomocou trojuholníkov, pretože grafická karta vie spracovať len rovinné polygóny. Každý objekt je takto opísaný sadou trojuholníkov s vrcholmi, v ktorých sú definované dodatočné atribúty ako je farba, normála, textúrová súradnica a iné. Vstupné vrcholy sú spracovávané grafickou kartou, kde pre každý vstupný vrchol existuje práve jeden výstupný vrchol z vrcholovej jednotky. V tejto jednotke sa dané vrcholy transformujú (rotujú, posúvajú, škálujú, atď.) pričom sa ich atribúty upravujú podľa potreby aplikácie. Vrcholy, ktoré nie sú viditeľné, sú odstránené zo zobrazovacieho procesu. Takto spracované vrcholy vstupujú do jednotky zodpovedajúcej za teseláciu, kde sú dané polygóny rozdelené na trojuholníky. Najnovšie GPU (DirectX 11) umožňujú ovplyvňovať proces

teselácie podľa potreby aplikácie. Takto vygenerované trojuholníky vstupujú do poslednej jednotky spracovávajúcej geometriu, kde môžu byť doplnené dodatočnou geometriou, prípadne odstránené zo zobrazovacieho procesu. Z tejto časti môže byť výsledná geometria stiahnutá naspäť na CPU, prípadne iteratívne ďalej spracovávaná. Toto umožňuje novšia technológia s názvom *stream out* (Direct3D), respektívne *transform feedback* (OpenGL). Väčšina aplikácií však pokračuje ďalej v zobrazovacom procese v rasterizačnej časti.

Rasterizácia
V tejto časti sú výsledné trojuholníky rasterizované na fragmenty. Pre každý fragment sú interpolované údaje z atribútov obsiahnutých vo vrcholoch. Na základe týchto atribútov je možné vo fragmentovej jednotke vykonávať rôzne operácie. Medzi hlavné výpočty patrí výpočet osvetlenia, mapovanie textúr, mapovanie farieb a materiálov. Výsledný vygenerovaný fragment postúpi do záverečnej fázy spracovania. Moderné GPU umožňujú v prípade potreby v tejto fáze odstrániť jednotlivé fragmenty z procesu zobrazovania a umožňujú zapisovať fragmenty do viacerých pamäťových oblastí (stereo zobrazovanie), čo je často využívane GPGPU aplikáciami.

Fragmentové operácie
Výsledný fragment prechádza cez sadu testov, ktoré rozhodnú o jeho zapísaní, respektíve nezapísaní do výsledného obrazu. Medzi významné testy patrí test na vzdialenosť od pozorovateľa (*depth test*), ktorý rieši problém viditeľnosti. Ďalším testom je test na priehľadnosť (*alpha test*), ktorým sa dajú simulovať rôzne objekty so zložitou hranicou (listy, tráva, oheň). Posledným procesom je proces zmiešavania farebnej hodnoty daného fragmentu s hodnotou uloženou vo výslednom obraze (*alpha blending*). Týmto procesom sa dá dosiahnuť efekt polopriehľadných materiálov. Takýto fragment sa stáva pixelom, ktorý má už len jediný atribút, a to farbu, pričom je na základe logických operácii zapísaný do výsledného obrazu alebo je odstránený.

2.4 GPGPU

S postupným vývojom a funkcionalitou grafických kariet sa rozširovali oblasti použitia nie len na zobrazovanie 3D scén. Zo začiatku to boli iné oblasti počítačovej grafiky, hlavne oblasť spracovania obrazu, kompresia/dekompresia videa. S rozširovaním programovateľnosti výpočtových jednotiek a unifikovaním architektúry sa grafické karty začali využívať aj v úplne iných oblastiach ako je počítačová grafika. Jedným z prvých výpočtov, ktoré sa dostali do povedomia mimo komunity GPU bolo oceňovanie opcií na GPU [KP05]. Potom to boli výpočty riešiace štandardné matematické problémy ako riešenie systému lineárnych rovníc [KW05b] a triediace algoritmy [KW05a]. Ďalším úspešným projektom je projekt GPUGRID [GPU10, BHG+10] z oblasti biomedicíny, ktorý využíva distribuovanú sieť počítačov vybavených GPU na výpočty. Tieto oblasti, ako už bolo spomenuté, označujeme skratkou GPGPU [GPG10, LHK+04]. V dnešnej

dobe existuje viacero oblastí výskumu, ktoré využívajú GPU na urýchľovanie. Pre tieto výpočty sa môže použiť jedno z grafických programovacích rozhraní. Najčastejšie je to OpenGL, pretože je podporované rôznymi operačnými systémami, čo je požadované vedeckými komunitami. OpenGL je však primárne určené na zobrazovanie 3D scén a z toho dôvodu kladie rôzne obmedzenia na využívanie potenciálu GPU. Hlavným problémom je správa pamäte, kde má používateľ možnosť využívať ako úložisko dát najmä textúry. Ďalším obmedzením je, že do textúr môžeme zapisovať len na vopred definované pozície (adresu výstupnej pozície nie je možné zmeniť). Toto obmedzenie vyplýva z logiky vývoja OpenGL a potrieb na renderovanie, ktoré je vo väčšej miere ľahko paralelizovatelné. Výrobcovia súčasných GPU však prispôsobujú architektúru tak, aby vo väčšej miere vyhovovala GPGPU výpočtom a kládla menej obmedzení. Táto oblasť má veľký potenciál vo využívaní GPU a tým pádom aj v ich predaji. Tento potenciál je daný hlavne vysokým výpočtovým výkonom GPU a jeho využitím. Jedným z takýchto prispôsobení požiadavkám vedeckých aplikácií je podpora výpočtov s plávajúcou desatinnou čiarkou v dvojitejej presnosti, pretože výpočty v jednoduchej presnosti sú často nedostatočné. V súčasnosti však nie je dvojitá presnosť podporovaná ovládačmi grafických kariet. Preto vznikli nové rozhrania, ktoré sú primárne určené pre potreby GPGPU aplikácií. Jedným z dôvodov sú už spomínané obmedzenia grafického rozhrania, zjednodušenie týchto rozhraní a priblíženie ku klasickému programovaniu na CPU. Priekopníkom v zavedení rozhrania pre GPGPU je firma NVIDIA s jej rozhraním CUDA (*Compute Unified Device Architecture*) [CUD10]. Toto rozhranie odstraňuje podstatné obmedzenia grafického rozhrania. Umožňuje zapisovanie dát na ľubovoľné pozície v pamäti, čo ale na druhej strane vyžaduje synchronizáciu výpočtových jednotiek. Pomocou tohto rozhrania sa dajú implementovať všeobecnejšie dátové štruktúry, čo doposiaľ nebolo možné, alebo len s veľkou réžiou, pod grafickým rozhraním. Nevýhodou tohto rozhrania je jeho podpora len pre GPU od firmy NVIDIA. Firma AMD vydáva taktiež svoje GPGPU rozhranie s označením ATI Stream [AMD10]. Toto rozhranie je podobné ako CUDA. Koncom roku 2008 bola konzorciom Khronos Group vydaná špecifikácia OpenCL (*Open Computing Language*) [Ope09], na ktorej pracovali viacerí výrobcovia GPU a CPU. Táto špecifikácia zoskupuje paralelné programovanie ako také a zavádza potrebný štandard. OpenCL špecifikácia je univerzálna a neplatí len pre GPU, ale aj rôzne typy procesorov (Intel, Cell BE) a mobilné zariadenia. Týmto krokom sa zabezpečuje prenositeľnosť programov vyhovujúcich danej špecifikácii. V súčasnosti je OpenCL podporované firmami AMD (GPU, CPU), Apple, Intel, NVIDIA.

Kapitola 3

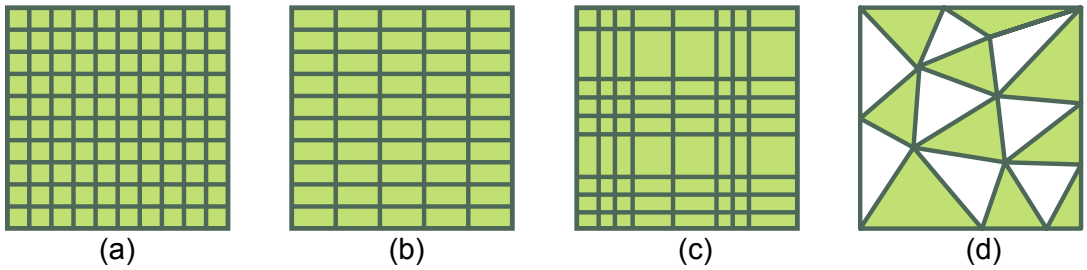
Prúdové spracovanie objemových dát

V tejto kapitole sa budeme venovať využitiu moderných GPU s podporou OpenGL rozhrania pri prúdovom spracovaní objemových dát. Uvedieme princíp prúdového spracovania objemových dát po rezoch s cieľom minimalizovať pamäťové nároky algoritmov. Taktiež predstavíme všeobecné techniky, ktoré je možné jednoducho modifikovať pre rôzne algoritmy spracovávajúce objemové dáta. Predstavené techniky demonštrujeme v reálnej aplikácii, ktorá využíva funkcionalitu a výkon GPU.

3.1 Objemové dáta

V počítačovej grafike sú trojrozmerné objekty definované pomocou rôznych reprezentácií. Najvyužívanejšie sú povrchové reprezentácie, ktoré opisujú objekty pomocou rovinných mnohouholníkov, najčastejšie trojuholníkov. Objemová reprezentácia opisuje vnútornú štruktúru objektov, na rozdiel od povrchovej, ktorá definuje len ich povrch. Objemové dáta bývajú uložené v rôznych formátoch. Štandardné formáty používajú kartézsku mriežku, kde každý bod tejto mriežky opisuje vlastnosti dát (obrázok 3.1(a)). Tento bod sa označuje pojmom voxel (*volume element*). Ďalšími možnými usporiadaniami dát sú regulárne (obrázok 3.1(b)) a rovnobežné mriežky (obrázok 3.1(c)). Špeciálnu kategóriu tvoria neštruktúrované mriežky, ktoré sú opísané štvorstenmi pomocou špeciálnych dátových štruktúr (obrázok 3.1(d)). Vlastnosti, ktoré popisujú objemové dáta závisia hlavne od spôsobu ich získavania. Najčastejšie sa objemové dáta využívajú v medicíne, kde ich produkujú rôzne snímacie zariadenia ako počítačová tomografia (*CT*), magnetická rezonancia (*MRI*), 3D ultrazvuk, konfokálna mikroskopia a mnohé iné [Gf05]. Ďalším veľkým producentom objemových dát sú rôzne odvetvia priemyslu, ktoré skúmajú prúdenie kvapalín a plynov. Niektoré dáta sú umelo generované z rôznych simulácií a výpočtov. Rôzne odvetvia produkujú dáta s rôznymi vlastnosťami, najčastejšie sú to však hodnoty intenzít reprezentované pomocou odtieňov šedej. Objemové dáta môžu obsahovať viacrozmerné informácie, napríklad farbu vo forme RGB informácie, prípadne smerový vektor prúdenia kvapaliny a podobne. Dáta získavané z dnešných medicínskych modalít zaberajú stovky MB a niekedy až GB (časové snímky, snímky celého tela pacienta). Štandardné rozmery

tomografického rezu sú 512×512 pixelov a ich počet závisí od skúmanej časti pacienta a častokrát presahujú 1000 rezov. Tieto dáta mávajú obvykle 12-bitovú presnosť. Algoritmy a techniky predstavené nižšie pracujú so skalárnymi dátami uloženými v kartézskej mriežke, avšak je možné ich upraviť aj na spracovávanie dát v iných typoch mriežok, prípadne dáta prevzorkovať.



Obrázok 3.1: Štandardné usporiadania rezov objemových dát do kartézskej (a), regulárnej (b), rovnobežnej (c) a neštruktúrovanej (d) mriežky.

3.2 Prúdové spracovanie dát

Objemové dáta obsahujú veľké množstvo údajov a bez podporných algoritmov na ich spracovanie a vizualizáciu je veľmi náročné sa v nich orientovať. Medzi hlavné algoritmy spracovávajúce objemové dáta patria algoritmy na odstraňovanie šumu, ostrenie hrán, detekciu hrán, označovanie regiónov, segmentáciu a mnohé ďalšie. Tieto algoritmy môžeme rozdeliť podľa prístupu k jednotlivým elementom dát na bodové, lokálne a globálne. Pod bodovými rozumieme algoritmy, ktoré spracovávajú iba hodnotu príslušného voxela (napr. prahovanie). Pod lokálnymi rozumieme algoritmy, ktoré potrebujú aj isté okolie príslušného voxela (napr. filtrovanie, detekcia hrán). Veľkosť tohto okolia závisí na danom algoritme. Globálne algoritmy potrebujú na výpočet výslednej hodnoty pre spracovávaný voxel informáciu o všetkých voxeloch, niekedy dokonca vo viacerých prechodoch spracovania (napr. Fourierova transformácia). Pri štandardnom spracovaní objemových dát sa načítajú do pamäte, kde k nim pristupuje algoritmus, vykonáva výpočty a generuje výstupné hodnoty. Nakoľko reálne objemové dáta vždy obsahovali relatívne veľa údajov v porovnaní s aktuálnymi pamäťovými možnosťami bežných počítačov, je toto spracovanie pamäťovo náročné a niekedy pomocou štandardných prístupov na bežne dostupných počítačoch až nemožné.

Odstránením problému nedostatku pamäti sa venuje prúdové spracovanie dát, kde je objem rozdelený na časti a tieto sú spracovávané postupne – prúdovo. Do pamäti sa nahrá prvá časť dát, spracuje sa, uloží (zobrazí) výsledok a pokračuje sa ďalšou časťou pokiaľ sa nespracujú celé dáta. Pri tomto procese je v pamäti uložená len časť dát, ktorá nezaťažuje natoľko pamäť a tak dovoľuje spracovanie aj takých dát, ktoré sa nezmestia celé do operačnej pamäte počítača. Častým prístupom je rozdelenie dát na bloky (kocky/kvádre) s následným spracovaním [LSMT99]. Toto delenie je výhodné hlavne

pre vizualizáciu dát, kde je potrebný „náhodný“ prístup k dátam. Taktiež je vo veľkej miere využívané pri urýchľovaní objemového zobrazovania [HKRs+06]. Štandardne sa dáta spracovávajú postupuje po voxeloch od začiatku dát po koniec. Pri takomto spracovaní je výhodnejšie spracovanie po rezoch [VVS+07]. Hlavným z dôvodov je potreba duplikovania voxelov na hraniciach blokov pri lokálnych operáciách. Táto duplicita výrazne zvyšuje pamäťové nároky. Špeciálnym blokom je aj rez, kde rozmer v smere Z je rovný jednej. Pri spracovaní po rezoch táto duplicita v smeroch X,Y odpadá a v smere Z je automaticky dodržaná nahraním potrebného počtu rezov do pamäte. Tento proces je výhodnejší pre bodové a lokálne operácie. Prúdové spracovanie po rezoch sa dá aplikovať aj na viaceré algoritmy globálneho charakteru, ale s potrebou ukladania medzivýsledku na disk, čo predstavuje isté spomalenie [VVS+07].

Jednotlivé algoritmy potrebujú rôzny počet rezov v závislosti na vykonávanej operácii (pri bodovej je to jeden rez). Na výpise 3.2 je zobrazené rozhranie, ktoré sme navrhli pre prúdového spracovania objemových dát, kde si metóda (ozn. process) na začiatku spracovania interne ukladá potrebný počet rezov, následne spracuje príslušný rez a vráti ho aplikácii. Interné ukladanie rezov je vykonávané pomocou cyklického zásobníka (spájaný zoznam), kde je už nepotrebný rez nahradený novým. Takýmto spôsobom sa postupne spracujú všetky rezy. Uvedený prístup je univerzálny a nezávislý na hardvérovej optimalizácii (CPU, SSE, GPU).

```
1  if(process.open())
2  {
3    while(process.isReadyForUpload() || process.isReadyForDownload())
4    {
5      if(process.isReadyForUpload())
6      {
7        ...
8        process.uploadSlice(sliceIn);
9      }
10     if(process.isReadyForDownload())
11     {
12       process.downloadSlice(sliceOut);
13       ...
14     }
15   }
16   process.close();
17 }
```

Výpis 3.2: Rozhranie pre prúdové spracovanie objemových dát po rezoch.

3.3 Prúdové spracovanie dát s využitím GPU

V tejto časti uvedieme základné dátové štruktúry, ktoré budeme využívať pri spracovaní objemových dát. Taktiež predstavíme všeobecné techniky, ktoré poskytujú moderné GPU a je možné ich využiť pri prúdovom spracovaní objemových dát.

3.3.1 Dátové štruktúry

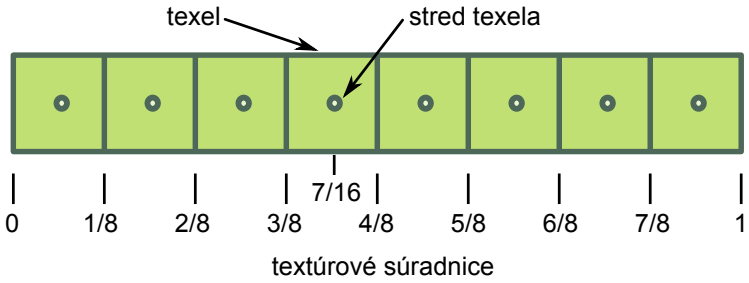
Postup spracovania objemových dát prúdovým prístupom na GPU vychádza z vyššie uvedeného všeobecného návrhu. Je prispôsobený možnostiam OpenGL rozhrania a dátovým štruktúram. Na simulovanie cyklického zásobníka sme použili 3D textúru. Rozmer textúry v smere X, Y je rovnaký ako je rozmer spracovávaného rezu. V smere Z tento rozmer zodpovedá počtu potrebných rezov na výpočet. Pri bodových operáciách, kde je potrebný iba jeden rez, môže byť táto 3D textúra nahradená 2D textúrou. Pri 3D textúre je ďalej vhodné nastaviť orezávací mód v smere Z na opakovanie (*GL_REPEATE*), ktorý ignoruje celú časť textúrových súradníc a tým pádom simuluje cyklický zásobník bez väčšej réžie. Orezávací mód v smeroch X a Y môže byť nastavený na orezanie na hranicu alebo hranu, prípadne na zrkadlenie v závislosti na požiadavke aplikácie. Pri orezaní na hranu je pri čítaní z textúry mimo rozsah vrátená hodnota z hrany, t.j. prvá alebo posledná hodnota. Pri orezaní na hranicu je vrátená hodnota hranice, ak bola špecifikovaná pri vytváraní textúry. Pri nešpecifikovaní tejto hodnoty je v tomto prípade vrátená nulová hodnota. Pri nastavenom zrkadlení sa zrkadlenie vykonáva zarovnané na hranicu, čo znamená že sa zrkadlia všetky hodnoty v textúre a to aj prvá, respektíve posledná hodnota. Pri požiadavke zrkadlenia zarovnanom na hranu (krajná hodnota sa nezrkadlí) je potrebné toto zrkadlenie robiť manuálne vo fragmentovom programe. Na začiatku sú postupne nahrávané rezy do 3D textúry, pokiaľ nie je nahratý minimálny potrebný počet rezov na zahájenie výpočtu. Podľa požiadaviek aplikácie musia byť zvyšné rezy vynulované (zarovnanie na hranicu), naplnené hodnotami z prvého rezu (zarovnanie na hranu) alebo naplnené hodnotami z nahratých rezov v opačnom poradí (zrkadlenie) pre simulovanie orezávacieho módu v smere Z. Interná presnosť textúr je nastavená podľa požiadaviek aplikácie, štandardne pre spracovanie objemových dát je to 32-bitová presnosť s plávajúcou desatinnou čiarkou.

Predstavili sme si základné dátové štruktúry využívané pri všeobecnom spracovaní objemových dát. V ďalších častiach budú prípadne tieto dátové štruktúry upravené podľa požiadaviek predstavenej techniky.

3.3.2 Základný postup

V tejto časti si predstavíme základný postup spracovania objemových dát z pohľadu prenosu dát na GPU. Na začiatku je nahratý potrebný počet rezov z CPU do 3D textúry na GPU. Následne prebieha spracovanie vstupného rezu. Spracovaný rez je uložený, zapísaný do výstupnej pamäte (*framebuffer*), ktorá je uložená na disk, prípadne využitá na iné spracovanie. Samotné spracovávanie je vykonávané vo fragmentovom programe, kde

vstupom je 3D textúra s potrebným počtom rezov a výstupom sú údaje (voxely) zapísané do výstupnej pamäti. Samozrejme, vstupom do fragmentového programu sú aj dodatočné údaje potrebné na výpočet. Potrebným údajom pri spracovaní príslušného rezu sú jeho textúrové súradnice v 3D textúre. Na základe týchto súradníc GPU interpoluje aktuálnu textúrovú súradnicu pre daný voxel. Pre dosiahnutie správneho čítania hodnôt z textúr je vhodné posunúť textúrové súradnice o pol texela v zápornom smere (obrázok 3.3) a vypnúť OpenGL filtrovanie (nastaviť na filtrovanie pomocou najbližšieho suseda).

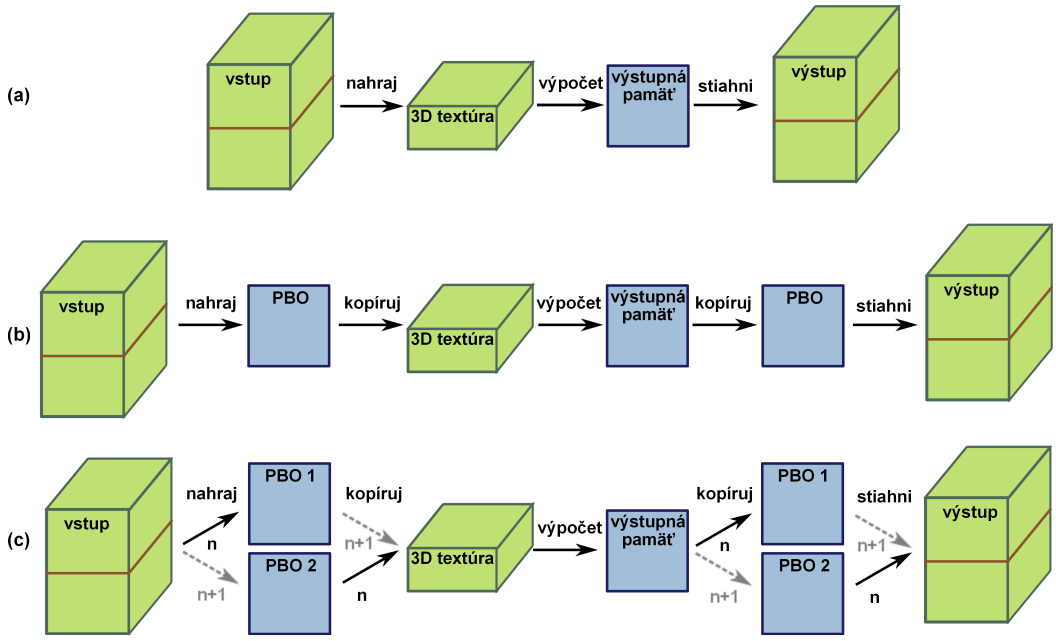


Obrázok 3.3: Nastavenie textúrových súradníc pre získavanie hodnôt zo stredu texelov.

3.3.3 Asynchrónny prenos dát

Vyššie uvedený priamočiary postup je možné aplikovať aj na starších GPU. Aktuálne modely GPU však umožňujú využiť asynchrónny prenos dát medzi CPU a GPU pamäťou bez potreby synchronizácie. OpenGL volania sú ukladané vo fronte ovládača GPU a pri dostatočnom počte volaní sú zaslané na GPU a vykonané. Väčšina OpenGL volaní nevyžaduje synchronizáciu, čo znamená, že pri danom volaní ovládač len zaradí volanie do fronty a umožňuje CPU ďalšie vykonávanie. Avšak niektoré volania vyžadujú synchronizáciu CPU a GPU. Medzi takéto volania patria aj volania vykonávajúce transfer dát na/z GPU. Pri takomto volaní ovládač neukončí dané volanie, ale čaká na dokončenie výpočtu na GPU, prípadne GPU čaká na prenos potrebných dát. Pri takýchto synchrónnych volaniach dochádza k plytvaniu výpočtovými zdrojmi či už CPU alebo GPU a dochádza k spomaleniu výpočtov. Na obrázku 3.4(a) je zobrazený štandardný prenos dát bez využitia asynchrónnej technológie. Asynchrónnu technológiu prenosu dát je možné využiť pomocou OpenGL rozšírenia *ARB_pixel_buffer_object* (ozn. *PBO*). Toto rozšírenie prináša vyššiu kontrolu nad správou pamäti a umožnuje priamy pamäťový prístup medzi CPU a GPU [Ahn07]. Toto rozšírenie je možné využiť aj na kopírovanie medzi rôznymi časťami GPU pamätí. Pomocou tohto rozšírenia je možné odstrániť volania, ktoré vyžadujú synchronizáciu CPU s GPU ako je znázornené na obrázku 3.4(b). Avšak táto technika musí byť správne použitá, aby sa dosiahol požadovaný efekt asynchrónneho prenosu dát. Pri použití jedného PBO sa síce docieli asynchrónny prenos dát, ale GPU musí čakať na tieto dáta a pracuje neefektívne. Z tohto dôvodu je potrebné použiť viac PBO (minimálne dva). Jeden PBO nahráva dáta na GPU priamo do video pamäti bez obmedzenia prebiehajúcich výpočtov. GPU počas tohto nahrávania spracováva dáta

uložené v druhom PBO. Pri skončení výpočtu na aktuálnom PBO nasleduje výpočet na ďalšom a pôvodný PBO sa môže použiť na asynchrónne nahrávanie dát. Tento proces pokračuje až do skončenia výpočtu. Obdobne sa postupuje aj pri sťahovaní dát z GPU. Asynchrónny prenos dát a výsledný výpočtový proces je zobrazený na obrázku 3.4(c). Týmto postupom sa odstráni pôvodná synchrónna funkcionalita prenosu dát medzi CPU a GPU. Takýto proces prenosu dát je vhodný práve pre prúdové spracovanie dát, prípadne na veľké prenosy dát.



Obrázok 3.4: Zobrazenie rôznych verzií prenosu dát. Pôvodná technológia pomocou synchrónnych volaní (a). Asynchrónny prenos dát pomocou PBO technológie (b). Prúdová verzia asynchrónneho prenosu dát (c).

3.3.4 Spracovanie viacerých rezov súčasne

V tejto časti si predstavíme techniku, ktorú využijeme na spracovanie viacerých rezov súčasne v jednom fragmentovom programe.

Staršie verzie OpenGL špecifikácie povoľujú renderovanie len do výstupnej pamäti (*framebuffer*) grafickej karty. Odtiaľto je možné výsledky skopírovať na CPU alebo do textúry na GPU pre ďalšie spracovanie. Priame renderovanie (zapisovanie) do textúry špecifikuje OpenGL rozšírenie *ARB_framebuffer_object* (ozn. *FBO*). Toto rozšírenie umožňuje priame renderovanie aj do 3D textúry na danú pozíciu (rez). Pomocou tohto rozšírenia tak nie je potrebné kopírovať medzi sebou časti video pamätí.

S využitím OpenGL rozšírenia *ARB_draw_buffers* je možné zapisovať do viacerých renderovacích oblastí súčasne. Táto technológia sa označuje ako *Multiple Render Targets – MRT*. Pri použití MRT rozšírenia spolu s FBO rozšírením je takýmto spôsobom možné

renderovať do viacerých textúr súčasne. Počet týchto renderovacích oblastí (textúr) je hardvérovo závislý, avšak maximálny počet je stanovený na 16 (OpenGL 4.0). Na toto rozšírenie je automaticky naviazaná aj glsl špecifikácia [Ros05], kde sa vo fragmentovom programe použijú viaceré výstupné farebné hodnoty (RGBA).

MRT technika sa najčastejšie využíva pri *odloženom tieňovaní* (*deferred shading*), kde sa náročný výpočet osvetľovacieho modelu vykonáva v záverečnej fáze zobrazovania iba pre viditeľné fragmenty [Shi05, HH04]. Počas štandardného renderovania scény sa nevykonáva výpočet osvetlenia pre každý fragment, ale sa do výstupných textúr ukladajú len hodnoty potrebné na tieňovanie (pozícia fragmentu, normála v príslušnom fragmente, materiálové vlastnosti), ktoré je následne vykonané . Bez možnosti zapisovania do viacerých textúr súčasne, by bol takýto výpočet tieňovania neefektívny.

Pomocou tohto rozšírenia je možné spracovať viacero rezov súčasne. Pri spracovaní viacerých rezov súčasne sa odľahčí rasterizačná jednotka, ktorá tak nemusí počítať textúrové súradnice a dodatočné interpolované hodnoty pre všetky spracovávané rezy, ale iba pre jeden. Taktiež sa zníži počet OpenGL volaní.

Samotné využitie tejto techniky si vyžaduje vykonať drobné zmeny v OpenGL volaniach, konkrétne vo volaniach získavajúcich výsledky z fragmentového programu. Potrebné je zväčšiť rozmer 3D textúry v smere Z o počet spracovávaných rezov súčasne (ozn. $\#mrt$) znížený o jeden. Taktiež je potrebné adekvátne zvýšiť počet 2D textúr reprezentujúcich spracovávané rezy. Zmení sa krok určujúci renderovanie príslušného rezu z 3D textúry. Tento krok zodpovedá aktuálne použitému počtu renderovacích oblastí. Zmeny sa dotknú aj fragmentového programu pre konkrétny výpočet. Pri výpočtoch, ktoré využívajú čítanie dát iba v rámci jedného rezu (napr. bodové operácie) je tento výpočet opakovaný pre každý spracovávaný rez ($\#mrt$). Časť z možného fragmentového programu je na výpise 3.5.

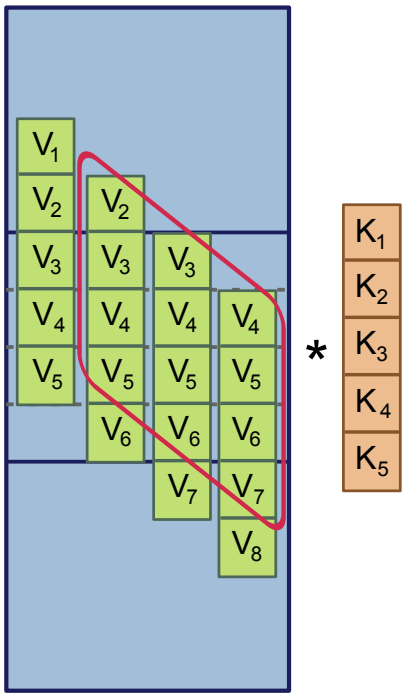
Výraznejšia zmena nastane vo fragmentovom programe využívajúcom dáta aj zo susedných rezov (napr. konvolúcia v smere Z). Pri štandardnom postupe by sa pre každý spracovávaný rez načítali hodnoty zo susedných rezov, pričom by vznikali duplicitné čítania dát v závislosti od počtu potrebných rezov (obrázok 3.6). Nakoľko čítanie z textúry je časovo náročná operácia, je vhodné toto duplicitné čítanie dát odstrániť. Jednou z možností je predčítať potrebné dáta, pokiaľ to výpočet umožňuje (fixný počet dát, dostatok registrov na GPU), prípadne využiť iné optimalizácie. Tieto optimalizácie sú prispôsobené konkrétnym algoritmom a budú predstavené nižšie.

3.3.5 Balenie voxelov

Táto technika je založená na štandardnej vlastnosti OpenGL, vo využití viackanálových – farebných – textúr. Súčasné GPU podporujú viackanálové textúry s rôznym počtom kanálov. Najčastejšie využívané sú štvorkanálové textúry reprezentujúce farebné hodnoty RGB (Red, Green, Blue) a priehľadnosť A (Alpha). Ako už bolo spomenuté, čítanie z textúr je časovo náročná operácia a preto je vhodné počet čítaní z textúr minimalizovať. Jednou možnosťou je zoskupovať vstupné dáta po štvoriciach. Týmto sa docieli získanie štyroch hodnôt na jedno čítanie z textúry, ktoré reprezentujú štyri voxely zo vstupných

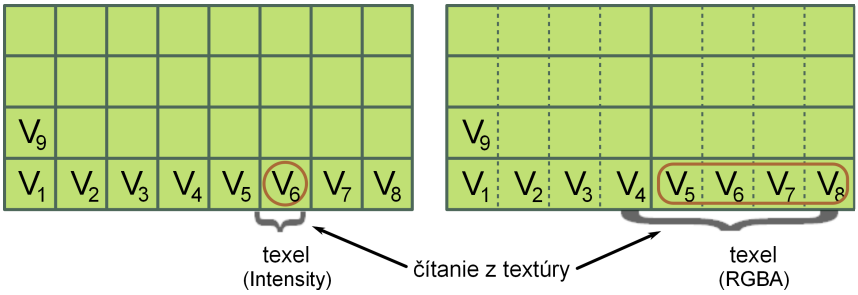
```
1   ...
2   //načítaj voxel z každého spracovávaného rezu
3   sliceVoxel0 = texture3D(slice0, volPos);
4   sliceVoxel1 = texture3D(slice1, volPos+vec3(0,0,1));
5   sliceVoxel2 = texture3D(slice2, volPos+vec3(0,0,2));
6   sliceVoxel3 = texture3D(slice3, volPos+vec3(0,0,3));
7
8   //spracuj dáta, napr. prahovanie
9   result0 = step(threshold, sliceVoxel0);
10  result1 = step(threshold, sliceVoxel1);
11  result2 = step(threshold, sliceVoxel2);
12  result3 = step(threshold, sliceVoxel3);
13
14  //zapíš viaceré spracované rezy
15  gl_FragData[0] = result0;
16  gl_FragData[1] = result1;
17  gl_FragData[2] = result2;
18  gl_FragData[3] = result3;
```

Výpis 3.5: Časť z možného fragmentového programu spracovávajúceho viaceré (4) rezy súčasne pre bodovú operáciu prahovania.



Obrázok 3.6: Proces výpočtu lokálnej operácie v smere Z využívájúci techniku zapisovania do viacerých (4) renderovacích oblastí súčasne. Načrtnutá je lokálna operácia pracujúca s piatimi rezmi. Červenou farbou sú znázornené duplicitné čítania voxelov.

dát. Balenie je možné vykonať v smere X alebo Y na vstupnom reze (obrázok 3.7). Tento proces si vyžaduje zarovnanie posledných voxelov v riadku nulami pokiaľ veľkosť spracovávaného objemu nie je v smere X násobkom 4. Objem prenosu dát medzi CPU a GPU je takmer nezmenený až na prípadné zarovnanie. Balenie dát si nevyžaduje špeciálnu úpravu OpenGL volaní. Stačí pri nastavovaní textúr znížiť rozmer v smere X štvornásobne a modifikovať formáty textúr z *GL_INTENSITY* na *GL_RGBA*. Výraznejšie zmeny sa však dotknú fragmentových programov, hlavne výpočtov v smere X, kde je potrebné brať ohľad na balenie voxelov.



Obrázok 3.7: Balenie vstupných voxelov v smere X po štyroch na redukciu počtu čítaní z textúry. Textúra vľavo s rozmerom 8×4 texelov a formátom *Intensity*. Textúra vpravo využívajúca techniku balenia voxelov s rozmerom 2×4 texelov a formátom *RGBA*

V tejto časti sme uviedli viaceré všeobecné techniky, ktoré rozširujú základný prístup prúdového spracovania objemových dát. Tieto techniky je potrebné prispôsobiť danému algoritmu a jeho špecifikám. Uvedené techniky je možné navzájom kombinovať za účelom dosiahnutia urýchlenia spracovávania dát.

3.4 Filtre

Počítačová grafika a hlavne jej oblasť spracovanie obrazu využíva rôzne operácie za účelom vylepšenia obrazu (odstránenie šumu, ostrenie hrán a iné). V oblasti spracovania a vizualizácie objemových dát sa tieto procesy tiež využívajú, pričom sú aplikované na jednotlivé rezy samostatne alebo priamo na celé dáta vo všetkých troch smeroch. Týmto operáciám spracovávajúcim obraz (objem) za účelom získania nového obrazu (objemu) hovoríme filtre a výpočet je označovaný ako filtrácia. Medzi hlavné filtre, ktoré sú využívané v spracovaní obrazu patrí výpočet konvolúcie. Konvolúcia môže byť využitá na rôzne účely. Pomocou konvolúcie vieme v spracovávanom obraze odstrániť šum, vyhladiť ho, prípadne ostriť hrany a iné. Tento proces je algoritmicky jednoduchý, ale v dôsledku veľkého objemu dát je časovo náročný, a preto je vhodné využiť výpočtový výkon GPU na jeho urýchlenie. Ďalším výpočtom, ktorým sa budeme zaoberať v tejto časti je filter detekujúci tubulárne štruktúry (cievy), ktorý je používaný v medicínskych aplikáciách. Predstavíme jednotlivé implementácie týchto filtrov spolu s využitím vyššie prezentovaných techník.

3.4.1 Konvolúcia

Konvolúcia je často využívaný proces v oblasti spracovania obrazu a taktiež aj v spracovaní objemových dát. Je založená na matematickom procese spracovania signálu, kde z dvoch vstupných signálov (funkcií) získavame tretí výstupný podľa daných kritérií. Nakoľko v počítačovej grafike sa väčšinou pracuje s diskrétnymi dátami, tak je proces konvolúcie definovaný na diskrétnych dátach v jednorozmernom prípade nasledovne

$$(f * g)(x) = f'(x) = \sum_{0 \leq i < d} f(x+i)g(i), \tag{3.1}$$

kde f je dátová funkcia a g je konvolučné jadro (ďalej len jadro). Bez ujmy na všeobecnosti predpokladajme, že g je definované na intervale $\{0, ..., d-1\}$ a f je definovaná na intervale $\{0, ..., n-1\}$, kde $d, n \in \mathbb{N}$. Označenie $*$ je zaužívané označenie pre konvolúciu. Pre trojrozmerný prípad je konvolúcia definovaná nasledovne

$$(f * g)(x, y, z) = f'(x, y, z) = \sum_{0 \leq i,j,k < d} f(x+i, y+j, z+k)g(i, j, k), \tag{3.2}$$

kde f je definovaná na intervale $\{0, ..., n-1\} \times \{0, ..., n-1\} \times \{0, ..., n-1\}$ a jadro g na intervale $\{0, ..., d-1\} \times \{0, ..., d-1\} \times \{0, ..., d-1\}$. Ak pre funkciu g platí

$$g(i, j, k) = g_1(i)g_2(j)g_3(k), \tag{3.3}$$

tak výslednú konvolúciu môžeme rozložiť na postupnosť troch konvolúcií aplikovaných v príslušných smeroch, pre ktoré platí:

$$\begin{aligned} f_1(x, y, z) &= \sum_{0 \leq i < d} f(x+i, y, z)g_1(i) \\ f_2(x, y, z) &= \sum_{0 \leq j < d} f_1(x, y+j, z)g_2(j) \\ f'(x, y, z) &= \sum_{0 \leq k < d} f_2(x, y, z+k)g_3(k). \end{aligned} \tag{3.4}$$

Sústavu rovníc (3.4) nazývame separovateľná konvolúcia. Vo všeobecnosti sa nedá konvolučné jadro transformovať na súčin samostatných funkcii ako vo vzťahu (3.3). V takomto prípade hovoríme o neseparovateľnej konvolúcii (3.2). Separovateľná konvolúcia je výhodnejšia z výpočtového hľadiska, keďže časová zložitosť separovateľnej konvolúcie je $O(3dn^3)$ a neseparovateľnej konvolúcie $O(d^3n^3)$.

Ďalšou možnosťou, ako vypočítať konvolúciu je previesť vstupné funkcie do frekvenčnej oblasti pomocou Fourierovej transformácie, následne spraviť súčin po zložkách a výslednú funkciu previesť späť pomocou spätnej Fourierovej transformácie. Tento prípad sa dá vyjadriť nasledovne

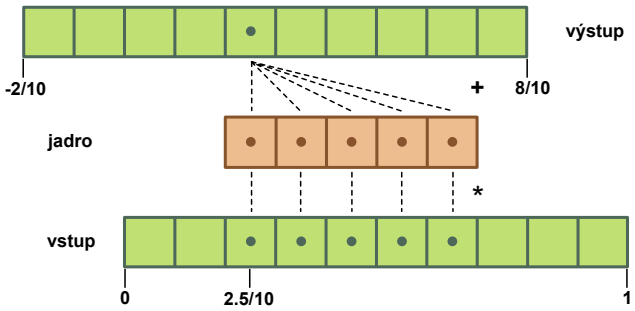
$$\mathcal{F}^{-1}\{f * g\} = c\mathcal{F}\{f\}\mathcal{F}\{g\}, \tag{3.5}$$

kde $\mathcal{F}$ označuje Fourierovu transformáciu a $\mathcal{F}^{-1}$ označuje spätnú Fourierovu transformáciu. Konštanta c závisí od špecifickej normalizácie Fourierovej transformácie. Viac o matematickom pozadí konvolúcie, Fourierovej transformácie a ich podmienkach je možné nájsť v [Bra99]. Výpočet konvolúcie pomocou Fourierových transformácií má opodstatnenie pri neseparovateľných filtroch s veľkými rozmermi, nakoľko samotný prevod do frekvenčnej oblasti a späť predlžuje výsledný čas výpočtu (zložitosť algoritmu pre rýchlu Fourierovu transformáciu je $O(n^3/log(n^3))$. Taktiež, je to algoritmus globálneho charakteru, ktorý potrebuje pristupovať ku všetkým dátam pre výpočet hodnoty jedného voxela.

Výpočet konvolúcie je možné aplikovať na rôznych grafických kartách s rôznou úrovňou podpory. Na výpočet konvolúcie bolo možné využiť už grafické karty podporujúce OpenGL rozšírenie *EXT_convolution*, ktoré bolo uvedené v roku 2002 a podporovalo výpočet jedno/dvoj rozmernej konvolúcie, či už separovateľnej alebo neseparovateľnej. V súčasnosti však rozšírenie *EXT_convolution* nie je podporované výrobcami a tým je prakticky nepoužiteľné. Z nášho pohľadu ďalšou nevýhodou tohto rozšírenia je, že nepodporuje trojrozmernú konvolúciu. Prvým využitím profesionálnych grafických kariet na filtrovanie sa venoval Ertl v [HE99], kde využíva tento hardvér na trojrozmerné filtrovanie pomocou OpenGL rozhrania. Ďalším využitím už bežne dostupných GPU na filtrovanie sa venovali [HTHG01, Bjo04], kde využívali programovateľné fragmentové programy, prípadné ich predchodcu *register combiners*. Vyššie uvedené práce filtrovali textúry či už dvojrozmerné alebo trojrozmerné a používali ich ďalej na textúrovanie, prípadne na objemové zobrazovanie. V prácach [MA03, SL05] využívajú GPU na výpočet rýchlej Fourierovej transformácie pri spracovaní obrazu. Tieto postupy je možné upraviť a využiť na spracovanie objemových dát a spracované výsledky uložiť na disk. Vyššie spomínané metódy pracujú s celými dátami uloženými v pamäti grafickej karty, keďže sa využívajú hlavne na objemové zobrazovanie, čo nie je náš prípad. Náš predkladaný algoritmus je založený na prúdovom spracovaní, aby minimalizoval pamäťové nároky a tým umožnil spracovanie takmer ľubovoľne veľkých dát.

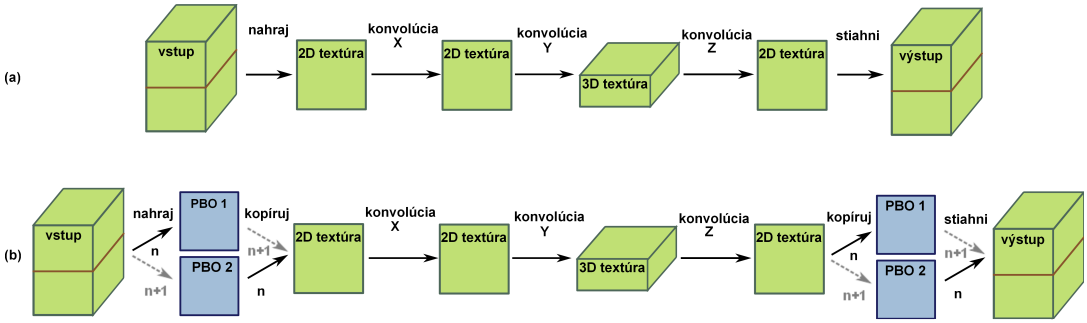
3.4.1.1 Separovateľná konvolúcia s využitím OpenGL podpory

Základný algoritmus trojrozmernej separovateľnej konvolúcie využíva tri typy textúr, jedno, dvoj, troj - rozmerné, ako základné dátové štruktúry na ukladanie dát. Jednorozmerné textúry sa využívajú na uloženie hodnôt jadra. Pre každý smer je použitá samostatná textúra, keďže v každom smere môžu byť použité rôzne jadrá. Pre výpočet konvolúcie na vstupnom reze sú potrebné dvojrozmerné textúry, v ktorých sú uložené medzivýsledky z jednotlivých konvolúcií v smeroch X a Y. Na výpočet konvolúcie v smere Z je potrebná trojrozmerná textúra, ktorá slúži na ukladanie spracovaných rezov. Veľkosť trojrozmernej textúry v smere Z je rovnaká ako rozmer jadra v tomto smere. Nastavenie textúrových súradníc zodpovedá veľkosti jadra, pričom sú posunuté v kladnom smere o pol texela (stred texela) a v opačnom smere o polomer jadra (obrázok 3.8). Toto riešenie je efektívnejšie ako vykonávanie úpravy textúrovej súradnice jednotlivo pre každý fragment vo fragmentovom programe.



Obrázok 3.8: Nastavenie textúrových súradníc pre jednorozmerný prípad výpočtu konvolúcie v smere X a veľkosti jadra 5.

Samotný výpočet konvolúcie prebieha vo fragmentových programoch, kde je na vstupe dvojrozmerná, respektíve trojrozmerná textúra s dátami a jednorozmerná textúra s jadrom. Na obrázku 3.9(a) je zobrazený základný postup výpočtu separovateľnej konvolúcie. Na výpise 3.10 je možné vidieť výpočet konvolúcie v smere X. Pre výpočet konvolúcie v smere Y a Z je daný fragmentový program patrične upravený.



Obrázok 3.9: Zobrazenie rôznych verzií prenosu dát pri výpočte separovateľnej konvolúcie. Pôvodná technológia pomocou synchrónnych volaní (a). Prúdová verzia asynchrónneho prenosu dát (b).

Asynchrónny prenos dát

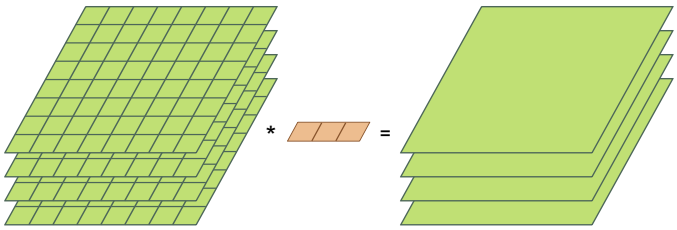
Na využitie techniky asynchrónneho prenosu dát je potrebné predošlý postup rozšíriť o PBO objekty. Výpočtový proces začína asynchrónnym nahratím rezu do príslušného PBO, odkiaľ je nakopírovaný do 2D textúry. Následne je vykreslený štvorec alebo obdĺžnik reprezentujúci daný rez. Pri vykresľovaní prebehne výpočet konvolúcie v smere X. Výsledné hodnoty sú zapísané priamo do ďalšej 2D textúry s využitím FBO rozšírenia. Rovnakým spôsobom prebehne výpočet aj v smere Y. Výsledný rez je priamo zapísaný na príslušnú pozíciu do 3D textúry, ak to GPU podporuje. Pri nedostatočnej podpore je výsledok zapísaný do 2D textúry, odkiaľ je nakopírovaný do 3D textúry. Konvolúcia v smere Z prebehne obdobným spôsobom, pričom výsledok je zapísaný do 2D textúry. Odtiaľ je nakopírovaný do príslušného PBO a asynchrónne stiahnutý na CPU. Celý proces je zobrazený na obrázku 3.9(b).

```
1  uniform sampler2D slice;     //vstupný rez
2  uniform sampler1D kernel;   //konvolučné jadro
3  uniform float kerposadd;     //rozmer texela jadra
4  uniform float volposadd;     //rozmer texela rezu v smere x
5
6  void main()
7  {
8     float result = 0.0;
9
10    vec2 volpos = gl_TexCoord[0].xy;  //pozícia prvého voxela
11    float kerpos = 0.5*kerposadd;        //pozícia prvej hodnoty jadra
12
13    while (kerpos < 1.0)
14    {
15       float kerVoxel = texture1D(kernel, kerpos).x;
16       float volVoxel = texture2D(slice, volpos).x;
17       kerpos += kerposadd;    //posun na daľšiu hodnotu jadra
18       volpos.x += volposadd;  //posun na daľší voxel
19
20       result += volVoxel*kerVoxel;  //výpočet
21    }
22    gl_FragData[0] = result.xxxx;
23 }
```

Výpis 3.10: Základný fragmentový program pre výpočet separovateľnej konvolúcie v smere X.

Spracovanie viacerých rezov súčasne

Nakoľko výpočet separovateľnej konvolúcie sa skladá z troch samostatných častí, tak sa využitie tejto techniky dotkne každej z nich. Keďže spracovávame viacero rezov naraz, ktorých počet je rovnaký ako počet renderovacích oblastí ($\#mrt$), je výpočet zahájený až po načítaní dostatočného počtu rezov už pri výpočte konvolúcie v smere X. Do fragmentových programov pre výpočet konvolúcie v smere X a Y vstupuje $\#mrt$ rezov, ktoré sú spracované v príslušnom smere (obrázok 3.11). Výsledok je ukladaný do 3D textúry pre výpočet konvolúcie v smere Z. Pri spracovávaní v smere X a Y sa dáta (voxely) čítajú zvlášť pre každý rez, pričom hodnoty jadra sú čítané práve raz pre všetky súčasne spracovávané rezy.



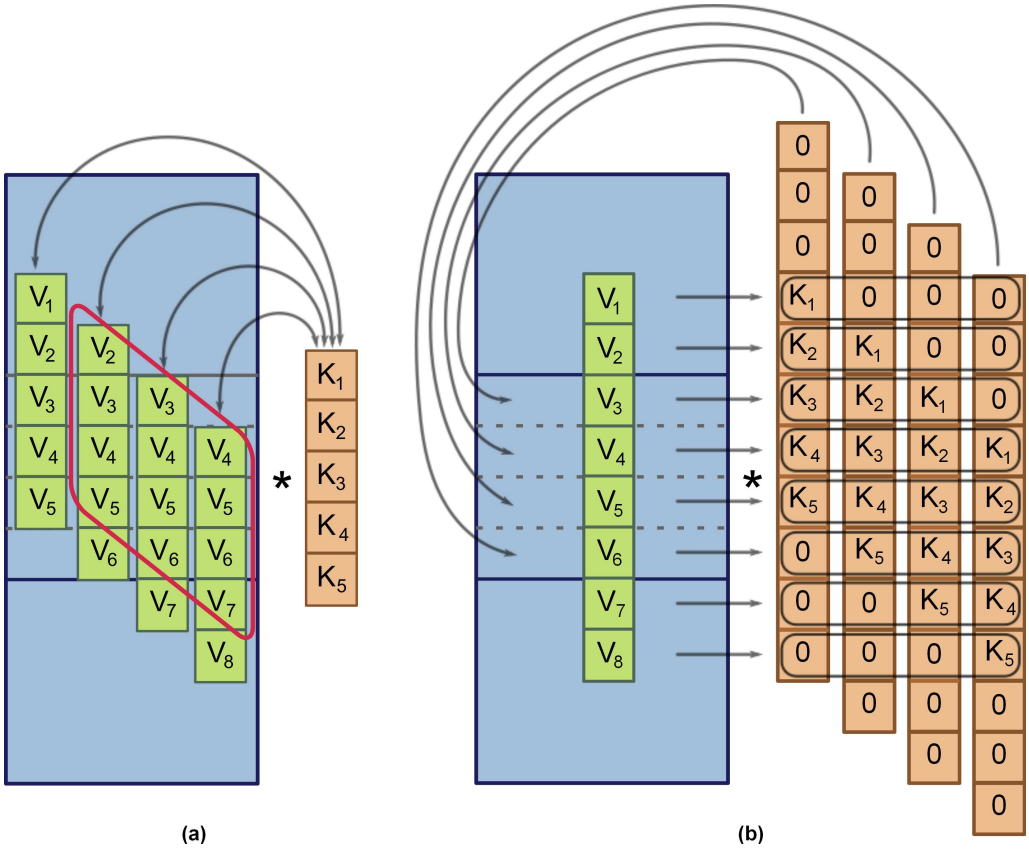
Obrázok 3.11: Proces výpočtu separovateľnej konvolúcie pri spracovaní viacerých (4) rezov súčasne v smere X.

Po spracovaní dostatočného počtu rezov v smere X,Y prebehne výpočet konvolúcie v smere Z (obrázok 3.12(a)). Ako bolo uvedené v časti 3.3, je potrebné minimalizovať duplicitné čítania z textúry, a to napríklad pomocou predčítania iba potrebných hodnôt. Tento prístup je v prípadne všeobecnej separovateľnej konvolúcie nepoužiteľný, pretože pri väčších rozmeroch jadier by dochádzalo k nedostatku registrov potrebných na uloženie týchto hodnôt. Preto sme navrhli metódu rozšíreného konvolučného jadra. Konvolučné jadro sa rozšíri o $\#mrt$ nulových hodnôt na začiatku a na konci. Toto nové rozšírené jadro bude mať veľkosť

$$dz_{new} = dz + 2(\#mrt - 1),$$

kde *dz* je veľkosť pôvodného jadra v smere Z. Vo fragmentovom programe sa dáta prechádzajú cyklom v smere Z a načíta sa potrebný voxel práve raz. Pre každý voxel je použitých $\#mrt$ hodnôt jadra, každá posunutá o jeden texel v smere Z. Každá posunutá hodnota jadra zodpovedá ďalšiemu spracovávanému rezu. Voxely sú korektne násobené hodnotou jadra pre daný rez, prípadne nulovou hodnotou rozšíreného jadra pre voxely potrebné na spracovanie ďalších rezov. Použitie rozšíreného konvolučného jadra je načrtnuté na obrázku 3.12(b).

Uvedeným spôsobom sa obmedzí počet čítaní z 3D textúry na minimum. Čítanie z konvolučného jadra je viacnásobné ($(\#mrt)$-násobné), čo sa dá odstrániť použitím cyklického posunutia hodnôt vo vektore zodpovedajúcemu jadru doprava (*swizling*) a nahradením prvej zložky novou hodnotou jadra. Na výpise 3.13 je časť z fragmentového programu spracovávajúceho štyri rezy v smere Z.



Obrázok 3.12: Proces výpočtu separovateľnej konvolúcie pri použití renderovania do viacerých oblastí súčasne (4) v smere Z. Štandardný prístup (a), kde červená oblasť predstavuje duplicitné čítania voxelov. Použitie rozšíreného konvolučného jadra (b), kde sa potrebné voxely čítajú práve raz.

```
1  ...
2  //načítanie hodnôt jadra,
3  // každá zložka vektora je použitá pre príslušný rez
4  vec4 kerVoxel;
5  kerVoxel.x = texture1D(kernel, (3.5)/kerdim );
6  kerVoxel.y = texture1D(kernel, (2.5)/kerdim );
7  kerVoxel.z = texture1D(kernel, (1.5)/kerdim );
8  kerVoxel.w = texture1D(kernel, (0.5)/kerdim );
9
10 for(i=0; i <= voldim+MRT; i++)
11 {
12    volVoxel = texture3D(volume, volpos);
13
14    //výpočet konvolúcie, voxel je prenásobený príslušným jadrom
15    result.xyzw += kerVoxel.xyzw*volVoxel.xxxx;
16
17    //posun na ďalší voxel
18    volpos.z += volposadd;
19
20    //posun hodnôt jadra pre spracovanie ďalšieho voxela
21    kerVoxel.yzwx = kerVoxel.xyzw; //swizling
22    kerVoxel.x = texture1D(kernel, (4.5+i)/kerdim );
23 }
24 ...
25 gl_FragData[0] = result.x;
26 gl_FragData[1] = result.y;
27 gl_FragData[2] = result.z;
28 gl_FragData[3] = result.w;
```

Výpis 3.13: Časť z fragmentového programu pre výpočet separovateľnej konvolúcie v smere Z spracovávajúceho štyri rezy súčasne.

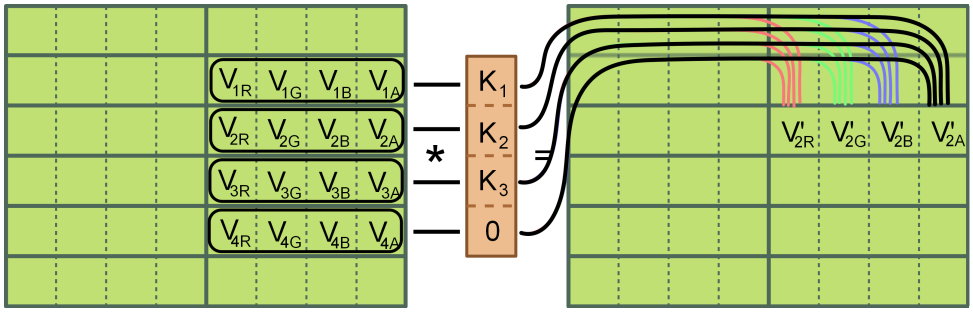
Balenie voxelov

Hlavným prínosom tejto techniky je využitie podpory výpočtov na vektoroch. GPU podporuje čítanie štvorrozmerných vektorov z textúry a operácie na nich. Operácie na vektoroch sú rýchlejšie ako tie isté operácie na skalárnych dátach vykonávaných 4-krát, pretože sú podporované priamo v inštrukčnej sade GPU. Pri takomto balení vstupných dát je potrebné upraviť fragmentové programy tak, že jedným výpočtom konvolúcie sa spracujú štyri voxely súčasne. Tieto voxely ležia v jednom reze (v smere balenia), pričom predošlá technika spracovávala súčasne voxely z viacerých rezov v smere Z.

V smere Y a Z sú fragmentové programy jednoduchšie. Vonkajší cyklus prechádzajúci po textúre reprezentujúcej jadro je rozšírený o vnútorný cyklus vykonávaný štyrikrát na texely zodpovedajúcemu zbalenému jadru. V tomto cykle sa načítava texel (4 voxely) zo vstupného rezu a prenásobí sa hodnotou z prvého kanálu textúry reprezentujúcej jadro. Následne nastáva cyklické posunutie hodnôt jadra doprava (*swizling*). Taktiež nastáva posun textúrovej súradnice pre čítanie nasledujúceho texela zo vstupného rezu. Pseudokód reprezentujúci tento výpočet je uvedený na výpise 3.14 a na obrázku 3.15 je tento proces zobrazený.

```
1  ...
2  //prechádzanie po texeloch jadra
3  while (kerpos < 1.0)
4  {
5    //načítaj zbalené jadro
6    vec4 kerVoxels = texture1D(kernel, kerpos);
7    kerpos+=kerposadd; //posun na ď alší voxel jadra
8
9    //pre každý voxel jadra
10   for(i=0; i<4; i++)
11   {
12     //načítaj štyri vstupné voxely
13     volVoxels = texture2D(slice, volpos);
14
15     //výpočet konvolúcie na štyroch susedných voxeloch
16     result.xyzw += kerVoxels.xxxx*volVoxels.xyzw;
17
18     //posun jadra dol ava
19     kerVoxels.xyzw = kerVoxels.yzwx; //swizling
20     //posun na ď lšie voxely v smere Y
21     volpos.y += volposadd;
22   }
23 }
24 ...
```

Výpis 3.14: Časť z fragmentového programu využívajúci balenie voxelov pre výpočet separovateľnej konvolúcie v smere Y.



Obrázok 3.15: Proces výpočtu separovateľnej konvolúcie v smere Y, respektíve Z, pri použití balenia voxelov a jadra o veľkosti 3.

V smere X je situácia o niečo komplikovanejšia, keďže dochádza k výpočtu v rámci jedného rezového texela (štyri voxely) a medzi viacerými rezovými texelmi súčasne. Hlavnou využívanou funkcionalitou je skalárny súčin jadra a rezového texela, preusporiadanie voxelov v rámci jedného rezového texela a taktiež doplnenie voxelov z nasledujúceho rezového texela. Pre využitie skalárneho súčinu musia byť správne „napasované“ hodnoty medzi jadrom a jednotlivými voxelmi z čítaného texela. Nakoľko je jednoduchšie predspracovať hodnoty v konvolučnom jadre, upravili sme jeho veľkosť nasledovne

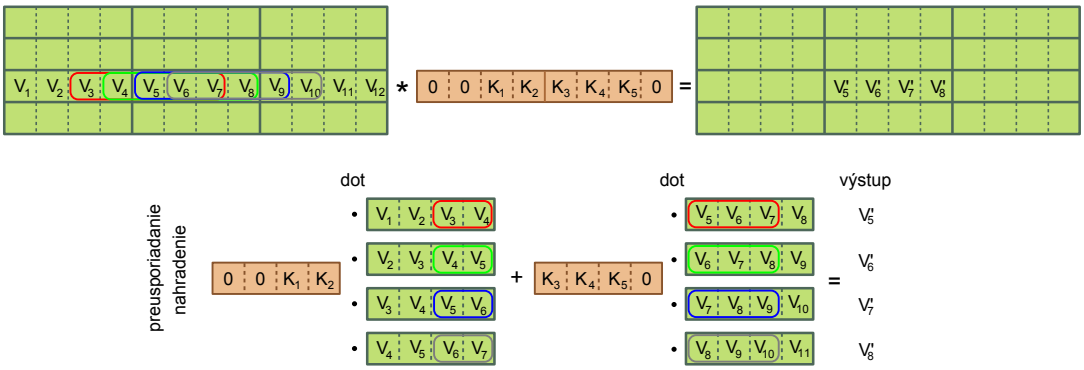
$$sizex_{new} = (rx - 1) >> 2 + rx >> 2 + 2,$$

kde rx je polomer veľkosti jadra v smere X a $>>$ je operácia bitového posunu vpravo. Taktiež je potrebné posunúť začiatočné hodnoty jadra, pričom prvej pozícii zodpovedá hodnota $(4 - (rx)\%4)\%4$, kde $\%$ označuje opráciu modulo. Zostávajúce hodnoty sú nastavené na nuly. Na obrázku 3.16 sú znázornené rôzne nastavenia jadra pre rôzne vstupné veľkosti. Uvedený posun zabezpečí, že stredná hodnota z pôvodného jadra sa nachádza vždy na prvom mieste jednej zo štvorice zodpovedajúcej novému jadru podľa jeho veľkosti. Týmto posunom docielime, že prvý spracovávaný rezový texel je správne zarovnaný s texelom z jadra, čo vedie ku korektnému skalárnemu súčinu. Výsledok tohto skalárneho súčinu je čiastočným výsledkom celej konvolúcie pre prvý výstupný kanál. Pre výpočet čiastočných výsledkov pre ďalšie kanály (voxely) je potrebné hodnoty v aktuálnom rezovom texeli posunúť doľava a posledný kanál nahradiť hodnotou z prvého kanála nasledujúceho rezového texela, novým voxelom. Po tomto úkone nastáva správne zarovnanie rezového texela s texelom z jadra. Tento proces posunutia a nahradenia sa vykoná štyrikrát pre každý výstupný kanál. Celý výpočet sa vykonáva podľa veľkosti konvolučného jadra. Jednoduchý náčrt vykonávaných operácií je zobrazený na obrázku 3.17 a na výpise 3.18 je znázornený príslušný pseudokód.

3.4.1.2 Neseparovateľná konvolúcia s využitím OpenGL podpory

Neseparovateľná konvolúcia vykonávaná na GPU za pomoci OpenGL rozhrania vychádza z vyššie opísanej separovateľnej konvolúcie. Konvolučné jadro je reprezentované 3D textúrou. Všetky vyššie spomínané techniky je potrebné prispôsobiť trojrozmernému

Obrázok 3.16: Nastavenia vlastností konvolučného jadra využívaného pri výpočte separovateľnej konvolúcie v smere X pomocou balenia voxelov.



Obrázok 3.17: Proces výpočtu separovateľnej konvolúcie v smere X pri použití techniky balenia voxelov.

```
1  ...
2  //načítaj štyri susedné voxely
3  vec4 volVoxels = texture2D(slice, volpos);
4
5  //prechádzaj po zbalených hodnotách jadra
6  while(kerpos < 1.0)
7  {
8     //načítaj hodnoty jadra
9     kerVoxels = texture1D(kernel, kerpos);
10    kerpos += kerposadd;   //posun na ďalšie štyri hodnoty jadra
11    volpos.x += volposadd; //posun na ďalšie štyri voxely
12
13    //načítaj ďalšiu štvoricu vstupných voxelov
14    volVoxels1 = texture2D(slice, volpos);
15
16    //výpočet pre prvý kanál
17    result.x += dot(kerVoxels, volVoxels);
18    volVoxels.x = volVoxels1.x; //nový voxel zo susednej štvorice voxelov
19
20    //posun voxelov a výpočet pre druhý kanál
21    result.y += dot(kerVoxels, volVoxels.yzwx);
22    volVoxels.y = volVoxels1.y; //nový voxel zo susednej štvorice voxelov
23
24    //posun voxelov a výpočet pre tretí kanál
25    result.z += dot(kerVoxels, volVoxels.zwxy);
26    volVoxels.z = volVoxels1.z; //nový voxel zo susednej štvorice voxelov
27
28    //posun voxelov a výpočet pre štvrtý kanál
29    result.w += dot(kerVoxels, volVoxels.wxyz);
30    volVoxels.w = volVoxels1.w; //nový voxel zo susednej štvorice voxelov
31 }
32 ...
```

Výpis 3.18: Časť z fragmentového programu výpočtu separovateľnej konvolúcie v smere X využívajúci techniku balenia voxelov.

prípadu. Príslušné fragmentové programy obsahujú viac inštrukcií a tri vnorené cykly. Zjednodušenie výpočtového procesu nastáva pri volaní OpenGL funkcií, kde sa výpočet konvolúcie vykonáva v jednom fragmentovom programe (obrázok 3.4(c)).

V tejto časti sme podrobnejšie uviedli techniky využívané pri výpočte trojrozmernej konvolúcii, ktoré je možné kombinovať. Výsledky z meraní sú uvedené v časti 3.5.

3.4.2 Detekcia tubulárnych štruktúr

V tejto časti sa budeme venovať filtru a jeho implementácii na GPU, ktorý je využívaný v medicínskej oblasti na detekciu tubulárnych štruktúr (ciev) v CT dátach. Na detekciu využívame filter uvedený v [FNVV98]. Tento filter je založený na analýze vlastných čísel matice druhých derivácií (hessian) pre každý voxel. Výsledkom analýzy je pravdepodobnosť nakoľko daný voxel reprezentuje tubulárnu štruktúru. Na odhad druhých derivácii používame metódu stredových rozdielov druhého rádu. Následne je vykonaný výpočet vlastných čísel z matice druhých derivácií. Na všeobecný výpočet vlastných čísel sa používajú iteračné metódy, ktoré sú ale časovo náročné [GNU10]. Nájdenie vlastných hodnôt matice druhých derivácií vedie k problému riešenia rovnice tretieho stupňa, ktorý nie je vo všeobecnosti jednoznačný. Existuje však aj neiteračná metóda odhadujúca korene rovnice tretieho stupňa [Ebe08], ktorú použijeme v našej implementácii na GPU. Táto metóda je univerzálna a ľahko implementovateľná na CPU alebo GPU. Použitá metóda vyžaduje, aby bola matica druhých derivácií symetrická, čo je v našom prípade splnené.

3.4.2.1 Implementácia na GPU

Vyššie uvedený výpočet matice druhých derivácií a vlastných čísel je vykonávaný v jednom fragmentovom programe. Následne sú výsledné hodnoty stiahnuté na CPU, kde prebehne výpočet pravdepodobnosti umiestnenia ciev. Na výpočet druhých derivácií je potrebných 5 vstupných rezov, čomu zodpovedá aj rozmer 3D textúry. Zmena oproti všeobecnému postupu spracovania objemových dát na GPU uvedenom v časti 3.3 nastáva pri výstupných hodnotách, kde sú generované tri vlastné čísla pre každý voxel. V technikách uvedených vyššie sme predpokladali pre vstupný voxel jednu výstupnú hodnotu, spracovaný voxel.

Pri tomto výpočte je taktiež možné využiť všeobecné techniky predstavené v predchádzajúcich častiach. Treba ich však prispôsobiť počtu výstupných hodnôt, čo tieto techniky trochu modifikuje. Asynchrónny prenos dát je obdobný ako pri všeobecných postupoch (obrázok 3.4(c)). Využitie MRT techniky na spracovanie viacerých rezov súčasne je tiež len modifikáciou všeobecného prístupu. Keďže pri tomto výpočte je potrebný fixný počet rezov (5), je možné potrebné voxely načítať v inicializačnej fáze a zamedziť tak duplicitnému čítaniu dát. Pri použití techniky balenia voxelov sú vstupné aj výstupné dáta zoskupované do štvoríc. Pri výstupných hodnotách však nastáva komplikácia,

ktorá spočíva v tom, že výpočet generuje tri výstupné hodnoty. Preto je pri tejto technike potrebné zároveň využiť aj MRT techniku, kde sa výsledky zapisujú do viacerých renderovacích oblastí súčasne. Pre jeden spracovávaný rez vo forme RGBA textúry (balenie voxelov) sa generujú tri výstupné RGBA textúry, kde každá textúra obsahuje zbalené príslušné vlastné čísla. Pri kombinovaní týchto techník rýchlo narazíme na limit pre počet renderovacích oblastí (16). Pri použití balenia voxelov a štyroch spracovávaných rezov súčasne je na výstup potrebných dvanásť trojkanálových textúr.

Do výpočtu vlastných čísel vo fragmentovom programe je možné priamo zakomponovať aj výpočet pravdepodobnosti tubulárnych štruktúr. Pri takejto implementácii by sa z fragmentového programu získavalo len jedno číslo (pravdepodobnosť) pre každý voxel. Táto situácia by sa potom podobala výpočtu konvolúcie (získavanie výsledkov) a dali by sa využiť vo vyššej miere limity GPU. Avšak pre dodatočný výskum v tejto oblasti je výhodné získavať priamo vlastné čísla a používať ich na dodatočnú analýzu. V rámci fragmentových programov je možné počítať aj vlastné vektory pre matice druhých derivácií a využiť ich taktiež na dodatočnú analýzu. Pri takomto výpočte by sa už pri základnom programe pracujúcom na jednom reze bez balenia voxelov museli použiť tri výstupné RGBA textúry (12 hodnôt). Každá by obsahovala príslušný vlastný vektor a vlastné číslo.

Problémom súčasného riešenia je znížená presnosť, ako bude uvedené v nasledujúcej časti. Všetky dostupné riešenia výpočtu vlastných čísel sú vykonávané v dvojitej presnosti s plávajúcou desatinou čiarkou, prípadne ju odporúčajú [GNU10, Ebe08]. Naša implementácia je však v jednoduchej presnosti, pričom ju nie je problém v budúcnosti previesť do dvojitej presnosti. Dnešné moderné GPU obsahujú aj výpočtové jednotky rátajúce v dvojitej presnosti, avšak tieto jednotky sú pod staršími verziami OpenGL nedostupné. Dostupné sú pomocou nových GPGPU rozhraní, prípadne novej špecifikácie OpenGL 4.0 [SA10] (uvedenej 10. marca 2010). Jednou z možností ako dosiahnuť vyššiu presnosť výpočtu aj na starších GPU je simulácia výpočtov v dvojitej presnosti na dvoch dátových typoch pracujúcich v jednoduchej presnosti. Táto problematika je opísaná v [ORO05], pričom je možné dosiahnuť vyššiu presnosť s istou chybou oproti dvojitej presnosti aj na starších GPU. Samozrejme, tento výpočet má dopad na výsledný výpočtový čas.

3.5 Merania

Vyššie prezentované filtre a techniky využívajúce výkon GPU sme implementovali do balíka *f3d* [ŠD02, f3d10], konkrétne do knižnice *f3dfilter*. Tento balík sa skladá z nasledujúcich knižníc a programov:

- *f3dla* – knižnica slúžiaca na výpočet základných transformácií lineárnej algebry.
- *f3dformat* – knižnica, ktorá poskytuje základnú funkcionalitu na prácu s objemovými dátami vo formáte *f3d*, definovanom v tejto knižnici.

- *f3dfilter* – knižnica, v ktorej sú implementované rôzne časovo náročné výpočty využívajúce hardvérové zdroje počítačov. Konkrétne ide o využitie SSE inštrukcií hlavného procesora, CUDA technológie a OpenGL prostredia využívajúce výkonový potenciál GPU.
- *f3dclass* – knižnica poskytujúca rozšírenú funkcionalitu na prácu s objemovými dátami a k nej jednoduché programy pracujúce v príkazovom riadku.
- *f3dview* – progam zobrazujúci objemové dáta vo formáte *f3d*, vo forme rovinných rezov.
- *f3dvr* – program na interaktívne zobrazovanie objemových dát využívajúci výkon GPU [ŠDSČ04].

V nasledujúcich častiach budú uvedené výsledky meraní, ktoré boli vykonané na demonštráciu časových urýchlení pre vyššie spomínané techniky. Budú tu taktiež prezentované výsledky z ďalších dostupných hardvérových riešení výpočtu filtrov implementovaných v knižnici *f3dfilter*. Všetky prezentované testy boli vykonané na počítači s nasledujúcimi parametrami: Intel QuadCore 2.4GHz, 8GB RAM, 500GB HDD, NVIDIA 9800 GTX s 512MB VRAM, Ubuntu 8.04. Výsledky meraní rýchlosti sú uvedené v sekundách. Všetky testy boli spúšťané trikrát a vo výsledkoch sú prezentované minimálne hodnoty. Pri konvolučných jadrách s menším rozmerom je bežné, že nameraná hodnota je vo veľkej miere ovplyvnená pevným diskom. Pomalý pevný disk spôsoboval, že rozdiely medzi rôznymi hardvérovými variantmi a veľkosťami jadier boli minimálne. Pre testy boli preto vstupné dáta generované pomocou špeciálneho nástroja bez diskových operácií a spracované dáta neboli ukladané na disk, ale do pseudosúboru (/dev/null), aby sa zamedzilo jeho vplyvu na merania.

3.5.1 OpenGL algoritmy

V tejto časti budú uvedené výsledky z meraní rôznych techník využívajúcich OpenGL prostredie, prezentovaných v predchádzajúcich častiach.

3.5.1.1 Neseparovateľná konvolúcia

Základná verzia algoritmu bez asynchrónneho prenosu dát nie je v balíku *f3dfilter* implementovaná nakoľko využíva zastaranú funkcionalitu. Preto budú navzájom porovnávané len jednotlivé techniky. Všetky využívajú asynchrónny prenos dát a pri výpočte konvolúcie techniku rozšíreného konvolučného jadra. V tabuľke 3.1 sú uvedené výsledky z techník využívajúcich spracovanie viacerých rezov súčasne (ozn. *Mrt1* - pre jeden, *Mrt4* - pre štyri, *Mrt8* - pre osem rezov) s balením voxelov (ozn. *Pack*) alebo bez balenia voxelov (ozn. *NonPack*). Ako je vidieť z uvedenej tabuľky, balenie voxelov, okrem prípadu najmenšieho jadra, prinieslo výrazné urýchlenie výpočtov oproti verzii bez balenia voxelov. Toto urýchlenie bolo v priemere 2.7 násobné. Z prezentovaných výsledkov

vidieť, že využitie ôsmich renderovacích oblastí je až na jeden prípad najpomalšie, pre obe verzie (*Pack*, *NoPack*). Môžeme taktiež skonštatovať, že rozdiely medzi verziami *Mrt1* a *Mrt4* sú závislé na veľkosti testovaného objemu a konvolučného jadra. Pre menšie rozmery je výhodnejšie použiť verziu *Mrt1* a pre väčšie hodnoty verziu *Mrt4*. Hranicu, kedy použiť verziu *Mrt1* a kedy verziu *Mrt4* je náročné určiť, nakoľko táto hranica sa môže meniť s použitím inej grafickej karty, prípadne iného ovládača. Taktiež vidíme, že absolútne časové prínosy pre verziu *Mrt1* oproti verzii *Mrt4* sú pre malé jadrá menšie ako pre veľké jadrá. Z týchto dôvodov sme ako najvýhodnejšiu verziu pre pevné nastavenie v knižnici *f3dfilter* zvolili kombináciu využívajúcu balenie voxelov s využitím štyroch renderovacích oblastí (*PackMrt4*). Táto verzia bude používaná aj v nasledujúcich meraniach pri použití neseparovateľnej konvolúcie.

Neseparovateľná konvolúcia

	NonPack					**Pack**				
Mrt1	$\mathbf{3^3}$	$\mathbf{5^3}$	$\mathbf{9^3}$	$\mathbf{15^3}$	$\mathbf{31^3}$	$\mathbf{3^3}$	$\mathbf{5^3}$	$\mathbf{9^3}$	$\mathbf{15^3}$	$\mathbf{31^3}$
$\mathbf{128^3}$	0,100	0,190	0,400	1,430	10,950	0,120	0,150	0,250	0,560	3,480
$\mathbf{256^3}$	0,230	0,550	2,450	10,640	89,670	0,220	0,360	1,080	3,350	25,100
$\mathbf{512^3}$	1,120	3,580	18,710	84,470	729,600	1,030	1,970	7,460	25,390	198,240

Mrt4	$\mathbf{3^3}$	$\mathbf{5^3}$	$\mathbf{9^3}$	$\mathbf{15^3}$	$\mathbf{31^3}$	$\mathbf{3^3}$	$\mathbf{5^3}$	$\mathbf{9^3}$	$\mathbf{15^3}$	$\mathbf{31^3}$
$\mathbf{128^3}$	0,150	0,190	0,410	1,180	7,970	0,150	0,190	0,260	0,490	2,450
$\mathbf{256^3}$	0,320	0,630	2,140	8,140	61,910	0,350	0,470	1,060	2,770	17,770
$\mathbf{512^3}$	1,720	4,040	16,110	63,710	499,650	1,790	2,740	7,530	21,170	142,620

Mrt8	$\mathbf{3^3}$	$\mathbf{5^3}$	$\mathbf{9^3}$	$\mathbf{15^3}$	$\mathbf{31^3}$	$\mathbf{3^3}$	$\mathbf{5^3}$	$\mathbf{9^3}$	$\mathbf{15^3}$	$\mathbf{31^3}$
$\mathbf{128^3}$	0,150	0,230	0,490	1,340	8,400	0,180	0,220	0,400	0,730	3,490
$\mathbf{256^3}$	0,400	0,790	2,730	9,130	65,320	0,520	0,750	1,850	4,550	27,380
$\mathbf{512^3}$	2,290	5,450	19,950	71,400	252,580	2,870	4,430	12,010	30,970	200,900

Tabuľka 3.1: Časy výpočtu neseparovateľnej konvolúcie pre rôzne techniky a parametre. V stĺpcoch sú uvedené verzie bez balenia alebo s balením voxelov a parameter veľkosť jadra. V riadkoch sú uvedené verzie využívajúce rôzny počet renderovacích oblastí a veľkosť spracovávaného objemu. Červenou farbou sú označené najrýchlejšie výpočty medzi rôznymi technikami a parametrami.

3.5.1.2 Separovateľná konvolúcia

Výsledky zo separovateľnej konvolúcie sú rádovo rýchlejšie ako z neseparovateľnej konvolúcie. Aby boli viditeľné rozdiely medzi jednotlivými verziami výpočtov, tak sme pri

týchto meraniach použili aj väčšie vstupné dáta a veľkosti konvolučných jadier. V tabuľke 3.2 sú uvedené merania z príslušných algoritmov. Väčšina záverov uvedených pre neseparovateľnú verziu platí aj pre separovateľnú, avšak rozdiely sú menej viditeľné. Verzia využívajúca osem renderovacích oblastí je vo väčšine prípadov najpomalšia. Takýto výsledok sme neočakávali a je pravdepodobne spôsobený vyššími nárokmi GPU na obsluhu ôsmich renderovacích oblastí, pričom sa urýchlenie dosiahne až pri najnáročnejších výpočtoch. Pre veľké rozmery testovaných objemov a jadier je výhodnejšie použiť verziu využívajúcu balenie voxelov a verziu *Mrt4* alebo *Mrt8*. Pri menších veľkostiach jadier je výhodné požiť verziu *Mrt1* či už využívajúcu balenie voxelov alebo nie. Hranica, určujúca akú techniku je kedy vhodné použiť je nejednoznačná. Jednotlivé verzie sa pri porovnaní absolútnych časov líšia len vo veľmi malých hodnotách. Preto sme aj v separovateľnej verzii ako najvýhodnejšie riešenie v knižnici *f3dfilter* pre danú hardvérovú kombináciu zvolili algoritmus využívajúci balenie voxelov so štyrmi renderovacími oblasťami (*PackMrt4*).

Separovateľná konvolúcia

	NonPack					Pack				
Mrt1	**3**	**7**	**15**	**31**	**63**	**3**	**7**	**15**	**31**	**63**
128^3	0.140	0.130	0.190	0.180	0.180	0.130	0.150	0.170	0.200	0.170
256^3	0.250	0.310	0.250	0.340	0.490	0.270	0.270	0.250	0.280	0.400
512^3	1.190	1.200	1.200	1.690	2.690	1.190	1.180	1.210	1.370	2.000
1024^3	8.970	8.900	9.930	12.050	19.730	8.990	9.000	10.240	10.350	14.400

Mrt4	**3**	**7**	**15**	**31**	**63**	**3**	**7**	**15**	**31**	**63**
128^3	0.140	0.150	0.170	0.180	0.170	0.130	0.150	0.160	0.160	0.170
256^3	0.310	0.280	0.270	0.340	0.430	0.300	0.280	0.270	0.300	0.330
512^3	1.170	1.210	1.240	1.610	2.270	1.200	1.220	1.210	1.420	1.610
1024^3	9.030	9.940	10.130	11.460	16.520	8.980	10.250	10.000	10.030	11.380

Mrt8	**3**	**7**	**15**	**31**	**63**	**3**	**7**	**15**	**31**	**63**
128^3	0.160	0.170	0.160	0.180	0.190	0.180	0.160	0.170	0.170	0.170
256^3	0.290	0.290	0.320	0.390	0.500	0.270	0.270	0.310	0.280	0.320
512^3	1.300	1.300	1.410	1.820	2.590	1.210	1.250	1.210	1.450	1.600
1024^3	10.450	10.650	11.370	13.150	18.890	10.340	10.340	10.190	10.290	11.200

Tabuľka 3.2: Časy výpočtu separovateľnej konvolúcie pre rôzne techniky a parametre. V stĺpcoch sú uvedené verzie bez balenie alebo s balenínm voxelov a veľkosti jadier. V riadkoch sú uvedené verzie využívajúce rôzny počet renderovacích oblastí a veľkosť spracovávaného objemu. Červenou farbou sú označené najrýchlejšie výpočty medzi rôznymi technikami a parametrami.

3.5.1.3 Detekcia tubulárnych štruktúr

V knižnici *f3dfilter* sme implementovali vyššie prezentované techniky na detekciu tubulárnych štruktúr. Implementácia je v jednoduchej presnosti s plávajúcou desatinnou čiarkou. Z výsledkov v tabuľke 3.3 je zrejmé, že najrýchlejšou technikou je klasický prístup bez balenia voxelov spracovávajúci jeden rez. Samotný výpočet vlastných čísel a matice druhých derivácií nie je zrejme príliš výpočtovo náročný a z toho dôvodu náklady na správu viacerých renderovacích oblastí a balenia voxelov nepriaznivo ovplyvnili výsledný výpočtový čas.

Tubulárne štruktúry

	128^3	256^3	512^3	1024^3
NonPack Mrt1	0,240	0,870	5,830	44,640
NonPack Mrt4	0,270	0,950	5,870	46,310
NonPack Mrt8	0,350	1,130	6,800	53,920
Pack Mrt1	0,270	1,010	7,790	64,930

Tabuľka 3.3: Časy detekcie tubulárnych štruktúr pre rôzne techniky (riadky) a veľkosti objemov (stĺpce). Červenou farbou sú označené najrýchlejšie výpočty.

3.5.2 Hardvérové verzie

V tejto časti sa zameriame na porovnanie aktuálne dostupných hardvérových verzií v knižnici *f3dfilter*. V tomto balíku je implementovaná základná CPU verzia, verzia využívajúca asemblerovú inštrukčnú sadu SSE procesora [Vaš05], [Cor99] a naša implementácia využívajúca GPU pomocou OpenGL rozhrania (ozn. *GL*) pre výpočet konvolúcie. Pre separovateľnú konvolúciu tento balík ponúka aj využitie GPU za pomoci CUDA technológie [Pod07]. Filter detekujúci tubulárne štruktúry využíva neiteračnú metódu na výpočet vlastných čísel implementovanú na CPU, SSE a našu implementáciu pomocou OpenGL rozhrania. V knižnici *f3dfilter* je implememementovaná aj iteračná metóda využívajúca knižnicu GSL (*GNU Scientific Library*) [GNU10]. Prezentované výsledky budú rozdelené do rôznych kategórií.

3.5.2.1 Výkon

V tejto časti si uvedieme výpočtový výkon jednotlivých filtrov. V tabuľke 3.4 sú uvedené výsledky pre neseparovateľnú konvolúciu, z ktorej je možné urobiť záver, že verzia využívajúca OpenGL je 8.1-krát rýchlejšia oproti SSE verzii a 275.1-krát rýchlejšia oproti CPU verzii v priemernom prípade. Vysoký výkon GPU je vidieť na výsledkoch, kde iba v dvoch prípadoch bola verzia SSE rýchlejšia.

Pre separovateľnú verziu sú uvedené výsledky v tabuľke 3.5. SSE verzia je tu rýchlejšia vo viacerých prípadoch, čo je spôsobené časom potrebným na inicializáciu GPU a na prenos dát. V priemernom prípade je však OpenGL verzia 3.1-krát rýchlejšia oproti SSE

Neseparovateľná konvolúcia

GL	3^3	5^3	9^3	15^3	31^3
128^3	0,150	0,180	0,260	0,490	2,440
256^3	0,360	0,470	1,070	2,780	17,750
512^3	1,790	2,740	7,550	21,130	142,640

SSE	3^3	5^3	9^3	15^3	31^3
128^3	0,060	0,150	0,490	2,500	16,740
256^3	0,330	0,770	4,840	20,670	147,040
512^3	2,320	7,050	39,420	166,850	1225,130

CPU	3^3	5^3	9^3	15^3	31^3
128^3	0,340	0,970	4,000	21,270	302,440
256^3	2,570	7,650	39,800	169,180	2153,580
512^3	20,800	68,510	319,570	1287,290	51115,141

Tabuľka 3.4: Časy výpočtov hardvérových verzií neseparovateľnej konvolúcie. V stĺpcoch sú uvedené veľkosti konvolučných jadier, v riadkoch veľkosti spracovávaných objemov a použité hardvérové verzie. Červenou farbou sú označené najrýchlejšie výpočty medzi rôznymi hardvérovými verziami.

verzii a 20.7-krát rýchlejšia oproti CPU verzii. CUDA verzia je najrýchlejšou verziou a je v priemere 1.1-krát rýchlejšia ako OpenGL verzia. Toto urýchlenie je pravdepodobne spôsobené lepším prispôsobením CUDA technológie na všeobecné výpočty a hlavne jednoduchšou správou pamäte, kde využívanie OpenGL rozhrania kladie rôzne obmedzenia pri programovaní.

V tabuľke 3.6 sú uvedené výsledky pre filter detekujúci tubulárne štruktúry. Z nameraných hodnôt vyplýva, že OpenGL verzia ja najrýchlejšia. Tento stav je spôsobený hlavne implementáciou v jednoduchej presnosti, pričom ostatné verzie využívaju dvojitú presnosť výpočtu. Preto je OpenGL verzia výpočtu v takom veľkom nepomere k ostatným verziám.

3.5.2.2 Presnosť

Oblasti pracujúce s objemovými dátami potrebujú vysokú presnosť spracovávaných dát, na čo bol zameraný test merajúci chybu výpočtu testovaných verzií. Na tento test sme použili dve sady dát, jednu reálnu a jednu umelo vygenerovanú s náhodnými hodnotami z intervalu $\langle 0, 1 \rangle$ s presnosťou na štyri desatinné miesta. Obdobným spôsobom bolo vygenerované aj konvolučné jadro s veľkosťami 3, 9 a 13 pixelov. Na testovacie dáta bol aplikovaný príslušný filter, kde v tabuľke 3.7 sú uvedené maximálne absolútne odchýlky

Separovateľná konvolúcia

GL	3	5	9	15	31
128^3	0,170	0,160	0,170	0,180	0,190
256^3	0,290	0,300	0,340	0,310	0,320
512^3	1,210	1,230	1,230	1,440	1,600
1024^3	9,080	10,540	10,400	10,270	11,310

SSE	3	5	9	15	31
128^3	0,030	0,040	0,060	0,070	0,150
256^3	0,140	0,170	0,310	0,650	1,100
512^3	1,010	1,670	2,820	5,170	8,960
1024^3	11,970	15,670	25,160	42,410	73,840

CPU	3	5	9	15	31
128^3	0,090	0,090	0,220	0,480	0,910
256^3	0,420	0,690	1,500	3,350	6,280
512^3	3,590	5,480	13,440	26,930	79,650
1024^3	42,460	52,780	108,110	271,270	641,450

CUDA	3	5	9	15	31
128^3	0,130	0,140	0,140	0,140	0,150
256^3	0,210	0,230	0,250	0,280	0,350
512^3	0,880	0,950	1,090	1,370	1,890
1024^3	6,370	6,980	8,110	10,320	14,820

Tabuľka 3.5: Časy výpočtov hardvérových verzií separovateľnej konvolúcie. V stĺpcoch sú uvedené veľkosti konvolučných jadier, v riadkoch veľkosti spracovávaných objemov a použité hardvérové verzie. Červenou farbou sú označené najrýchlejšie výpočty medzi rôznymi hardvérovými verziami.

Tubulárne štruktúry

	128^3	256^3	512^3	1024^3
GL	0,220	0,870	5,800	44,550
SSE	0,740	5,800	47,820	384,080
CPU	1,300	9,820	96,430	773,200
GSL	3,610	27,930	226,980	1832,360

Tabuľka 3.6: Časy výpočtov hardvérových verzií filtra detekujúceho tubulárne štruktúry. V stĺpcoch sú uvedené veľkosti spracovávaných objemov, v riadkoch rôzne hardvérové verzie. Červenou farbou sú označené najrýchlejšie výpočty.

medzi výsledkami z rôznych verzií a výsledkami z CPU verzie . Ako je možné vidieť, tak absolútna odchýlka v prípade separovateľnej verzie je nemenná pre rôzne veľkosti jadra a dostatočne malá (rádovo 10^{-7}). Pri neseparovateľnej konvolúcii chyba rastie s veľkosťou jadra pre všetky verzie. Táto chyba je spôsobená veľkým počtom aritmetických operácií počas výpočtu, kde sa daná nepresnosť prejavuje vo výraznejšej forme. Výsledky pre filter detekujúci tubulárne štruktúry sú uvedené tiež v tabuľke 3.7. Ako sme už spomínali, OpenGL verzia pracuje v jednoduchej presnosti s plávajúcou desatinnou čiarkou, čo je pre tento výpočet nedostatočné. Výsledné nepresnosti sú spôsobené hlavne výpočtom vlastných čísel, na ktorý sa používajú goniometrické funkcie [Ebe08].

Neseparovateľná konvolúcia

	GL	SSE
3	2,86E-06	2,86E-06
9	3,66E-04	3,81E-04
13	2,26E-03	1,89E-03

Separovateľná konvolúcia

	GL	SSE
3	1,43E-06	9,54E-07
9	2,93E-07	2,38E-07
13	2,38E-07	2,38E-07

Tubulárne štruktúry

GSL	GL	SSE
0,00E+00	1,38E-04	3,56E-04

Tabuľka 3.7: Meranie maximálnych odchýlok, ktoré vznikli počas výpočtu hardvérových verzií (stĺpce) pre rôzne veľkosti jadier (riadky) oproti CPU verzii.

3.5.2.3 Rôzne

V tejto časti sa budeme venovať testom, ktorými sme sa snažili odhaliť rôzne nešpecifické správanie hardvérových verzií. Jeden z testov bol zameraný na zistenie vplyvu veľkosti rezu na výsledný čas. Testovaný bol objem s veľkosťou 512MB, ktorého rez nadobúdal nasledujúce hodnoty - 128×128, 256×256, 512×512, 1024×1024 a menil sa len počet rezov. Pre CPU a SSE separovateľnú aj neseparovateľnú verziu v priemere platí, že menší rez vedie k rýchlejšiemu spracovaniu. Pre OpenGL separovateľnú verziu bol najvhodnejší rez o veľkosti 512×512 a neseparovateľnú rez o veľkosti 1024×1024. CUDA verzia si najlepšie poradila s rezom o veľkosti 1024×1024.

Ďalším testom bolo meranie, ktoré pracovalo taktiež s veľkosťou rezu, ale s menšími odchýlkami. Ako je známe, odporúčané veľkosti textúr používaných na GPU by mali byť mocninami čísla dva. Vyššie uvedené testy pracovali s takýmito textúrami. My sme však

otestovali objemy s rozmermi nerovnými mocninám čísla dva. Postupne sme zmenšovali vstupný rez (256×256) v smere X, alebo Y, alebo X a Y súčasne o 16 hodnôt. Vykonaný test nepreukázal žiadne výrazné odchýlky pre CPU a SSE verzie. Separovateľná OpenGL verzia bola 1.8-krát rýchlejšia pri zmenšení vstupného rezu v smere X, X a Y pre násobky štvorky. Spomalenie pri ostatných rozmeroch je spôsobené potrebou zarovnávať posledné texely v riadkoch pri technike balenia voxelov. Pri neseparovateľnej verzii sa vplyv zarovnania neobjavil, respektíve v porovnaní s celkovým výpočtom bol zanedbateľný. Pri CUDA verzii v smere X bol objem s rezom 256×256 a 240×256 1.4-krát rýchlejší ako ostatné rozmery. V smere Y, X a Y bol výpočet 2.2-krát pomalší v rozmeroch 256×255, 256×243, 256×241 oproti ostatným meraniam. Je to zrejme spôsobene špeciálnou implementáciou algoritmu alebo zarovnaním dát v pamäti.

3.5.2.4 Obmedzenia

Z použitých technológií a rozšírení vyplývajú aj isté obmedzenia na používané GPU. Použité GPU musia podporovať špecifikáciu OpenGL 2.0 a potrebné rozšírenia uvedené pri konkrétnych technikách. Výsledné algoritmy implementované v knižnici *f3dfilter* sme úspešne testovali za pomoci nástrojov obsiahnutých v knižnici *f3dclass* na grafickej karte NVIDIA 6600GT, ktorú považujeme za starší typ GPU.

Ďalším obmedzením závislým na GPU je podporovaný rozmer textúr. Pri výpočte konvolúcie v smere Z, respektíve vo všetkých troch smeroch X, Y a Z, kedy sa využíva 3D textúra, je rozmer vstupného rezu obmedzený maximálnym možným rozmerom 3D textúry. Pre súčasné GPU je to obvykle $2048 \times 2048 \times 2048$, čo je vo väčšine prípadov dostatočné. Staršie grafické karty majú maximálny rozmer 3D textúry obmedzený na $512 \times 512 \times 512$. Veľkosť 3D textúry je v smere Z definovaná veľkosťou konvolučného jadra v smere Z. Preto je rozmer 512 z praktického hľadiska postačujúci. Pri použití separovateľnej konvolúcie je možné vykonať konvolúciu v ľubovoľnom smere a pokiaľ nie je použitý smer Z, tak sa používajú len 2D textúry, ktorých maximálny rozmer pri súčasných GPU je 8192×8192. Staršie GPU podporovali rôzne veľké 2D textúry od 1024×1024, cez 2048×2048 až 4096×4096.

Pri použití dát s rezom s veľkými rozmermi a veľkým jadrom v smere Z je možné naraziť na pamäťový limit GPU. CUDA technológia umožňuje využiť maximálne toľko pamäte koľko je dostupnej na GPU. Ovládače pre OpenGL dovoľujú alokovať celkovo viac pamäťového miesta ako je fyzicky na GPU, ale dochádza pritom k opakovanému prenosu dát medzi GPU pamäťou a pamäťou počítača pod správou ovládačov. Preto je tento limit pri použití OpenGL verzie vyššie položený ako pri využití CUDA technológie. Možnou nevýhodou pri použití OpenGL algoritmov je skutočnosť, že ovládače alokujú rovnaké množstvo pamäte nielen na GPU, ale aj v hlavnej pamäti počítača, čo pri rozsiahlych dátach výrazne znižuje veľkosť pamäte pre iné výpočty (SSE verzie). Na tento pamäťový limit je možné naraziť hlavne pri spustení viacerých procesov (výpočtov) súčasne. Takéto viacprocesové testy budú uvedené v nasledujúcej časti.

3.5.3 Paralelné prostredie

Nakoľko GPU ťaží nielen zo svojho hrubého výkonu, ale aj z rozsiahlej paralelizácie, je potrebné spomenúť paralelizáciu aj pri CPU a SSE verziách podporovaných novými viacjadrovými procesormi. Viacjadrové procesory je možné využiť dvojakým spôsobom. Prvý spôsob je vnútorná paralelizácia výpočtu, tak ako je to využívané pri grafických kartách. Druhý spôsob je vonkajšia paralelizácia, kde sa spúšťajú viaceré procesy naraz. Tento spôsob je často využívaný v reálnych aplikáciách, kde sú na vstupných dátach vykonávané rôzne operácie, ktoré sú potrebné pre používateľa alebo vizualizáciu. Na takéto viacprocesové spracovanie objemových dát slúžia samostatné systémy, ktoré poskytujú aj grafické prostredie [MeV09, SCI09]. Rôzne vizualizačné aplikácie využívajú napríklad predfiltrované objemové dáta s rôzne veľkými jadrami na detekciu ciev [FNVV98]. Programy poskytované v knižnici *f3dclass* umožňujú jednoduchým spôsobom modelovať takúto aplikáciu s využitím viacerých dostupných prostriedkov v počítači ako GPU a viacjadrový procesor.

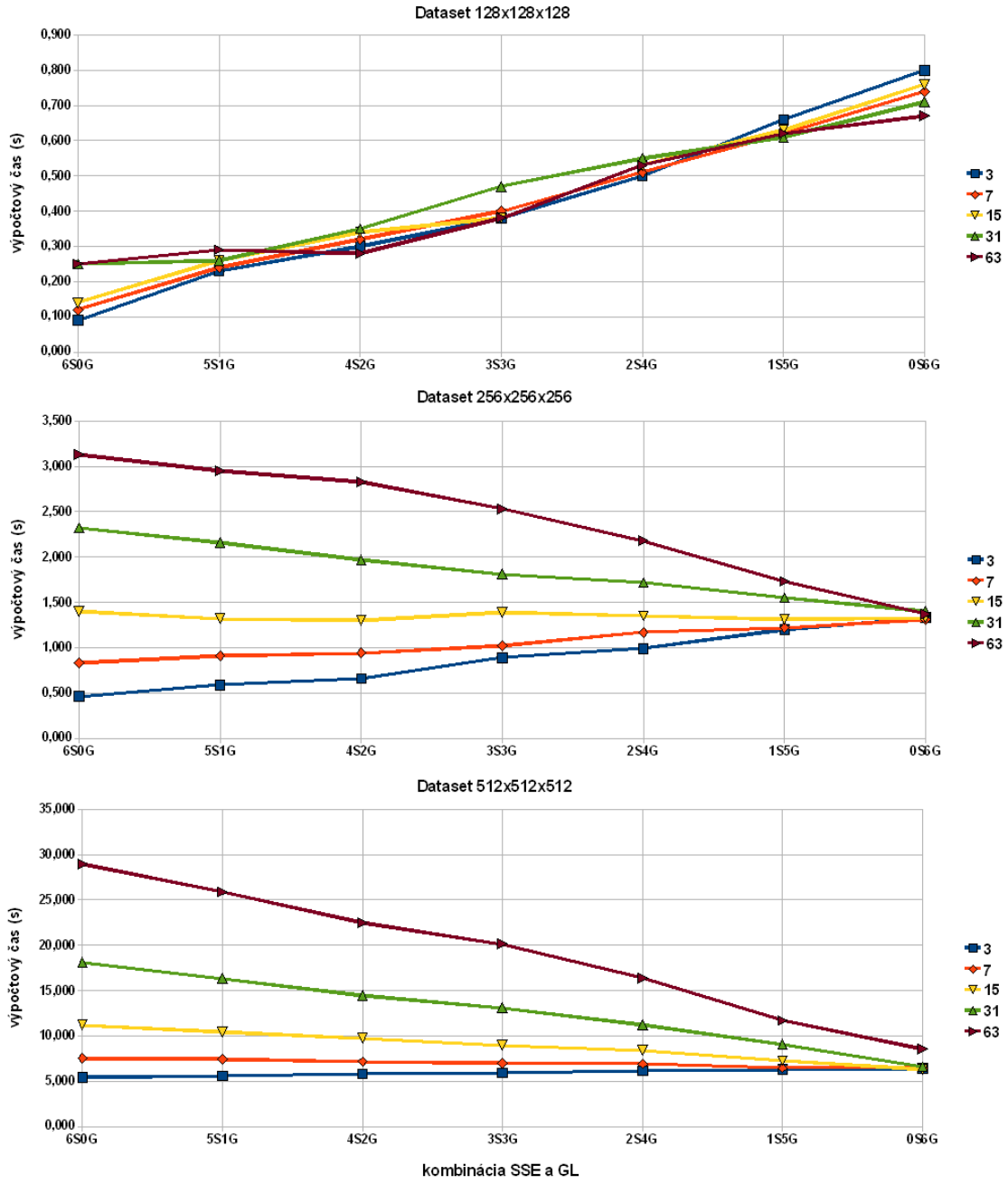
Na demonštráciu využitia takéhoto paralelného prostredia sme vytvorili jednoduchý skript, ktorý opakovane spustí 6 nezávislých procesov, pričom každý proces počíta separovateľnú konvolúciu s jadrom rovnakej veľkosti. Pri každom opakovaní testu sa mení počet procesov využívajúcich SSE v prospech OpenGL. Na obrázku 3.19 sú zobrazené grafy pre rôzne veľké vstupné dáta. Ako je možné vidieť, tak pre paralelné prostredie platia obdobné závery ako z predošlých testov. Čím je rozmer testovaných dát a jadra menší, tým je výhodnejšie používať viacej SSE verzií procesov, naopak pre vzrastajúce veľkosti dát a jadier je výhodnejšie používať GPU pre všetky procesy.

V ďalšom teste paralelného prostredia bol modifikovaný predošlý experiment, kde každý výpočet používal jadro s inou veľkosťou (použité veľkosti 3, 7, 15, 27, 45, 63). Pomocou tohto testu sme hľadali najvhodnejšiu kombináciu využitia SSE a GPU verzií programu. V tabuľke 3.8 sú uvedené tri najvýhodnejšie kombinácie s časmi pre testovaný objem. Z výsledkov je vidieť, že využitie GPU je najvýhodnejšie pre najnáročnejšie výpočty (najväčšie jadrá).

Kombinácia paralelných výpočtov separovateľnej konvolúcie

	128^3	256^3	512^3
1.	SSE+SSE+SSE+SSE+SSE+SSE 0.200	SSE+SSE+GL+GL+GL+GL 1.020	SSE+GL+GL+GL+GL+GL 6.280
2.	SSE+SSE+SSE+SSE+SSE+GL 0.250	SSE+SSE+SSE+GL+GL+GL 1.060	GL+GL+GL+GL+GL+GL 6.490
3.	SSE+SSE+GL+SSE+SSE+SSE 0.270	SSE+GL+SSE+GL+GL+GL 1.140	SSE+SSE+GL+GL+GL+GL 6.570

Tabuľka 3.8: Zobrazenie troch najvhodnejších kombinácií podľa času spracovania využívajúcich kombináciu SSE a OpenGL verzí programu pri spracovaní vstupných dát. V dolnej časti riadkov sú uvedené príslušné časy behu programov v sekundách pri použití nasledujúcich veľkostí jadier - 3,7,15,27,45,63.



Obrázok 3.19: Grafy zobrazujúce časový priebeh kombinácie SSE a OpenGL verzií algoritmu pracujúcich v šiestich paralelne bežiacich procesoch a rôznych veľkostiach jadier. Na osi Y sú znázornené časy výpočtu. Na osi X sú zvolené kombinácie, kde prvá číslica označuje počet procesov využívajúcich SSE verziu a druhá číslica OpenGL verziu algoritmu. Jednotlivé krivky reprezentujú veľkosti použitých konvolučných jadier pre všetky spustené procesy.

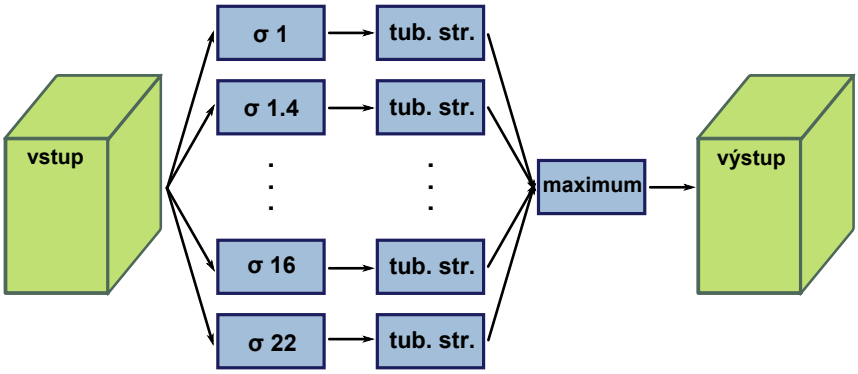
Vyššie uvedené testy ukázali, že je výhodné zapájať GPU do náročných výpočtov. Taktiež je vhodné kombinovať všetky dostupné hardvérové zdroje počítača. Tieto testy pracovali bez diskových operácii. Na záver sme použili test, ktorý vychádza z reálnej

aplikácie a je podobný predošlému testu. Počítala sa v ňom separovateľná konvolúcia na vstupných dátach (s diskovými operáciami) s desiatimi rôzne veľkými gaussovými jadrami s veľkosťami sigma 1, 1.4, ..., 22, čo viedlo k jadrám s veľkosťou 9, 11, ..., 177. Na výsledných dátach prebieha dodatočná detekcia tubulárnych štruktúr pomocou výpočtu matíc druhých derivácií a výpočtu vlastných čísel, na základe ktorých je vypočítaná hodnota zvýraznenia cievy [FNVV98]. Následne sú vytvorené výstupné dáta reprezentujúce maximá z jednotlivých výpočtov (procesov), ktoré sú uložené na disk alebo vizualizované (obrázok 3.21). Na obrázku 3.20 je znázornený diagram prebiehajúceho výpočtu. Pri testovaní sa menilo zastúpenie SSE a OpenGL verzií algoritmov na výpočte separovateľnej konvolúcie tak, že postupne boli SSE verzia nahrádzané OpenGL verziami od najväčších jadier k najmenším. Na detekciu tubulárnych štruktúr boli použité CPU, SSE a OpenGL verzie, pričom všetky spustené procesy využívali iba jednu z týchto verzií. Na testovanie sme použili CT snímku dolných končatín s rozmermi $512 \times 512 \times 1950$ a 12 bitovej fixnej presnosti, prekonvertovanú do 32 bitovej presnosti s pohyblivou desatinnou čiarkou, čo predstavovalo 1.9GB dát. Pri spracovaní desiatimi filtrami išlo teda o súčasné spracovanie 19GB dát. V tabuľke 3.9 sú uvedené výsledky z meraní, kde $RGT_{CPU}MW$ predstavuje originálnu verziu testu s využitím CPU algoritmu na detekciu tubulárnych štruktúr, $RGT_{SSE}MW$ so SSE verziou a $RGT_{GL}MW$ s OpenGL verziou výpočtu (názov testu je odvodený od prvých písmen jednotlivých operácií z angličtiny – Read, Gauss filtering, Tubular structure detection, Maximum, Write). Pri meraní sme použili aj test, ktorý nevykonáva detekciu tubulárnych štruktúr (ozn. $RGMW$).

	RGT128$_{CPU}$MW	**RGT$_{SSE}$MW**	**RGT$_{GL}$MW**	**RGMW**
10S0G	402,14	421,92	374,38	261,75
9S1G	356,86	373,71	325,69	214,93
8S2G	333,21	341,13	316,42	180,12
7S3G	315,16	317,72	298,02	155,46
6S4G	317,43	305,56	298,21	146,73
5S5G	309,59	295,90	306,29	131,84
4S6G	303,53	287,09	308,15	137,73
3S7G	310,85	283,54	330,14	145,68
2S8G	307,61	284,34	337,41	134,81
1S9G	316,43	282,25	354,40	144,27
0S10G	316,22	283,42	347,41	156,67

Tabuľka 3.9: Zobrazenie kombinácií SSE a OpenGL verzií programov na výpočet separovateľnej konvolúcie v reálnej aplikácii. Stĺpec RGTMW označuje výpočet reálnej aplikácie, stĺpec RGMW označuje výpočet bez detekcie tubulárnych štruktúr. Jednotlivé riadky predstavujú kombináciu (pomer) medzi SSE a OpenGL verziou výpočtu konvolúcie programu, kde prvá číslica predstavuje počet spustených SSE programov a druhá číslica počet OpenGL programov na štvorjadrovom procesore. Červenou farbou sú zvýraznené najrýchlejšie výpočty

Ako najvhodnejšia kombinácia sa ukázalo využitie jedného SSE a deväť OpenGL procesov pre výpočet separovateľnej konvolúcie a SSE verzia detekujúca tubulárne štruktúry, čo predstavovalo 1.49 násobné urýchlenie oproti verzii využívajúcej iba SSE algoritmy. Pri ostatných testoch je pomer využitia SSE a OpenGL verzií výpočtov separovateľnej konvolúcie rôzny, ale výsledné najrýchlejšie kombinácie sa výrazne od seba nelíšia. Pri teste detekujúcom tubulárne štruktúry len pomocou OpenGL nebol dosiahnutý kratší výpočtový čas, ako s využitím SSE verzie testu. Tento výsledok je pravdepodobne spôsobený veľkým počtom spustených procesov využívajúcich GPU a nedostatkom voľných pamäťových prostriedkov na GPU.



Obrázok 3.20: Zobrazenie výpočtu aplikácie využívanej na detekovanie (zobrazovanie) ciev v CT snímkach. Vstupný objem je filtrovaný pomocou desiatich rôzne veľkých Gaussových filtrov. Následne sú v dátach detekované tubulárne štruktúry. Výsledný objem je maximom jednotlivých výpočtov.

3.6 Zhrnutie

V tejto kapitole sme predstavili metódu prúdového spracovania objemových dát založenú na rezovom prístupe. Táto metóda umožňuje spracovávať dáta s veľkými rozmermi, ktoré presahujú pamäťové možnosti bežného počítača. Následne sme predstavili všeobecné techniky, ako využiť moderné GPU pri prúdovom spracovaní objemových dát. Najvýznamnejšia funkcionalitu GPU je asynchrónny prenos dát, ktorý práve pri prúdovom spracovaní umožní efektívnejšie využiť prostriedky GPU. Ďalšie významné techniky, ktoré je možné použiť na GPU sú spracovávanie viacerých rezov súčasne a balenie voxelov. Tieto techniky sme využili pri implementovaní algoritmov spracovávajúcich objemové dáta na GPU v prúdovom režime. Jednalo sa o dva filtre, všeobecný filter na výpočet separovateľnej a neseparovateľnej konvolúcie a filter detekujúci tubulárne štruktúry v objemových dátach. Implementované techniky urýchľovali výpočet oproti základnej implementácii, ktorá nevyužíva efektívne potenciál a funkcionalitu GPU. Tieto techniky sme taktiž porovnávali s rovnakými algoritmami využívajúcimi CPU, prípadne SSE, kde sme dosiahli výrazne urýchlenia, v závislosti od konkrétnych algoritmov. Tieto urýchlenia

Obrázok 3.21: Maximálna intenzitová projekcia s farebne zvýraznenými cievami na základe ich priemeru. Modrá - zelená - červená farba korešponduje s tenkými - strednými - hrubými štruktúrami.

sú výraznejšie pri náročnejších výpočtoch. Avšak musíme podotknúť, že využitie GPU pri menej zložitých problémoch, alebo menších objemových dátach nie je výhodnejšie, z dôvodu potreby inicializácie a prenosu dát na/z GPU. Implementované algoritmy sme využili aj v reálnej aplikácii, kde sa využívajú všetky dostupné zdroje počítača (GPU, SSE), čo sa ukázalo ako výhodné riešenie pri spracovaní objemových dát.

Z výsledkov je zrejmé, že moderné GPU sú vhodné nielen na vizualizáciu, na čo sú primárne určené, ale aj na rôzne náročné výpočty. Na výpočty, ktoré sa dajú vhodne paralelizovať je možné použiť GPU, kde môže dôjsť k výraznému urýchleniu oproti štandardným CPU riešeniam. Ako sa ukázalo, použitie GPU je výhodné hlavne pre spracovania veľkého množstva dát. Jednou nevýhodou využitia GPU pomocou OpenGL je potreba ovládať toto špecifické rozhranie a vedieť zapracovať jeho obmedzenia do návrhu algoritmu, čo nemusí byť vždy možné, respektíve efektívne.

Kapitola 4

Paralelné dátovo závislé triangulácie a rekonštrukcia obrazu

V tejto kapitole sa budeme venovať urýchleniu výpočtov optimálnych triangulácií pomocou GPU. V súčasnosti je generovanie optimálnych triangulácií výsostne doménou CPU. Generovanie je časovo náročný proces v závislosti na charaktere triangulácie.

Optimálne triangulácie sú často využívané vo vedeckých a technických aplikáciách rôznych odvetví. V tejto časti sa zameriame na získavanie lokálne optimálnych triangulácií iteračným spôsobom (optimalizáciou). Tie môžu byť použité v rôznych geometricky definovaných problémoch ako napríklad simulácii konečných prvkov alebo rekonštrukcii obrazu. Špeciálne sa zameriame na oblasť rekonštrukcie obrazu, najmä na zväčšovanie s dôrazom na zachovanie hrán. Takéto zväčšovanie je možné riešiť pomocou špeciálneho prípadu optimálnych sietí a to dátovo závislými trianguláciami (*data-dependet triangulations - DDT*) [DLR90]. Hlavnou výhodou tejto techniky je schopnosť prispôsobiť výslednú triangulovanú sieť vstupným dátam. Našim cieľom nie je vytváranie takejto triangulácie, ale iba jej optimalizácia. Na vstupe požadujeme ľubovoľne triangulovanú sieť a za pomoci špeciálnej cenovej funkcie a topologických operácií generujeme optimálnu sieť. Pre rôzne cenové funkcie sa dá dosiahnuť výsledná sieť s rôznymi vlastnosťami.

Rekonštrukcia obrazu je jedna z viacerých aplikačných oblastí, ktoré využívajú dátovo závislé triangulácie. Dôvodom výberu tejto aplikačnej oblasti bola možnosť vizuálneho zobrazenia výsledkov. Na vytváranie dátovo závislých sietí je použitý Lawsonov optimalizačný proces. Efektívna implementácia tejto techniky na GPU si vyžaduje jej paralelizáciu. Možnosti GPU nie sú priamo prispôsobené takémuto typu výpočtov a dátovým štruktúram. Hlavnou požiadavkou pri paralelnom návrhu bolo obísť tieto hardvérové obmedzenia a navrhnúť taký algoritmus, ktorý bude implementovateľný na GPU. Ďalšou časťou, ktorou sa zaoberá táto kapitola je aj skvalitnenie výslednej rekonštrukcie obrazu.

4.1 Dátovo závislé triangulácie

Digitálny obraz je dvojrozmerné pole pixelov, ktoré sú reprezentované dvojicou kartézskych súradníc (x, y) s farebnou hodnotou $f(x, y)$. Cieľom rekonštrukcie je vyčíslenie tejto funkcie v ľubovoľnom bode definičného oboru. Za uspokojivé výsledky považujeme také, ktoré sú bez výrazných artefaktov so zachovaním charakteristických znakov a vlastností. Požiadavkou na zväčšovanie obrázkov je najmä korektná rekonštrukcia vysokofrekvenčných oblastí, čo je hlavným kritériom kvality pre pozorovateľa.

Najbežnejšie používanými technikami sú konvolučné metódy využívané v spracovaní obrazu. Ideálnym rekonštrukčným filtrom je $sinc$ funkcia, ktorá však nie je vhodná na praktické použitie pre jej neohraničený definičný obor [GW06]. V praxi sa používajú rozličné ohraničené konvolučné filtre, ktoré často vnesú do výsledku artefakty rôzneho charakteru ako rozmazanie, zúbkovanie a iné. Výpočtová zložitosť týchto metód je závislá od veľkosti rekonštrukčného jadra. Vo všeobecnosti sú tieto techniky považované za menej výpočtovo náročné v porovnaní s inými technikami. Efektívnou implementáciou interpolácie s použitím viacerých konvolučných jadier na GPU sa zaoberal Hadwiger [HTHG01] a Bjorke [Bjo04].

Úplne odlišný prístup založený na geometrickej rekonštrukcii zaviedol Dyn [DLR90] s využitím dátovo závislých triangulácií. Tieto triangulácie sú prispôsobené vstupným hodnotám, čo vedie k po častiach lineárnej interpolácii. Na rozdiel od iných metód generujúcich triangulácie, dátovo závislé triangulácie prispôsobujú hrany vstupným dátam a organizujú trojuholníkové štruktúry do siete, ktorá zachováva črty zo vstupných dát. Kvalita výslednej triangulácie je definovaná pomocou špeciálnej *cenovej funkcie* v optimalizačnom procese. Yu a spol. [YBS01] využili DDT v rekonštrukcii obrazu. Zvyšovaním kvality rekonštrukcie sa zaoberal Tóth [Tó06]. Battiato a kolektív [BGM04] využili DDT na získavanie vektorových obrázkov z rastrových obrázkov.

Dátovo závislé triangulácie

Dátovo závislé triangulácie sú menej známe ako konvolučné techniky, preto si uvedieme základné definície a fakty.

Majme množinu rôznych (nie všetkých kolineárnych) bodov $\mathbf{V} = \{V_{i,j} = (x_i, y_j); i, j = 1, \ldots, n\}$ definovaných v E^2 (dvojrozmerný euklidovský priestor). Trianguláciou množiny $\mathbf{V}$ je množina trojuholníkov $T(\mathbf{V})$ s nasledujúcimi vlastnosťami:

- každý vrchol z ľubovoľného trojuholníka z $T(\mathbf{V})$ je z $\mathbf{V}$ a každý prvok z $\mathbf{V}$ je vrcholom minimálne jedného trojuholníka z $T(\mathbf{V})$,
- každá hrana z triangulácie obsahuje presne dva prvky z $\mathbf{V}$,
- zjednotenie všetkých trojuholníkov z $T(\mathbf{V})$ je konvexným obalom množiny $\mathbf{V}$, a
- prienikom dvoch ľubovoľných trojuholníkov z $T(\mathbf{V})$ je prázdna množina, alebo vrchol trojuholníkov, alebo ich spoločná hrana.

Dátovo závislá triangulácia vyššie spomenutej množiny **V** je triangulácia, ktorej topológia je závislá na funkcii $f(x, y)$ s

$$z_{i,j} = f(x_i, y_j); \;\; i, j = 1, \ldots, n,$$

kde $z_{i,j}$ sú dátové hodnoty, napríklad hodnoty jasu získané z farebných komponentov štandardnou konverziou.

Zameriame sa výlučne na hranovo založené dátovo závislé triangulácie, v ktorých sú oceňované iba hrany triangulácie. Existujú aj iné možnosti a to oceňovanie vrcholov triangulácie [Bro91]. Vzťah medzi hranami triangulácie a funkciou $f(x, y)$ môže byť ľubovolný, avšak my sa zameriame na nasledujúce podmienky. Každá vnútorná hrana je závislá na štyroch vrcholoch, ktoré tvoria štvoruholník (vytvorený z dvoch susedných trojuholníkov), kde daná hrana reprezentuje jeho diagonálu. Táto situácia je načrtnutá na obrázku 4.1, kde trojuholníky T_1 a T_2 definujú po častiach lineárnu plochu s hranou označenou ako e a vrcholmi $V_{i,j} = f(x_i, y_j), \; i, j \in \{0, 1\}$. Lineárne polynómy $P_1(x, y)$ a $P_2(x, y)$ definujú roviny, ktoré obsahujú trojuholníky T_1 a T_2:

$$\begin{aligned} P_1(x, y) &= a_1 x + b_1 y + c_1 \\ P_2(x, y) &= a_2 x + b_2 y + c_2 \end{aligned}$$

Tieto polynómy môžu byť použité na definovanie geometricky založených cenových funkcií. Sederbergova cenová funkcia (SCF) závislá na štyroch vrcholoch využíva projekciu normál trojuholníkov T_1 a T_2 do roviny xy nasledovne:

$$c^{SCF}(e) = \|\nabla P_1\| \cdot \|\nabla P_2\| - \nabla P_1 \cdot \nabla P_2,$$

kde

$$\nabla P_i = (a_i, b_i), \; i = \{1, 2\}$$

sú gradienty $P_i(x, y)$. Ich magnitúdy sú definované nasledovne:

$$\|\nabla P_i\| = (a_i^2 + b_i^2)^{1/2}, \; i = \{1, 2\}.$$

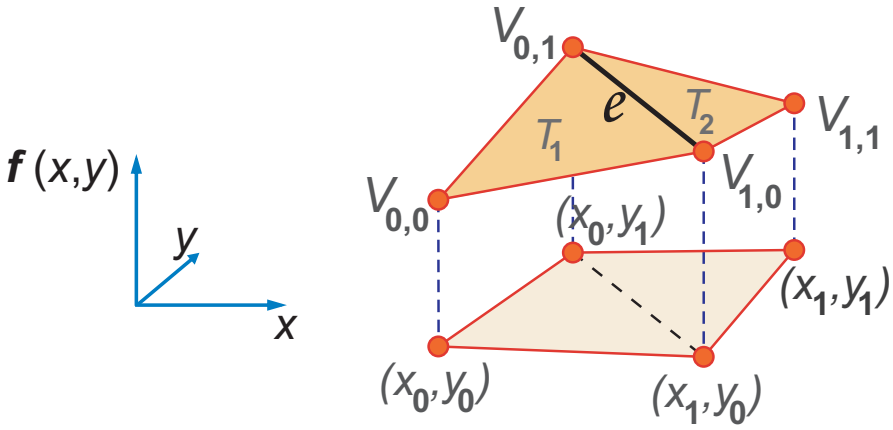
Existuje niekoľko ďalších prístupov, ktoré priraďujú cenu iným komponentom triangulácií (vrcholom, hranám, trojuholníkom) v závislosti na rôznych geometrických a negeometrických kritériách [AKTD00, Bro91, DLR90].

Generovanie dátovo závislých triangulácií vyžaduje proces optimalizácie siete. Cieľom je aproximácia triangulácie s minimálnu cenou (*minimal weight triangulation - MWT*), t.j. triangulácia množiny **V** s minimálnou cenou zo všetkých možných triangulácií:

$$c(MWT(\mathbf{V})) = \sum_{e \in MWT(\mathbf{V})} \|c(e)\| = \min\{c(T(\mathbf{V}))\}, \forall T(\mathbf{V}),$$

kde $c(T(\mathbf{V}))$ označuje cenu triangulácie. Mulzer a Rote [MR06] dokázali, že je to NP úplný problém. Pre rovinné dátovo závislé triangulácie je počet heuristických a aproximačných riešení [FNP96] problému hladania triangulácie s minimálnou cenou znížený

existenciou hranovej cenovej funkcie. Väčšina existujúcich optimalizačných prístupov je založená na bežnej topologickej operácii nazývanej *preklopenie hrany*. Každá vnútorná hrana je zjednotením dvoch susedných trojuholníkov, ktoré formujú štvoruholník. Ak je tento štvoruholník konvexný a nedegenerovaný, tak môže byť daná hrana nahradená druhou diagonálou štvoruholníka. Táto operácia zmeny diagonály sa nazýva preklopenie hrany. Hrana (diagonála) je lokálne optimálna ak je štvoruholník optimálne (s minimálnou cenou) triangulovaný. Ak je konkávny alebo degenerovaný, tak sa považuje takisto za lokálne optimálny. Uvádzame, že lokálna optimalita hrany je závislá od súčtu SCF cien piatich hrán (danej hrany a hrán štvoruholníka). Na určenie lokálnej optimality je preto potrebná informácia z ôsmich vrcholov (obrázok 4.1).



Obrázok 4.1: Zobrazenie geometrických závislostí v dvojrozmernej dátovo závislej triangulácii.

Najpopulárnejšou technikou na získavanie dátovo závislých triangulácií je Lawsonova optimalizácia [DLR90] (tiež nazývaná lokálna optimalizácia), ktorá využíva operáciu preklápania hrán. Táto technika spracováva hrany iteračne a vyhodnocuje ich lokálnu optimalitu. Pseudo kód je zobrazený na výpise 4.2. Je podobná Delaunayovej triangulácií [Aur91] pre rovinný prípad a ak cenová funkcia vyhovuje známej podmienke prázdneho kruhu [Aur91], tak sú totožné.

```
1  vytvor l'ubovoľnú trianguláciu
2  repeate
3    for ( každú hranu e z triangulácie )
4     if ( e nie je lokálne optimálna )
5       preklop(e)
6     endif
7    endfor
8  until nie je vypočítaná lokálna triangulácie
```

Výpis 4.2: Pseudo kód Lawsonovho optimalizačného procesu.

Vylepšenie tejto techniky (*the look-ahead*) prezentovali Yu a kolektí [YBS01]. Výpočtovo nenáročná modifikácia bola uvedená Su a Willisom [SW04]. Pomocou tohto rýchleho výpočtu triangulácie sa však dosiahli len nepatrne lepšie výsledky ako pomocou štandardnej bilineárnej alebo bikubickej interpolácie. Mimo vyššie uvedených deterministických metód boli uvedené aj stochastické metódy na hľadanie triangulácií s minimálnou cenou pomocou heuristík. Simulované žíhanie bolo použité Schumakerom [Sch93]. Rozšírením na kompresiu obrazu sa zaoberal Kreylos a kolektív [KH01]. Ďalším stochastickým procesom sú genetické optimalizácie využívané Kolingerovou [KF01] pri výpočte dátovo závislých triangulácií. Teoretické výsledky zaoberajúce sa počtom paralelne preklápaných hrán v trianguláciách boli publikované v [BCG+06].

Využitím GPU na aproximáciu Voronoiových diagramov v diskrétnom priestore sa zaoberali práce [HCK+99, FG06, RT06]. Práca Ronga a kolektívu [RTCS08] sa venuje generovaniu Delaunovej triangulácie z diskrétneho Voronoiovho diagramu. Bola tu použitá kombinácia GPU a CPU na výpočet Delaunovej triangulácie v spojitom priestore.

Práce, ktoré by sa zaoberali výpočtom dátovo závislých triangulácií na GPU nám nie sú známe, okrem našej práce [ČTS+10], ktorá bude popísaná v nasledujúcich častiach a rozšírená o dodatočné optimalizácie.

4.2 Paralelné dátovo závislé triangulácie

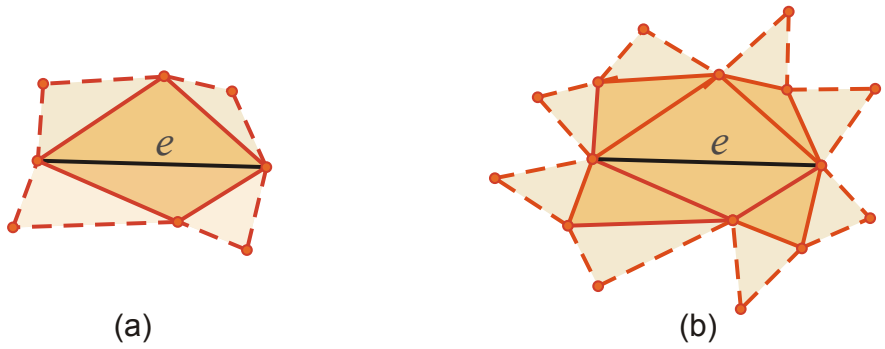
V tejto časti prezentujeme paralelnú verziu dátovo závislých triangulácií založenú na Lawsonovom optimalizačnom procese uvedenom v predošlej časti 4.1. Pri návrhu paralelného algoritmu vezmeme do úvahy možnosti a obmedzenia GPU. Technické detaily a implementácia sú rozobrané v časti 4.3.

Na začiatku potrebujeme definovať potrebnú terminológiu. *Susedmi* hrany *e* v triangulácii $T(\mathbf{V})$ označujeme množinu hrán z $T(\mathbf{V})$, ktoré tvoria spolu s hranou *e* trojuholník. Množina susedov hrany *e* je označená ako región stupňa 1 (1-*región*). Na základe tohto označenia definujeme n-*región* ($n > 1$) hrany *e* z $T(\mathbf{V})$ ako prienik nasledujúcich hrán:

- hrany z $(n-1)$-regiónu hrany *e* a
- susedov hrán z $(n-1)$-regiónu hrany *e*.

Lokálna optimalita hrany e v dátovo závislej triangulácii závisí od výberu cenovej funkcie. Preto definujeme región vplyvu (*region of influence* - *ROI*) hrany e ako množinu tých hrán, ktorých zmena (preklopenie) ovplyvní lokálnu optimalitu hrany e. Pre Sederbergovu cenovú funkciu uvedenú vyššie je región vplyvu stupňa 2 a obsahuje 12 hrán—obrázok 4.3(a).

V paralelnej verzii algoritmu by mali byť hrany spracované súčasne z čoho plynú isté obmedzenia. Predstavujeme iteračný algoritmus, ktorý pozostáva v každej iterácií z nasledujúcich troch krokov: *vytváranie kandidátov, akceptovanie a zamietanie kandidátov, preklápanie hrán*.



Obrázok 4.3: Región stupňa 2 (a) a región stupňa 3 (b) pre hranu *e*.

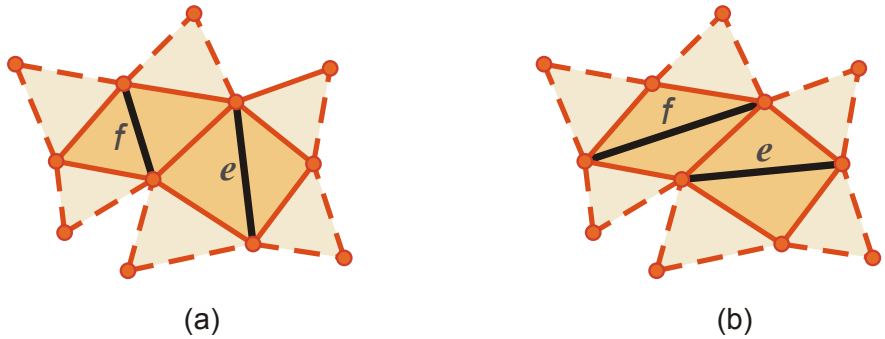
Vytváranie kandidátov

V prvom kroku jednotlivých iterácií je vyčíslená cenová funkcia pre každú hranu triangulácie a hrany sú rozdelené do dvoch množín. Prvá množina obsahuje lokálne neoptimálne *kandidátske* hrany, ktorých preklopenie má za následok zníženie ceny celej triangulácie. Ostatné hrany ostávajú nepreklopené v aktuálnej iterácii algoritmu. Vyčísľovanie cenovej funkcie sa navzájom neovplyvňuje a preto môže byť vykonávané paralelne na všetkých hranách.

Akceptovanie a zamietanie kandidátov

Lokálna optimalita hrany e je ovplyvnená všetkými topologickými zmenami (preklápanie hrán) v jej regióne vplyvu (ROI). V Lawsonovom optimalizačnom procese je preklápanie hrán vykonávané sekvenčne a preto nie je nutné brať tento fakt na zreteľ. V paralelnom spracovaní hrán je ale situácia odlišná.

Predstavme si situáciu s dvoma kandidátskymi hranami e a f, kde f je v ROI hrany e a naopak. Ak preklopíme jednu z týchto hrán, lokálna optimalita druhej hrany môže byť taktiež zmenená, nastane konflikt (obrázok 4.4). Na vyvarovanie sa takýchto konfliktov je potrebné vybrať z množiny kandidátov takú podmnožinu nezávislých hrán, ktorých ROI sa neprekrývajú takže môžu byť preklopené naraz.



Obrázok 4.4: Hrany *e* a *f* v konfliktnej situácií pred a po paralelnom preklopení (hrana *e* by mala byť neovplyvnená hranou *f* a naopak).

Inak povedané, množina kandidátov je rozdelená na dve disjunktné množiny

- *zamietnuté* hrany, ktoré nebudú preklopené a
- *akceptované* hrany, ktoré môžu byť preklopené paralelne.

Klasifikácia hrán je založená na porovnaní identifikačných čísel hrany (ID). Každá hrana má svoje jedinečné ID, ktoré je jej pridelené na začiatku algoritmu. Ak je hrana preklopená, uchováva si toto ID aj po preklopení.

Klasifikačný proces je spustený na všetkých hranách z kandidátskej množiny. Pre každú kandidátsku hranu e je potrebné otestovať všetky hrany z jej ROI. Ak existuje aspoň jedna akceptovaná hrana v jej ROI, hrana e je zamietnutá. Ak v jej ROI nie sú žiadne akceptované hrany a ID testovanej hrany má najmenšiu hodnotu z kandidátskych hrán v jej ROI, tak je hrana pridaná do množiny akceptovaných a odstránená z množiny kandidátov. Ak ID hrany e nie je najmenšie, tak táto hrana ostáva kandidátom a bude spracovaná v nasledujúcej iterácii tejto časti algoritmu.

Tento proces je iteratívny a pokračuje pokiaľ existujú hrany v množine kandidátov. Táto množina je konečná a v každej iterácií tohto procesu je možné nájsť hranu s minimálnym ID. Je zrejmé, že aspoň jedna kandidátska hrana je akceptovaná alebo zamietnutá v každej iterácii a preto sa počet hrán v množine kandidátov zmenšuje prinajmenšom o jednu hranu v každej iterácii. Preto je tento proces vždy konečný.

Opísaný proces vyhovuje možnostiam a obmedzeniam GPU ako bude uvedené v nasledujúcej časti. V prípade implementácie na CPU je možné využiť pri návrhu algoritmu sofistikovanejšie techniky.

Preklápanie hrán

V poslednom kroku algoritmu sú paralelne preklopené všetky hrany z množiny akceptovaných hrán vytvorenej v predchádzajúcom kroku algoritmu. Taktiež sú aktualizované príslušné dátové štruktúry spracovávanej hrany a hrán z jej ROI.

Celý iteračný proces je vykonávaný pokiaľ nie je dosiahnutá lokálne optimálna triangulácia. Pseudo kód algoritmu je uvedený na výpise 4.5.

4.3 Implementácia s využitím GPU

V predchádzajúcej časti sme navrhli paralelný algoritmus pre dátovo závislé triangulácie. Bol navrhnutý na vykonávanie vo fragmentovej jednotke tak ako väčšina GPGPU výpočtov. Vďaka tomu je možné vykonávať tento algoritmus aj na starších GPU.

Jedným z hlavných problémov implementácie bolo namapovanie algoritmu triangulácie na fragmenty. Najflexibilnejšou vstupnou dátovou štruktúrou pre fragmentový program je textúra, ktorá môže obsahovať najviac štyri hodnoty v jej elementoch – *texeloch*. Každý texel je reprezentovaný jedným farebným RGB kanálom a kanálom s priehľadnosťou A. GPU môže čítať z viacerých textúr a pristupovať k ľubovoľnému počtu texelov počas fragmentových operácií. Hlavné obmedzenie je v zapisovaní výstupu, kde iba

```
 1  repeat
 2    AcceptedSet.clear()
 3    CandidateSet.clear()
      // – Krok 1 –
 4    for ( každú hranu e z triangulácie )
 5      if ( e nie je lokálne optimálna )
 6        CandidateSet.add(e)
 7      endif
 8    endfor
      // – Krok 2 –
 9    while ( existuje hrana e z CandidateSet )
10      if ( existuje akceptovaná hrana f v ROI hrany e )
11        CandidateSet.remove(e)     // zamietni e
12      else
13        if ( e.ID < min ( ID všetkých kandidátskych hrán z ROI hrany e ) )
14          AcceptedSet.add(e)     // akceptuj e
15          CandidateSet.remove(e)
16        else
17              // e ostáva kandidátom, nezmenená
18        endif
19      endif
20    endwhile
      // – Krok 3 –
21    for ( každú hranu e )
22      if ( e z AcceptedSet )
23        flip(e)
24        zaktualizuj dátové štruktúry(e)
25      else
26        if ( existuje akceptovaná hrana v 1-regióne hrany e ) // susedná hrana
27          zaktualizuj dátové štruktúry(e)
28        endif
29      endif
30    endfor
31  until lokálne optimálna triangulácia nie je vypočítaná
```

Výpis 4.5: Pseudokód paralelnej verzie Lawsonovho optimalizačného algoritmu, vyhovujúceho možnostiam GPU.

konštantný počet hodnôt môže byť zapísaný do výstupnej pamäti (*framebuffer*) a počet týchto výstupných pamätí je taktiež limitovaný. Navyše, zapisovanie je obmedzené na predom definovanú fixnú pozíciu (*offset*) a táto pozícia je rovnaká pre všetky výstupné pamäte.

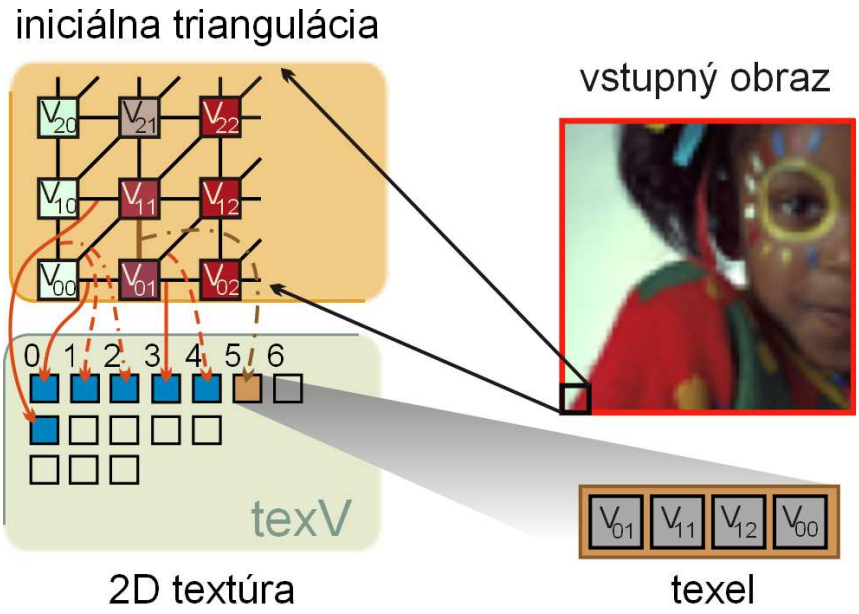
Návrh dátových štruktúr

Pri implementácii algoritmu sme preskúmali dve možnosti reprezentácie triangulácií na GPU. Prvá možnosť, označená ako *Point-centric approach* v [GW05], reprezentovala fragmentom jeden vrchol. Tento prístup vedie k problému s počtom vstupných a výstupných hrán pre každý vrchol keďže sa tento počet mení v priebehu optimalizácie. Takéto dynamické štruktúry sa vo všeobecnosti na GPU ťažko efektívne udržiavajú. Z tohto dôvodu bol vybratý prístup, kde fragment reprezentuje hranu triangulácie označený, ako *Edge-centric approach* v [GW05]. Tento prístup opísaný v [GW05] sa zaoberá podobným problémom reprezentácie siete, ale riešenie je rozdielne od toho, ktoré sme použili v našom algoritme. Každá hrana v triangulácii je definovaná jej dvoma koncovými bodmi – vrcholmi (označenými ako *hlavné vrcholy*). Tento *vrcholový dátový element* sme rozšírili o ďalšie dva vrcholy z 1-regiónu hrany (označené ako *priľahlé vrcholy*). Hlavné a priľahlé vrcholy sú tie vrcholy trojuholníkov, ktoré zdieľajú vybranú hranu ako spoločnú.

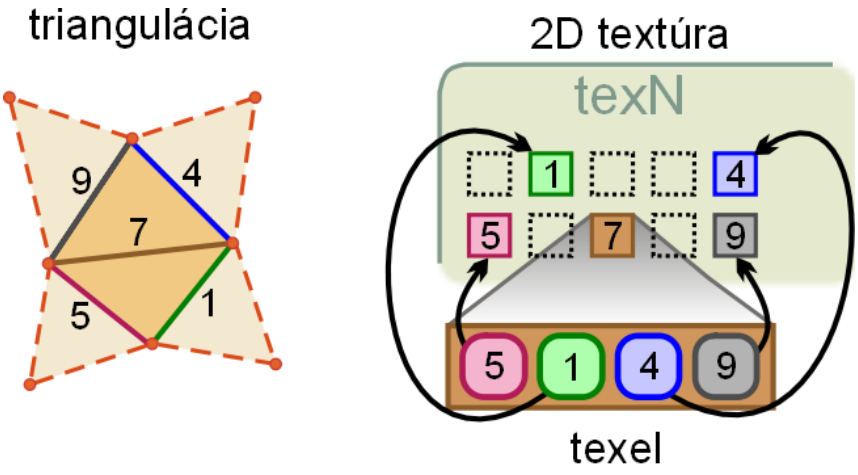
Počet hrán a vrcholov v triangulácii je počas optimalizácie konštantný. Pre každú hranu jej *vrcholový dátový element* obsahuje štyri jednoznačné identifikátory ID vrcholov (hlavné a priľahlé vrcholy hrany). Vrcholové dátové elementy sú uložené v textúre označenej *texV* (obrázok 4.6). ID vrcholov sú použité na výpočet pozícií vrcholov vo vstupnom obrázku na získanie hodnôt jasu pre vyčíslenie cenovej funkcie. Pre toto vyčíslenie je taktiež potrebné mať informáciu o susedných hranách. *Susedný dátový element* pre hranu *e* tak pozostáva zo štyroch ID susedných hrán hrany *e*. Tieto susedné dátové elementy sú uložené v textúre označenej ako *texN* (obrázok 4.7). Vrcholy iniciálnej triangulácie sa nachádzajú v strede pixelov, pričom sú medzi sebou pospájané tak, ako je naznačené na obrázku 4.6 (hore vľavo).

Textúry *texV* a *texN* obsahujú potrebné topologické informácie o triangulácii. Dodatočné textúry sú použité na udržanie informácií o kandidátskych hranách pre druhý krok algoritmu (akceptovanie a zmietanie kandidátov). V tejto časti je potrebné čítať a zapisovať z/do textúry v každej iterácii. Kvôli obmedzeniam GPU je potrebné mať dve takéto textúry označené ako $texC_1$, $texC_2$. Tieto textúry sú navzájom zamieňané ako zdrojové a cieľové (čítanie, zápis) v každej iterácii. V každom texeli týchto textúr je uložená informácia, či je príslušná hrana kandidátom, akceptovaná alebo zamietnutá spolu s ID tejto hrany. Dáta zo vstupného obrazu sú uložené v ďalšej textúre v šedotónovom formáte. Táto textúra je potrebná na výpočet cien hrán. Pre všeobecný výpočet triangulácie je potrebná ďalšia textúra na uloženie reálnych súradníc vrcholov v rovine. Pri spracovaní obrazu vrcholy formujú rovnomernú mriežku a tak táto textúra nie je potrebná, pretože súradnice vrcholov sa dajú vypočítať z ich *ID*.

V nasledujúcich častiach sú uvedené implementačné detaily všetkých troch krokov algoritmu ako boli uvedené v predošlej sekcii 4.2.



Obrázok 4.6: Časť iniciálnej triangulácie (hore vľavo). Vytváranie textúry texV z počiatočnej triangulácie (vľavo). Jeden texel textúry texV (vpravo).



Obrázok 4.7: Reprezentácia susedných hrán v textúre texN.

Vytváranie kandidátov
V tomto kroku algoritmu sú vybraní vhodní kandidáti na preklopenie. Vo fragmentovom programe je vyčíslená cenová funkcia pristupovaním k vrcholom (z ROI) pomocou ich ID uložených v *texN* a susedným hranám pomocou ID uložených v textúre *texV*. Toto je vykonané súčasne pre každý element textúry a tým pádom pre každú hranu triangulácie.

Podľa výsledku cenovej funkcie sa hrana stane kandidátom na preklápanie, alebo je lokálne optimálna a zostane nezmenená. Ak cenová funkcia označí hranu ako kandidátsku tento stav je uložený (zapísaný) do textúry $texC_1$. Ak testovaná hrana nie je kandidátom, tak príslušný fragment nie je zapísaný do textúry $texC_1$ (discarded) a príslušný fragment ostane nezmenený. V tomto procese je taktiež získaný počet kandidátov. Je na to použitá funkcionalita GPU označovaná *occlusion query*, ktorá vráti počet zapísaných fragmentov. Získaný počet fragmentov je použitý v nasledujúcom kroku (akceptovanie a zamietanie kandidátov). Ak je počet kandidátov nulový, je vypočítaná triangulácia lokálne optimálna a algoritmus je ukončený.

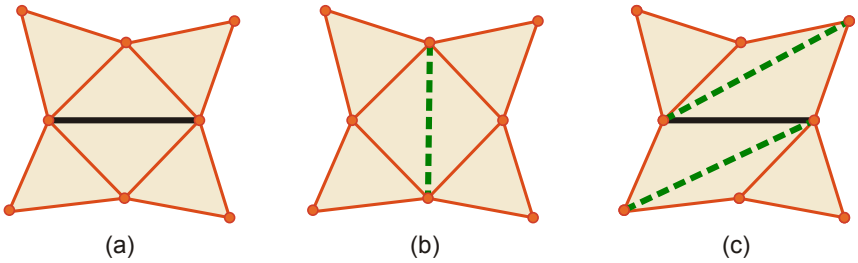
Vytváranie kandidátov
V druhom kroku algoritmu sú kandidátske hrany akceptované alebo zamietnuté. Je to iteratívny proces, kde textúry $texC_1$ a $texC_2$ sú vzájomne zamieňané na čítanie a zápis.

Na začiatku je obsah textúry $texC_1$ nakopírovaný do $texC_2$ na udržiavanie korektnej informácie o kandidátoch pre fragmenty, ktoré nebudú zapísané. Na spracovanie hrany *e* je potrebná informácia (ID, akceptácia) získaná vo fragmentovom programe z nespracovaných kandidátskych hrán z jej ROI. Pod spracovanou hranou rozumieme hranu, ktorá bola označená ako akceptovaná alebo zamietnutá. Ak je aspoň jedna hrana z ROI hrany *e* akceptovaná, testovaná hrana je zamietnutá a táto informácia je zapísaná do textúry. Ak v ROI spracovávanej hrany nie je žiadna akceptovaná hrana, sú porovnávané ID týchto (nespracovaných) hrán. Ak má testovaná hrana minimálne ID, tak je akceptovaná a táto informácia je zapísaná na výstup. Ak existuje nespracovaná hrana s nižším ID v ROI testovanej hrany, fragment nie je zapísaný (discarded).

Vo vyššie popísanom procese sú niektorí kandidáti označení ako akceptovaní alebo zamietnutí, ale nespracované hrany môžu stále existovať v množine kandidátov. Preto je tento proces opakovaný pokiaľ nie sú spracované všetky hrany. Technika occlusion query je použitá na zistenie počtu spracovaných kandidátov. Výsledok je uložený v závislosti na počte iterácií buď v textúre $texC_1$ alebo v $texC_2$.

Preklápanie hrán
Operácia preklápania hrany je prevedená výmenou priľahlých vrcholov za hlavné. Taktiež je potrebné zaktualizovať informácie o susedných hranách. Tieto zmeny musia byť zapísané do textúr *texV* a *texN* súčasne. Toto je zabezpečené technikou MRT (zapisovanie do viacerých renderovacích oblastí súčasne - uvedené v predošlej kapitole 3). Pre tento proces je potrebné čítať informácie z oboch textúr *texV* a *texN*. Preto sú vytvorené kópie týchto textúr a sú použité ako vstup do fragmentového programu. Ďalšou vstupnou textúrou je jedna z textúr $texC_1$ alebo $texC_2$, kde sú uložené informácie o akceptovaných kandidátoch.

Kvôli obmedzeniam GPU nie je možné zapisovať na ľubovoľnú pozíciu v textúre, ale iba na fixne definovanú pozíciu. Počas preklápania hrán môžu nastať tri situácie pri spracovávaní hrany. Prvá je najjednoduchšia, keď ani testovaná hrana ani hrana z jej okolia nie sú preklápané. V tomto prípade sú nezmenené hodnoty uložené na výstup (obrázok 4.8(a)). Druhý prípad nastane, keď je preklápaná spracovávaná hrana (obrázok 4.8(b)). V tomto prípade sú hlavné vrcholy zamenené s priľahlými v textúre *texV*. Prirodzene je potrebné nastaviť správne hodnoty aj do textúry *texN*, aby reflektovali zmeny susedov preklápanej hrany. Posledná situácia nastane, ak jedna hrana z okolia spracovávanej hrany e je preklopená (obrázok 4.8(c)). Pri preklopení susednej hrany nastávajú zmeny aj v okolí spracovávanej hrany e. Na ošetrenie tejto situácie je potrebné prehľadávať okolie spracovávanej hrany e. Ak nastane takáto zmena, tak sú získané a uložené na výstup ID nových susedov.



Obrázok 4.8: Rôzne možnosti pri preklápaní hrany (testovaná hrana je označená hrubou plnou čiarou, preklopená hrana je označená bodkovanou čiarou): bez preklopenia (a), preklopenie testovanej hrany (b), zmena okolia (c).

Vizualizácia

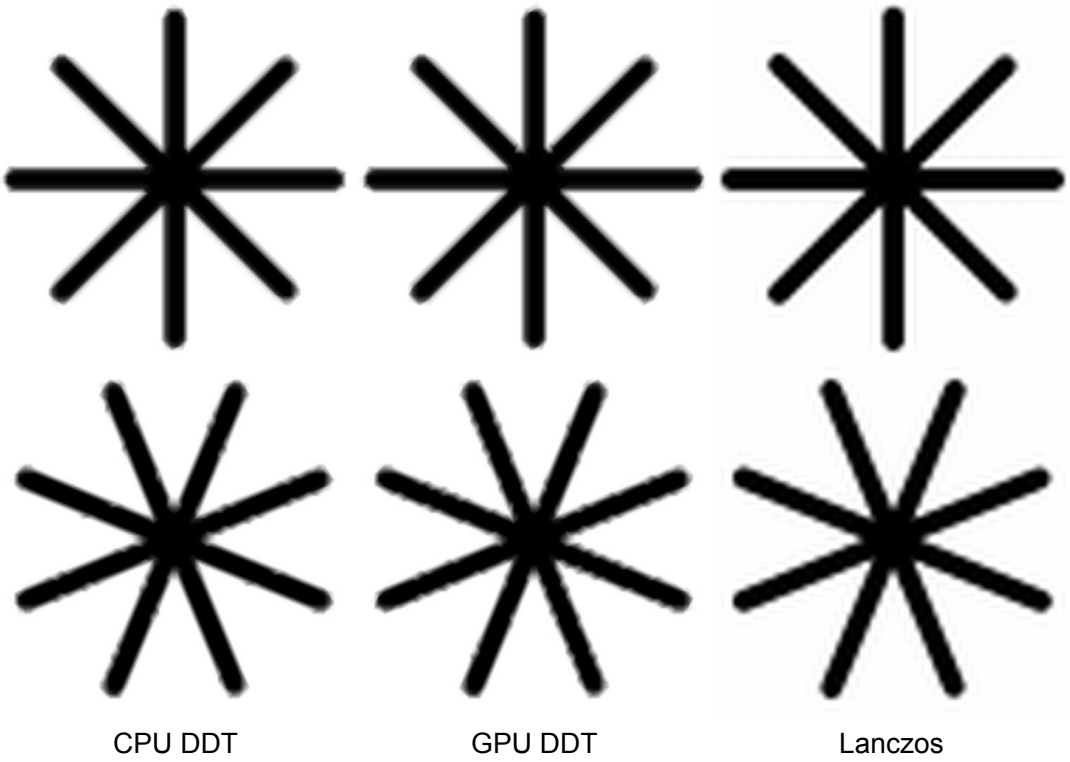
Akonáhle je dátovo závislá triangulácia vypočítaná, je možné stiahnuť dátové štruktúry (textúry *texN*, *texV*) na CPU a uložiť trianguláciu v príslušnom formáte. Vizualizácia triangulácie je v tomto prípade priamočiara, využijúc štandardné techniky na zobrazovanie trojuholníkov. Existuje však aj možnosť využiť jednotku spracovávajúcu geometriu na zobrazenie triangulácie priamo bez nutnosti sťahovania dátových štruktúr na CPU. Výsledná triangulácia z tejto jednotky môže byť ďalej spracovávaná vo fragmentovej jednotke alebo zapísaná do vrcholového zásobníku (*Vertex buffer object*) na dodatočné spracovanie a vizualizáciu. Navyše, triangulácia môže byť spracovaná využitím techniky *transform feedback* [Khr10]. V našej implementácii sme použili a testovali obe vizualizačné techniky (stiahnutím na CPU a pomocou geometrickej jednotky).

4.4 Vylepšenia

V predchádzajúcej časti sme uviedli nový paralelný optimalizačný algoritmus vykonávaný na GPU. Prvotné testy tohto prístupu ukázali isté obmedzenia v porovnaní s neparalelnou CPU verziou algoritmu (bola použitá CPU verzia algoritmu od Dr.Tótha [Tó06]).

Po prvé, sieť optimalizovaná pomocou GPU mala zvyčajne vyššiu cenu než sieť optimalizovaná pomocou CPU algoritmu. Z teoretického hľadiska je toto možné, pretože k iniciálnej triangulácii môže existovať viacero navzájom rozdielnych, lokálne optimálnych konfigurácií, ktoré majú prirodzene rozdielnu cenu. Avšak cieľom nie je len dosiahnuť lokálne optimum, ale taktiež aproximovať globálne optimum. Po druhé, v rekonštruovaných obrázkoch sa objavili rušivé artefakty, ktoré sa často nachádzali v hranových oblastiach s istými vlastnosťami (obrázok 4.9).

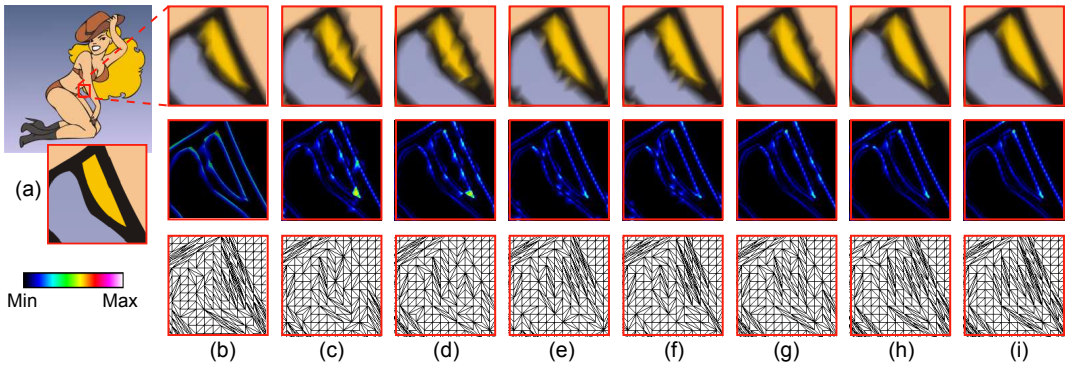
Na potlačenie týchto artefaktov boli navrhnuté metódy pracujúce v obrazovom priestore. Taktiež boli navrhnuté modifikácie algoritmu za účelom potlačenia artefaktov a zníženia ceny triangulácie pracujúce vo výpočtovom priestore.



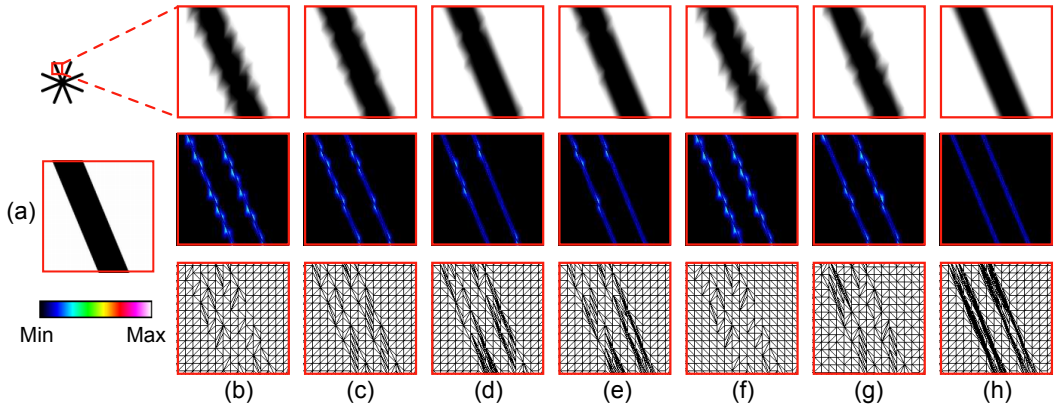
Obrázok 4.9: Rekonštrukcia obrázkov (zväčšenie o 800%) rôznymi technikami. V dolnom riadku je použitý obrázok, ktorého rekonštrukcia generuje výrazné artefakty v hranových oblastiach.

4.4.1 Vylepšenia vo výpočtovom priestore

Modifikácie algoritmu uvedené nižšie sme navrhli za účelom potlačenia viditeľných artefaktov a zníženia ceny triangulácie. Vplyv modifikácií na výsledky sú prezentované na obrázkoch 4.10 a 4.11. Stredný riadok v týchto obrázkoch zobrazuje diferenčný obrázok (v porovnaní s originálnym obrázkom) vypočítaný v perceptuálne lineárnom farebnom priestore L*,u*,v*.



Obrázok 4.10: Záber na 1200% zväčšenie rasterizovaného vektorového obrázka. Zvýraznená oblasť zachytáva časť s najväčším výskytom artefaktov. Originálny obrázok (a), CPU DDT algoritmus (b), Basic (c), MaxGain (d), ExpRoi (e), ExpRoi3 (f), ExpRoiMaxGain (g), MeshSobMax (h), ExpRoiMaxGainMeshSobMax (i) modifikácie.



Obrázok 4.11: Rekonštruovaný obrázok pri 1000% zväčšení s využitím rôznych modifikácií: originálny obrázok (a), Basic (b), MeshCanny (c), MeshSobAvg (d), MeshSobMax (e), MeshLO (f), MeshRand (g), ExpRoiMaxGainMeshSobMax (h).

Expanzia ROI

Analyzovali sme vyššie spomenuté rušivé artefakty zobrazené na obrázku 4.9. Spozorovali sme, že tieto artefakty sú spôsobené nevhodne orientovanými hranami v triangulácii, ktoré boli uniformne distribuované pozdĺž vysokofrekvenčných oblastí vo vzdialenosti približne rovnakej ako je veľkosť ROI oblasti. Preto sme pri procese výberu kandidátov (druhý krok algoritmu, riadky 9-20 na výpise 4.5) oblasť ROI rozšírili. V základnom prístupe je tento región veľkosti 2, čím pokrýva dvanásť hrán. Tento región sme rozšírili o dodatočné hrany na región stupňa 3. Táto situácia je znázornená na obrázku 4.3(b), kde nové pridané hrany sú označené prerušovanou čiarou. Táto modifikácia čiastočne potlačila artefakty a výsledná cena je nižšia. V nasledujúcom texte bude táto

modifikácia označená ako *ExpRoi*. Obrázok 4.10(e) zobrazuje vizuálny prínos v porovnaní so základným algoritmom označeným ako *Basic*.

Nevýhodou rozšírenej oblasti je zvýšená výpočtová náročnosť, čo vedie k dlhšiemu výpočtu v príslušnom fragmentovom programe. Pri skúmaní základného algoritmu sme zistili, že tieto artefakty (nevhodne preklopené hrany) zvyčajne vzniknú v prvých dvoch, troch iteráciách výpočtu. Náročnejší výpočet rozšírenej oblasti je preto potrebný len pri prvých N iteráciach, kde N môže byť nastavené používateľom. Po týchto N iteráciách je použitý prvotný prístup s regiónom stupňa 2.

Táto modifikácia bude označovaná ako *ExpRoiN*. Ako je znázornené na obrázku 4.10(f), táto modifikácia vedie k podobným výsledkom ako *ExpRoi* modifikácia (obrázok 4.10(e)).

Maximalizácia prírastku

Hlavným cieľom tejto modifikácie je zníženie ceny triangulácie. Na rozdiel od základného algoritmu, kde sú akceptované hrany vyberané výhradne na základe IDhrany, v tejto modifikácií je hodnotená veľkosť príspevku preklopenia hrany k minimalizácii ceny triangulácie. Teraz nie je lokálna cena hrany len vyčíslená, ale je uložená a použitá v kroku akceptovania a zamietania kandidátov algoritmu. Počas iterácie nad množinou kandidátov je vybraná hrana s najväčším prírastkom (rozdiel medzi cenou pred preklopením a cenou po preklopení hrany) z ROI aktuálne spracovávanej hrany. Ak niektoré z testovaných hrán v ROI majú rovnaký maximálny prírastok, tak sú použité ich ID pri rozhodovaní, podobne ako v základnom algoritme.

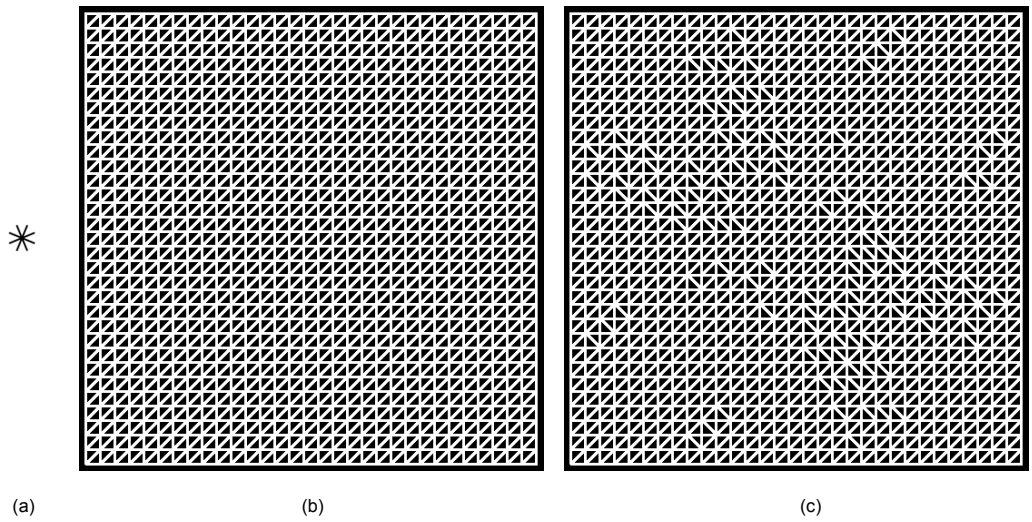
Táto modifikácia zvýši výpočtový čas algoritmu, ale výsledná cena triangulácie je zvyčajne nižšia a tak lepšie aproximuje globálne optimum. Zároveň táto verzia konverguje rýchlejšie k minimu. Preto nevýhodou tohto zlepšenia je možnosť dosiahnutia horšieho lokálneho optima v porovnaní so situáciou, kde znižovanie ceny je pomalšie. Podobné správanie sa dá pozorovať v minimalizovaní pomocou simulovaného žíhania so zle nastavenými kontrolnými parametrami [Sch93]. V [HD06] je spomínaný podobný postup pre neparalelnú verziu Lawsonovho optimalizačného procesu.

Táto modifikácia bude označovaná ako *MaxGain*. Obrázok 4.10(d) ukazuje výsledky v porovnaní so základným GPU algoritmom. Pri tejto modifikácie je zvyčajne cena triangulácie nižšia ako pri základnom algoritme.

Predspracovanie siete

Modifikácia *ExpRoi* sa snaží odstrániť nevhodne orientované hrany vo vysokofrekvenčných oblastiach priamo vo výpočte. Modifikácie predstavené v tejto časti budú upravovať iniciálnu trianguláciu za rovnakým účelom. Cieľom týchto metód je prispôsobenie iniciálnej triangulácie vstupnému obrázku. Jedná sa hlavne o diagonálne hrany iniciálnej triangulácie, ktoré môžu byť umiestnené dvoma spôsobmi. Štandardne majú tieto diagonálne hrany smer z ľavého dolného rohu do pravého horného rohu štvoruholníka (obrázok 4.12(b)). Predpokladáme, že vo vysokofrekvenčných oblastiach umiestnených približne v kolmom smere na iniciálne hranové diagonály môžu vznikať artefakty z nevhodného umiestnenia týchto diagonálnych hrán. Preto sú vo vysokofrekvenčných oblastiach diagonálne hrany iniciálnej triangulácie upravené podľa smeru tejto oblasti

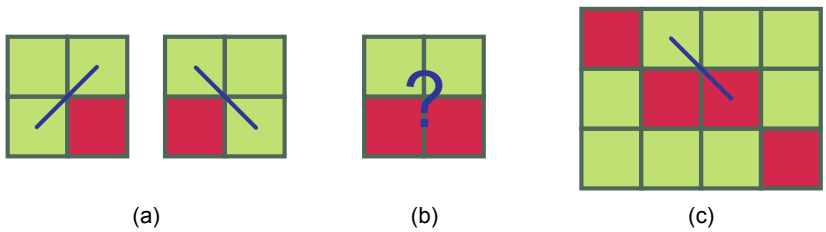
(obrázok 4.12(c)). Na zisťovanie významných vysokofrekvenčných oblastí boli použité rôzne hranové detektory. Na ich základe boli tieto hrany upravené.



Obrázok 4.12: Zobrazenie štandardnej inicializačnej triangulácie (b) a upravenej verzie (c) podľa vysokofrekvenčných oblastí vstupného obrázka (a) pri 1600% zväčšení.

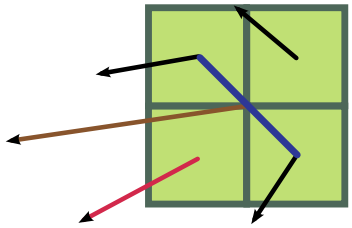
Cannyho hranový detektor

V prvom prípade bol použitý Cannyho hranový detektor [Can86] pre jeho vlastnosť generovať hranicu so šírkou jeden pixel. Na výpočet bola použitá knižnica OpenCV [BB08], nakoľko iniciálna triangulácia je generovaná na CPU. Vyhodnocovanie smeru diagonálnej hrany závisí na štyroch pixeloch zodpovedajúcich hlavným a vedlajším vrcholom diagonály. Podľa počtu pixelov reprezentujúcich vysoko frekvenčnú oblasť môže nastať 16 situácií, pričom šesť situácií je nejednoznačných (obrázok 4.13(b)). V nejednoznačných situáciách je hranová diagonála umiestnená štandardným spôsobom. Pre zníženie vplyvu nejednoznačných situácií by bolo nutné získavanie informácií z väčšieho okolia (obrázok 4.13(c)), čo ale úplne neodstraňuje možnosť výskytu nejednoznačných situácií. Označenie tejto modifikácie je *MeshCanny* avýsledky z rekonštrukcie sú na obrázku 4.11(c).



Obrázok 4.13: Zobrazenie umiestnenia hranových diagonál (modrá) podľa vysokofrekvenčných oblastí (červená). Jednoznačne určené umiestnenie diagonálnych hrán (a), nejednoznačná situácia (b), jednoznačná situácia s prehľadávaním širšieho okolia hrany (c).

Sobelov hranový detektor
V tomto prístupe sa využívajú gradienty v jednotlivých pixeloch. Gradienty sú vypočítané pomocou Sobelovho hranového operátora pre každý pixel. Na základe vypočítaných gradientov sa rozhoduje o smere inicializačnej diagonály. Využívajú sa dve možnosti, a to rozhodnutie na základe maximálneho gradientu zo štvorice gradientov, alebo priemerného gradientu (obrázok 4.14). Aj v tomto prístupe môžu nastať nejednoznačnosti, ale je ich menej ako v predošlom prístupe. Označenie modifikácie využívajúcej maximálny gradient je *MeshSobMax* a pre priemerný gradient je *MeshSobAvg*. Rekonštrukcie pre tieto modifikácie sú znázornené na obrázku 4.11(d),(e).



Obrázok 4.14: Zobrazenie umiestnenia hranovej diagonály (modrá) podľa maximálneho gradientu (červená) alebo priemerného gradientu (hnedá) na základe gradientov v jednotlivých pixeloch (čierna) vyrátaných pomocou Sobelovho operátora.

Lokálna optimalita hrany
Ďalším možným faktorom, ktorý sa dá využiť na rozhodnutie o smere diagonálnych hrán v inicializačnej triangulácii je lokálna optimalita hrany. V prvom kroku základného algoritmu (vytváranie kandidátov) je testovaná lokálna optimalita hrany pre každú hranu v triangulácii. Ak daná hrana nie je lokálne optimálna, tak je označená za kandidáta. V tomto prístupe sa testujú len diagonálne hrany na lokálnu optimalitu. Smer diagonály je určený na základe tejto optimality. Pri tomto predspracovaní sa neberú do úvahy možné konfliktné situácie medzi susednými diagonálami (obrázok 4.4). Táto modifikácia je označená ako *MeshLO* a zobrazená na obrázku 4.11(f).

Náhodné predspracovanie
Odhliadnuc od vstupných dát a v nich významných vysokofrekvenčných oblastiach boli smery inicializačných diagonál určené aj náhodne v modifikácii označenej *MeshRand*. Týmto prístupom sa snažíme čiastočne potlačiť vplyv iniciálnej triangulácie na priebeh optimalizácie. Vyššie prezentované prístupy založené na predspracovaní siete môžu skončiť v horšom lokálnom optime ako iné prístupy z dôvodu rýchlejšej konvergencie ako bolo spomenuté pri modifikácii *MaxGain*. Inicializačná sieť s náhodne orientovanými diagonálami by mala minimalizovať tento problém. Výsledky rekonštrukcie pre jednu z náhodných iniciálnych triangulácií sú zobrazené na obrázku 4.11(g).

Posledné prezentované modifikácie kombinujú predošlé prístupy. Je to kombinácia modifikácií využívajúcej maximalizáciu prírastku a rozšíreného ROI, označená ako *ExpRoiMaxGain*. Ďalšou je kombinácia troch modifikácií a to maximalizácie prírastku, rozšírená oblasť a predspracovanie siete označenej ako *ExpRoiMaxGainMesh* s príslušnou príponou *Canny, SobMax, SobAvg, LO, Rand*. V zhode s očakávaniami tieto posledné algoritmy generujú trianguláciu s najnižšou cenou a s minimálnym počtom vizuálnych artefaktov. Výsledky z niektorých kombinácií sú zobrazené na obrázkoch 4.10 a 4.11.

4.4.2 Vylepšenia v obrazovom priestore

Postupy uvedené nižšie nemenia priebeh výpočtu algoritmu, ovplyvňujú ho len zmenami vo vstupných obrázkoch. Tieto zmenené vstupné obrázky sú rekonštruované a následne prebieha prípadné spracovanie rekonštruovaných obrázkov. Cieľom týchto postupov je potlačenie viditeľných artefaktov.

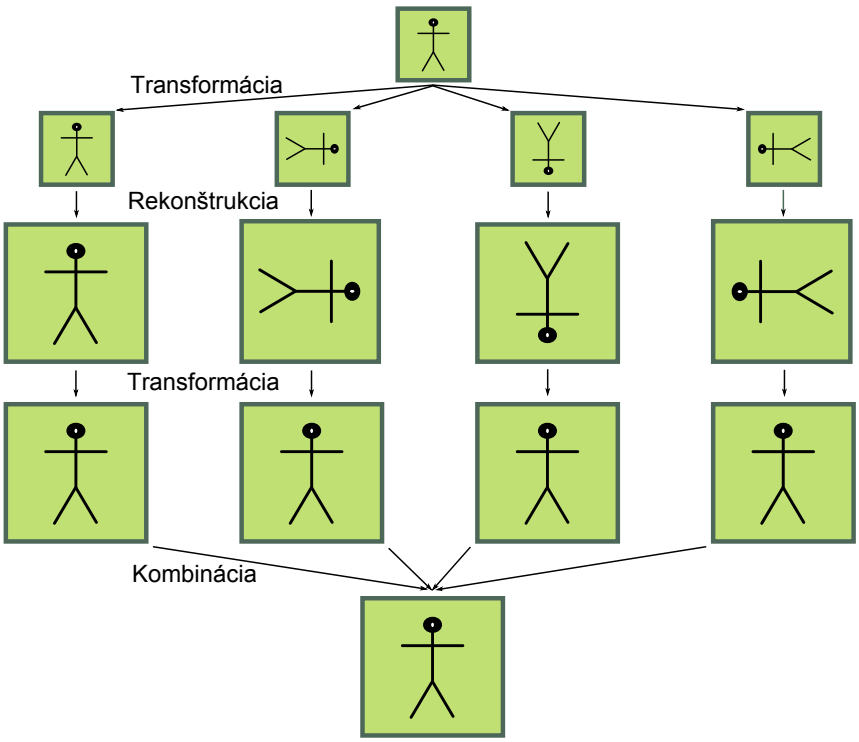
Rotácie

Základom tohto postupu je využitie viacerých rekonštrukcií obrázku na dosiahnutie výslednej rekonštrukcie. Postup je rozdelený do štyroch fáz: transformácie obrázku, výpočet rekonštrukcie, spätnej transformácie a zmiešavania. Postup je v jednoduchosti načrtnutý na obrázku 4.15. Cieľom tohto postupu je odstránenie artefaktov, ktoré vznikajú vo vysokofrekvenčných oblastiach s istou charakteristikou (sklon významných oblastí). Pri transformáciách v prvej fáze sa charakteristika týchto vysokofrekvenčných oblastí mení, čo by malo priniesť vylepšenia pri výslednej rekonštrukcii.

V prvej fáze algoritmu sú použité dve skupiny rotácií. Prvá využíva rotácie o 0, 90, 180, 270 stupňov (ozn. *RotT1*), pričom zo vstupného obrázku generuje 4 obrázky. Druhá skupina využíva rotácie o 0, 90 stupňov (ozn. *RotT2*) a generuje 2 obrázky. Tieto obrázky sú spracované základným algoritmom. Výsledné rekonštruované obrázky sú spätne transformované na pôvodnú orientáciu. Tieto obrázky sú následne zmiešavané do jedného výsledného obrázku. Výsledný obrázok môže vzniknúť ako priemer pixelov vstupných obrázkov (ozn. *Avg*). Ďalšou možnosťou je použitie mediánu z pixelov vstupných obrázkov (ozn. *Med*). Poslednou možnosťou je nájdenie najbližšieho pixela k priemernému pixelu vstupných obrázkov (ozn. *Clo*). Pri použití farebných obrázkov sa toto zmiešavanie vykonáva po jednotlivých farebných kanáloch. Uvedené zmiešavania sú použiteľné pre 4 vstupné obrázky (*RotT1*), pre dva vstupné obrázky (*RotT2*) je použité iba zmiešavanie na základe priemeru pixlov (*Avg*).

Postupné zväčšovanie

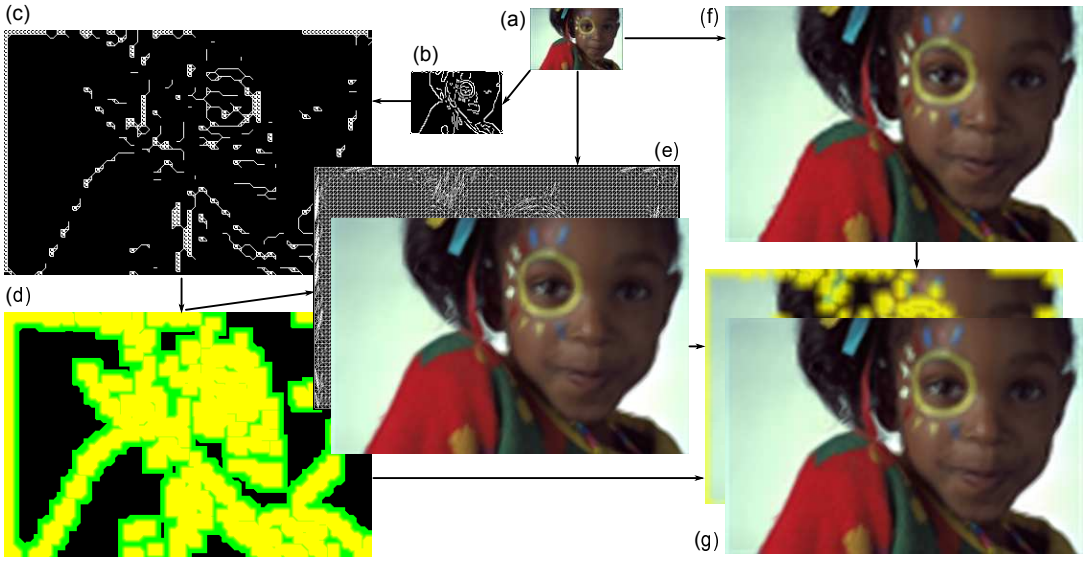
Tento postup pracuje taktiež zo skupinou obrázkov. Táto skupina je generovaná postupnými rekonštrukciami. Predošlé prístupy zväčšovali vstupný obrázok o zväčšovací faktor z_f. V tomto postupe je výsledný obrázok generovaný zo vstupného obrázka postupným zväčšovaním o zväčšovací faktor $z_{f1} = 2$ n-krát, kde $n = \lfloor log_2(z_f) \rfloor - 1$ a posledným zväčšením s faktorom $z_{f2} = \frac{z_f}{2^n}$. Tento postup je možné aplikovať pre $z_f \geq 2$. Hlavným využitím tohto postupu sú rekonštrukcie využívajúce väčšie faktory zväčšenia (≥ 10).



Obrázok 4.15: Náčrt procesu potláčania artefaktov v obrazovom priestore založenom na rotáciách.

4.4.3 Kombinácie konvolučných a DDT techník

V tejto časti predstavíme ďalšiu našu techniku využívajúcu rekonštrukcie založené na DDT spolu s konvolučnými technikami. Výhodou konvolučných techník je relatívne rýchle spracovanie oproti DDT technikám s primeranou kvalitou. Avšak kvalita DDT techník vo vysokofrekvenčných oblastiach je vyššia až na špecifické prípady. Z tohto dôvodu je výhodné kombinovať obe techniky, kde v homogénnych oblastiach bude použitá konvolučná technika a v nehomogénnych oblastiach DDT technika. Celý proces sa skladá zo šiestich fáz, pričom každá fáza generuje čiastkové výsledky – obrázky, ktoré sú použité v ďalších fázach. Tento postup je načrtnutý na obrázku 4.16.



Obrázok 4.16: Náčrt postupu rekonštrukcie obrázku s využitím kombinačných a DDT techník. Vstupný obrázok (a), aplikovaný hranový detektor (b), odhad hrán vo zväčšenom obrázku (c), DDT a kombinačná maska (d), DDT rekonštrukcia (e), konvolučná technika (f), výsledný kombinovaný obrázok (g)

1. fáza - hranový detektor
Na začiatku je na vstupný obrázok 4.16(a) aplikovaný Cannyho hranový detektor na GPU [Fun05]. Výsledkom je obrázok so zvýraznenými hranami o šírke jeden pixel (obrázok 4.16(b)), na základe ktorého sa budú odhadovať nehomogénne oblasti.

2. fáza - odhad hrán
Výsledok z hranového detektora je použitý na odhad hrán vo zväčšenom obrázku. Tento výpočet prebieha v jednotke spracovávajúcej geometriu, ktorý prechádza hranový obrázok z ľavého dolného rohu do pravého horného rohu. Pre každý pixel sa testuje či zodpovedá hrane v hranovom obrázku. Pre pixel reprezentujúci hranu sa testujú štyri susedné

pixely (pravý, pravý horný, horný, l'avý horný), či taktiež reprezentujú hranu. Ak niektorý z testovaných pixelov reprezentuje hranu, tak sa emituje čiara v príslušnom smere a veľkosti zodpovedajúcej veľkosti vo zväčšenom obrázku. Tieto čiary sú následne GPU rasterizované a zapísané do textúry, ktorá reprezentuje obrázok s odhadovanými hranami (obrázok 4.16(c)).

3. fáza - nehomogénne oblasti
Obrázok s odhadnutými hranami slúži ako základ na výpočet nehomogénnych oblastí. Tieto oblasti sa generujú z odhadnutých hrán dilatačným procesom na základe 8-susednosti. Nakoľko obmedzenia GPU neumožňujú zapisovať na rôzne miesta v textúre, ale len na fixne definované (pozícia fragmentu), je tento proces rozdelený na dva kroky. V prvom kroku sa oblasť rozširuje (dilatuje) v x-ovom smere a v druhom kroku sa rozširujú už dilatované oblasti v smere y. Veľkosť tohto rozšírenia, *šírka masky* je používateľom definovaný parameter, ktorý sa zadáva ako násobok zväčšovacieho faktoru. Vo fragmentovom programe sa pre aktuálnu pozíciu fragmentu testujú susedné hodnoty do vzdialenosti šírky masky v textúre s odhadovanými hranami v kladnom a zápornom smere x. Ak sa počas tohto prehľadávania okolia narazí na hodnotu zodpovedajúcu odhadovanej hrane, tak sa na aktuálnu pozíciu zapíše hodnota reprezentujúca *DDT masku*. Tento výsledok je uložený v textúre, ktorá je použitá na čítanie vo fragment programe zodpovedajúcom za rozširovanie v smere y. Výsledkom tohto procesu je binárny obrázok s DDT maskou, kde sú nehomogénne oblasti reprezentované jednotkami a homogénne nulami. DDT maska je použitá na obmedzenie výpočtu DDT len v nehomogénnych oblastiach.

V tejto fáze sa generuje aj *kombinačná maska*, ktorá pozostáva z troch oblastí: nehomogénnej, prechodovej a homogénnej. Nehomogénna oblasť je generovaná vyššie opísaným spôsobom. Následne je generovaná prechodová oblasť, ktorá nadobúda hodnoty z intervalu $\langle 1, 0 \rangle$ lineárne, kde hodnoty 1 susedia s nehomogénnou oblasťou a hodnoty 0 s homogénnou oblasťou. Generovanie prechodovej oblasti prebieha obdobne prehľadávaním susedov v danom smere. Na výstup sa zapisuje maximum z hodnôt aktuálneho fragmentu alebo hodnoty suseda. Hodnota suseda je naviac prenásobená linearizačným faktorom

$$lin_f = 1 - length/width,$$

kde $length$ je vzdialenosť medzi susedom a testovaným fragmentom normalizovaná veľkosťou prechodovej oblasti $width$, čo je používateľom definovaný parameter. Kombinačná maska bude použitá v záverečnej fáze algoritmu. Pre efektívnejšie generovanie masiek a správu pamäte sú obe uložené v jednej textúre, každá v samostatnom farebnom kanáli (obrázok 4.16(d)).

4. fáza - DDT rekonštrukcia
V tejto fáze sa generuje rekonštruovaný obrázok na základe vybranej DDT techniky (obrázok 4.16(e)). Je možné použiť klasickú DDT rekonštrukciu, kde sa optimalizuje celá sieť, prípadne využiť DDT masku a optimalizovať sieť len v nehomogénnych oblastiach. Táto maska je použitá v prvej časti DDT algoritmu, kde sa môžu stať kandidátmi iba tie

hrany, ktoré ležia v nehomogénnych oblastiach. Použitie tejto masky by malo priniesť urýchlenie samotného výpočtu DDT algoritmu.

5. fáza - konvolučná technika
Vstupný obrázok je rekonštruovaný pomocou konvolučnej techniky (obrázok 4.16(f)), kde výpočet je aplikovaný vo fragmentovej jednotke [HTHG01] s použitím Lanczos filtra.

6. fáza - kombinácia
Rekonštruované obrázky z predošlých fáz sú kombinované do výsledného obrázka. Táto kombinácia je riadená kombinačnou maskou, kde výsledná farebná hodnota je lineárnou interpoláciou medzi dvoma vstupnými obrázkami a linearizačný faktor je získaný z kombinačnej masky (obrázok 4.16(g)).

4.5 Výsledky

V tejto časti sa venujeme meraniam výpočtového času, kvalite a cene triangulácií algoritmov uvedených v predošlej časti. Na testovanie sme použili dve skupiny obrázkov. Prvá skupina obsahovala dvanásť obrázkov z reálneho života, ktoré boli vybrané z bežne používaných sád testovacích obrázkov pri spracovaní obrazu. Druhá skupina obsahovala osem umelo vytvorených obrázkov rasterizovaných vo vektorovom editore. Obe skupiny boli zmenšovné na štvrtinovú, respektíve osminovú veľkosť z pôvodnej veľkost iteratívnym zmenšovaním na polovicu veľkosti pomocou bilineárneho filtrovania. Zmenšené obrázky mali rozmery od 64×64 do 250×243 pixelov. Tieto zmenšené obrázky boli zväčšené na pôvodnú veľkosť rôznymi rekonštrukčnými technikami. CPU verzia algoritmu bola implementovaná v jazyku C++. GPU verzie algoritmov využívali OpenGL rozhranie a výpočtové jednotky boli programované v jazykoch glsl a Cg. Minimálna hardvérová požiadavka je *Shader Model 3.0* (DirectX 9c), čo by mali spĺňať takmer všetky bežne dostupné GPU. Pri návrhu algoritmu sme preskúmali možnosť využitia jednotky spracovávajúcej geometriu na priamy výpočet dátovo závislých triangulácií. Detailná analýza ukázala, že využitie tohto prístupu by nebolo dostatočne efektívne. Testy a merania sme vykonali na systéme vybavenom procesorom Intel Pentium 4 3.0GHz, 1GB RAM s nasledujúcimi GPU: NVIDIA GeForce 8800 GTS so 640MB pamäte, NVIDIA GeForce 9600GT s 512MB pamäte a ATI Radeon HD3850 s 512MB pamäte.

4.5.1 Výpočtový čas

Na začiatku boli porovnané výpočtové časy pre jednotlivé modifikácie oproti CPU implementácii výpočtu dátovo závislej triangulácie. Toto meranie bolo rozdelené na 3 časti: *inicializácia*, *výpočet* a *ukončenie*. V inicializačnej časti bol načítaný obrázok a boli vytvorené príslušné dátové štruktúry (východzia triangulácia). Pri GPU verzii boli navyše vytvorené textúry nahraté do GPU pamäte a vytvorený OpenGL kontext. V ďalšej časti bol meraný čas výpočtu lokálne optimálnej triangulácie. V poslednej fáze sa výsledná

triangulácia uložila na disk. Navyše pre GPU algoritmy boli textúry stiahnuté z GPU do CPU pamäte a skonvertované do príslušného formátu. Výsledné časy sú zobrazené v tabuľke 4.1, kde *inicial.* označuje trvanie prvej, *výpočet* druhej a *ukončenie* tretej fázy. Tieto časy sú sčítané a zobrazené v príslušnom stĺpci, označené ako *spolu*. Uvedené výsledky sú v sekundách a sú priemerom pre jednotlivé obrázky danej skupiny.

Meranie výpočtového času (s)								
priemerné	**reálne obrázky**				**vektorové obrázky**			
hodnoty	**inicial.**	**výpočet**	**ukončenie**	**spolu**	**inicial.**	**výpočet**	**ukončenie**	**spolu**
CPU DDT	0,212	6,834	0,221	7,268	0,315	32,104	0,331	32,751
Basic	0,659	0,130	0,043	0,832	0,663	0,180	0,066	0,910
ExpRoi	0,764	0,400	0,047	1,211	0,772	0,286	0,066	1,124
MaxGain	0,657	0,122	0,050	0,829	0,667	0,174	0,067	0,908
ExpRoiMaxGain	0,838	0,268	0,046	1,151	0,848	0,374	0,064	1,286
ExpROI2MaxGain	0,837	0,238	0,048	1,122	0,844	0,275	0,067	1,186
ExpROI3MaxGain	0,839	0,249	0,046	1,133	0,849	0,284	0,065	1,198
ExpROI4MaxGain	0,837	0,257	0,048	1,142	0,845	0,293	0,066	1,203
MeshCanny	0,658	0,126	0,045	0,829	0,668	0,178	0,061	0,907
MeshSobMax	0,667	0,123	0,047	0,838	0,676	0,180	0,066	0,923
MeshSobAvg	0,665	0,125	0,045	0,836	0,673	0,180	0,067	0,920
MeshLO	0,705	0,129	0,047	0,881	0,743	0,178	0,067	0,989
MeshRand	0,664	0,125	0,045	0,833	0,667	0,164	0,067	0,899
ExpRoiMeshSobMax	0,772	0,373	0,047	1,192	0,784	0,324	0,066	1,174
MaxGainMeshSobMax	0,669	0,122	0,044	0,835	0,678	0,174	0,067	0,916
ExpRoiMaxGainMeshSobMax	0,847	0,241	0,046	1,134	0,851	0,282	0,067	1,199
Basic Comb	0,559	0,403	0,043	1,005	0,565	0,433	0,060	1,057
ExpRoiMaxGain Comb	0,555	0,702	0,041	1,297	0,564	0,718	0,058	1,340

Tabuľka 4.1: Priemerné časy výpočtov pre reálne a vektorové obrázky (Cg implementácia na NVIDIA GeForce 8800 GTS GPU.)

Z tabuľky 4.1 vidieť, že GPU verzia je 6—8 krát rýchlejšia ako CPU verzia. Pre vektorové obrázky je toto urýchlenie dokonca ešte vyššie (27—36 krát). Tento vysoký rozdiel bol zapríčinený dlhším výpočtovým časom CPU verzie algoritmu. Bolo to spôsobené preklápaním zanedbateľného počtu hrán pri posledných iteráciách čo bolo zapríčinené vektorovými obrázkami, ktoré neboli poškodené šumom. Pri reálnych obrázkoch toto správanie nebolo pozorované.

Inicializačná fáza výpočtu bola ovplyvnená hlavne veľkosťou fragmentových programov, ich kompiláciou a prenosom na GPU. Najväčšie fragmentové programy sú použité pre modifikácie *ExpRoiMaxGain*, ktorých inicializácia trvala aj najdlhší čas oproti iným modifikáciam. Z celkovej inicializácie najdlhšie trvá vytváranie OpenGL kontextu, čo predstavuje v priemere okolo 60% času. Výpočet lokálne optimálnej triangulácie je taktiež závislý na použitej modifikácii, kde pre modifikáciu *ExpRoi* je tento čas najdlhší. Pri použití *ExpRoiMaxGain* optimalizácie je tento čas znížený rýchlejšou konvergenciou výpočtu vďaka optimalizácii využívajúcej maximalizáciu prírastku. Záverečná fáza je takmer identická pre všetky typy modifikácii.

Modifikácia využívajúca náhodné predspracovanie iniciálnej triangulácie bola spustená 300-krát. Uvedené výsledné časy sú priemerom zo všetkých iterácií. Percentuálny rozdiel medzi najkratším a najdlhším behom modifikácie bol asi 2.5%.

Predstavené modifikácie boli testované taktiež na viacerých GPU, čo bolo možné platformovou nezávislosťou implementácie využívajúcej glsl jazyk. Získané údaje sú uvedené v tabuľke 4.2. Výsledky zodpovedajú priemerným celkovým časom (*spolu*) z tabuľky 4.1. Výsledky pre GPU GeForce 8800 sú takmer identické pre obe implementácie (glsl a Cg), líšia sa v rozsahu pár percent. GeForce 9600 je rýchlejšia v priemere o 35 percent než GeForce 8800 vo všetkých prípadoch. Avšak situácia je odlišná pre GPU firmy ATI. Rozdiely závisia od konkrétnej modifkácie, kde je spomalenie 0.1 až 3.5 násobné v porovnaní s GeForce 8800 GPU. Výrazné spomalanie sa týka najmä modifikácii *ExpRoiMaxGain*, ktoré využívajú fragmentové programy s vyšším počtom inštrukcií. Tieto rozdiely sa dajú vysvetliť rozdielnou architektúrou GPU od firiem NVIDIA a ATI, čo vedie k rozdielnej efektívnosti rôznych operácií.

Výpočtový čas (s)	GeForce 8800GTS	GeForce 9600GT	Radeon HD3850
Basic	0,876	0,593	0,914
ExpRoi	1,176	0,893	1,760
MaxGain	0,865	0,582	0,996
ExpRoiMaxGain	1,230	0,864	4,024
ExpROI2MaxGain	1,171	0,769	4,006
ExpROI3MaxGain	1,180	0,786	4,021
ExpROI4MaxGain	1,184	0,792	4,017
MeshCanny	0,873	0,590	0,908
MeshSobMax	0,884	0,591	0,909
MeshSobAvg	0,878	0,595	0,909
MeshLO	0,934	0,627	0,947
MeshRand	0,868	0,580	0,893
MaxGainMeshSobMax	0,870	0,579	0,979
ExpRoiMeshSobMax	1,185	0,887	1,743
ExpRoiMaxGainMeshSobMax	1,181	0,858	3,975

Tabuľka 4.2: Priemerné časy výpočtov meraných na rôznych GPU (glsl implementácia).

Algoritmy pracujúce v obrazovom priestore

Tieto algoritmy sú založené na spracovaní viacerých obrázkov, ktoré vznikli z rekonštrukcie DDT algoritmom. Nakoľko vstupný obrázok bol spracovávaný viackrát s rôznymi vstupnými parametrami, je celý proces pomalší v závislosti na počte spracovávaných obrázkov (viaceré výpočty DDT rekonštrukcie). Pre výpočet založený na rotáciách je to 2-4 násobné spomalenie. Pre postupné zväčšovanie je výsledný čas závislý na zväčšovacom faktore, pričom môže byť spracovávaných viacero obrázkov. Tieto obrázky sú vždy väčšie a tým pádom je aj výpočet pomalší po každej iterácii. Tieto algoritmy neboli optimalizované, slúžia ako potenciálne možnosti na zvýšenie výslednej kvality. Je možné

ich optimalizovať a dosiahnuť priaznivejšie časové výsledky. Optimalizácia by sa dotkla hlavne komunikácie medzi CPU a GPU, kde by sa vstupný obrázok nemusel viackrát posielať na GPU, ale posielali by sa iba vstupné parametre. To isté platí aj o sťahovaní dát z GPU. Samotné kombinovanie rekonštruovaných obrázkov (rotačné techniky), by mohlo byť taktiež vykonané na GPU.

Kombinačná technika

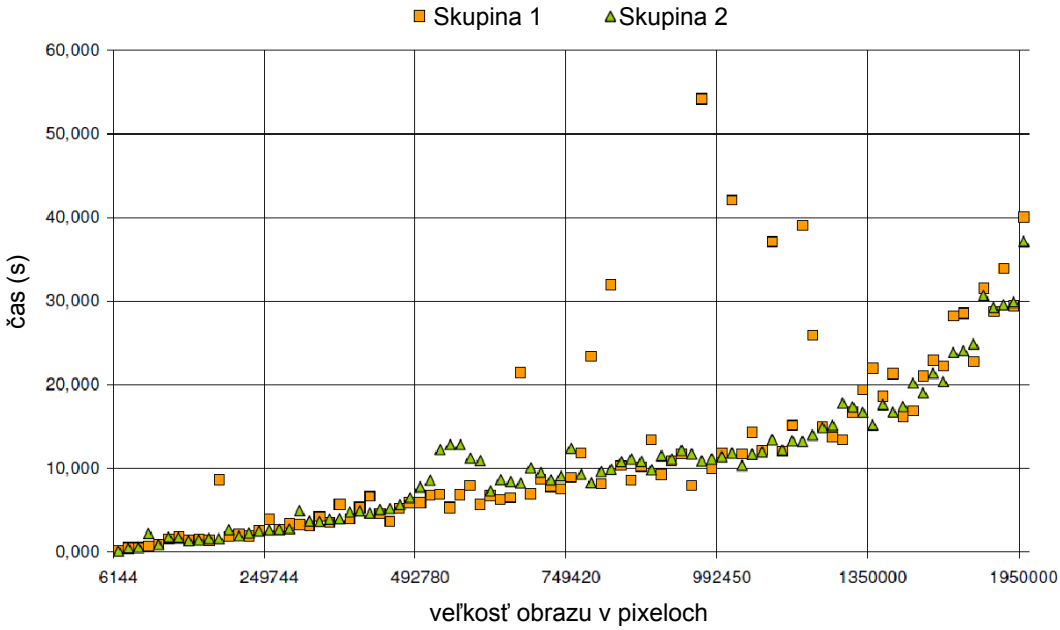
Pri testovaní kombinačnej techniky boli na DDT rekonštrukciu použité modifikácie *Basic*, *ExpRoi*, *MaxGain*, *ExpRoiMaxGain* a Lanczosov filter pre konvolučnú rekonštrukciu. Testovaným parametrom boli šírky masiek. Pre DDT masku je to jeden parameter a to šírka celej masky. Pre kombinačnú masku sa šírka skladá z dvoch parametrov a to šírky nehomogénnej časti a šírky prechodovej oblasti. Testované kombinácie boli *1-1-3*, *1-1-5*, *2-3-5*, *2-3-7*, kde prvé číslo reprezentuje šírku nehomogénnej oblasti, druhé šírku prechodovej oblasti pre konvolučnú masku a tretie nehomogénnu oblasť pre DDT masku. Taktiež bol testovaný prístup, kde pri DDT rekonštrukcii nebola použitá DDT maska a výpočet prebiehal na celej vstupnej triangulácii. Z testov vyplýva, že rôzne šírky masiek a taktiež použitie DDT masky nemá takmer žiaden vplyv na výsledný čas. Rodiely medzi najpomalšou a najrýchlejšou kombináciou boli do jedného percent pre všetky modifikácie. V tabuľke 4.1 sú uvedené niektoré výsledky kombinačnej techniky so šírkami masiek 1-1-5 a využívajúce DDT masku vo výpočte (ozn. *Comb*). Porovnanie kombinačnej techniky s použitými modifikáciami neprinieslo očakávané urýchlenie, dokonca je kombinačná technika zhruba o 15% pomalšia. Je to spôsobené nutnosťou generovať potrebné masky a rekonštrukciu založenú na konvolúcii. Ďalším obmedzením je implementácia algoritmu, ktorá nie je prispôsobená na takéto optimalizácie. Toto obmedzenie je spôsobené najmä OpenGL rozhraním s nedostatočnou správou pamäti. V nasledujúcej časti predstavíme úpravu algoritmu, ktorá bude efektívnejšie prispôsobená takýmto požiadavkám.

Časová zložitosť

Presné analytické vyjadrenie časovej zložitosti uvedených algoritmov závisí na mnohých faktoroch, ktorými sú typ cenovej funkcie, distribúcia dát a iné. Navyše tieto faktory sú špecifické pre každú oblasť použitia. Takáto komplexná analýza nie je cieľom tejto práce. Analýza časovej zložitosti pre podobný prípad ako je akceptovanie a zamietanie kandidátov je prezentovaná v [JP93].

Preto sme použili empirický spôsob merania časovej zložitosti rekonštrukcie obrazu. Zo štandardnej databázy obrázkov používaných v spracovaní obrazu bola vybraná skupina 91 obrázkov rozličných veľkostí. Výsledky sú prezentované na obrázku 4.17 (*skupina 1*), kde je znázornená závislosť medzi výpočtovým časom modifikácie *ExpRoiMaxGain* a veľkosťou vstupného obrázka. Pre malé a stredné obrázky je pozorovaná lineárna závislosť výpočtového času. Pre najväčšie obrázky táto závislosť rastie rýchlejšie ako lineárne. Taktiež boli spozorované niektoré extrémne prípady, kde výpočtový čas výrazne prekročil čas obrázkov s podobnou veľkosťou. Toto bolo pravdepodobne spôsobené „najhorším“ rozložením hrán z pohľadu paralelného spracovania. Počet týchto extrémnych

prípadov nie je významný v porovnaní s veľkosťou celej skupiny. Na porovnanie bola použitá iná skupina obrázkov vytvorená z jedného obrázku vybratého zo *skupiny 1*. Tento obrázok bol preškálovaný na veľkosti obrázkov ako v *skupine 1*. Táto množina obrázkov je označená ako *skupina 2* v obrázku 4.17. Priebeh časovej zložitosti je podobný ako pre *skupinu 1*, ale bez extrémnych prípadov.



Obrázok 4.17: Výpočtový čas modifikácie *ExpRoiMaxGain* pri 400% zväčšení s použitím *skupiny 1* a *skupiny 2*.

4.5.2 Kvalita

Cena triangulácie je dôležitý aspekt merajúci kvalitu triangulácie, pretože cieľom predstavených algoritmov je aj aproximácia triangulácie s minimálnou cenou. Preto sú v tabuľke 4.3 uvedené počiatočné ceny triangulácie (ozn. *cena-inic.*), ceny po rekonštrukcii (ozn. *cena-fin.*). Počet preklopení počas optimalizačného procesu je uvedený v príslušnom stĺpci (ozn. *preklop.*). Tieto informácie sa týkajú len algoritmov pracujúcich vo výpočtovom priestore. Vidieť, že najlepšou optimalizáciou z pohľadu výslednej ceny je *ExpRoiMaxGain MeshSobMax*. Táto optimalizácia dosiahla výslednú cenu triangulácie pri použití reálnej testovacej skupiny podobnú ako CPU verzia a pre vektorovú skupinu bola táto cene dokonca nižšia. Ceny triangulácií získané z rôznych GPU (NVIDIA, ATI) sa odlišujú len nepatrne (desatinné miesta). Toto je pravdepodobne spôsobené rozličnou internou presnosťou a rozdielmi v architektúrach použitých GPU.

Meranie cien triangulácií						
priemerné	**reálny dataset**			**vektorový dataset**		
hodnoty	**cena-inicial.**	**cena-finálna**	**preklopenia**	**cena-inicial.**	**cena-finálna.**	**preklopenia**
CPU DDT	1549,901	691,641	17727	1549,762	341,809	8626
Basic	1549,901	825,867	16440	1549,762	432,785	7263
ExpRoi	1549,901	795,837	17089	1549,762	436,413	7423
MaxGain	1549,901	777,825	14808	1549,762	420,602	6696
ExpRoiMaxGain	1549,901	753,367	15066	1549,762	365,502	7054
ExpRoi2MaxGain	1549,901	754,047	15109	1549,762	360,335	7161
ExpRoi3MaxGain	1549,901	753,288	15105	1549,762	363,982	7118
ExpRoi4MaxGain	1549,901	753,445	15092	1549,762	363,736	7106
MeshCanny	1421,372	782,363	16090	1178,585	371,752	6957
MeshSobMax	1158,469	747,882	13956	892,388	335,774	6251
MeshSobAvg	1327,295	765,469	15529	996,268	345,514	6533
MeshLO	2182,660	830,892	23410	2223,705	484,217	9405
MeshRand	1603,079	799,060	17620	1509,789	424,567	7477
ExpRoiMeshSobMax	1158,469	743,879	14267	892,388	324,463	6448
MaxGainMeshSobMax	1158,469	706,423	12767	892,388	291,721	6125
ExpRoiMaxGainMeshSobMax	1158,469	702,667	12847	892,388	287,173	6187

Tabuľka 4.3: Meranie priemerných cien triangulácií pre reálne a vektorové obrázky.

Vizuálnu kvalitu výsledkov sme hodnotili pomocou niekoľkých perceptuálnych metrík, ktoré sú bežne používané v oblasti spracovania obrazu na hodnotenie kvality. Vybranými metrikami boli:

- korelácia (*correlation*),
- krížová korelácia (*cros-correlation*),
- stredná kvadratická chyba (*mean square error - MSE*),
- odstup signál-šumu (*signal-to-noise ratio - SNR*) [GW06],
- špičkový odstup signál-šum (peak signal-to-noise ratio - *PSNR*) [GW06],
- univerzálny koeficient kvality obrazu (*universal image quality index - UIQI*) [WB02],
- index štrukturálnej podobnosti (*structural similarity index - SSIM*) [WBSS04].

Okrem vyššie spomínaných modifikácií sme použili štandardné rekonštrukčné techniky na porovnanie kvality rekonštrukcie: bilineárny, b-splinový, Lanczos filter. Knižnicu ImageMagic [Stu10] sme použili na generovanie výsledkov pre konvolučné techniky. Priemerné hodnoty získané z meraní výsledkov sú uvedené v tabuľke 4.4 pre reálne obrázky a tabuľke 4.5 pre vektorové obrázky. Vyššie hodnoty znamenajú vyššiu kvalitu okrem MSE, kde nižšia hodnota znamená lepšie rekonštrukčné výsledky.

Meranie kvality – reálna skupina obrázkov							
priemerné hodnoty	**Korelácia%**	**Krížová kor.%**	**MSE**	**SNR(dB)**	**PSNR(dB)**	**UIQI**	**SSIM**
bilinear filter	96,0128	94,8248	294,6944	17,7384	24,1055	0,3549	0,6368
b-spline filter	95,3048	93,3421	378,2071	16,5979	22,9650	0,2739	0,5958
Lanczos filter	96,3680	95,6046	249,8571	18,5692	24,9363	0,3972	0,6619
CPU DDT	96,1801	95,2550	267,7697	18,2405	24,6076	0,3712	0,6527
Basic	96,1418	95,1506	272,882	18,1580	24,5238	0,3830	0,6487
ExpRoi	96,1573	95,1779	270,689	18,1989	24,5647	0,3848	0,6500
MaxGain	96,1485	95,1618	272,204	18,1713	24,5372	0,3833	0,6491
ExpRoiMaxGain	96,1587	95,1725	270,879	18,1914	24,5572	0,3847	0,6499
ExpRoi2MaxGain	96,1600	95,1693	270,729	18,1952	24,5611	0,3847	0,6500
ExpRoi3MaxGain	96,1590	95,1728	270,801	18,1927	24,5586	0,3847	0,6500
MeshCanny	96,1503	95,2385	269,6155	18,2017	24,5696	0,3643	0,6512
MeshSobMax	96,1706	95,2919	266,6022	18,2560	24,6239	0,3664	0,6532
MeshSobAvg	96,1647	95,3088	267,7879	18,2344	24,6023	0,3654	0,6525
MeshLO	96,1252	95,1739	273,3184	18,1348	24,5027	0,3624	0,6489
MeshRand	96,1425	95,2478	270,6778	18,1859	24,5530	0,3714	0,6509
ExpRoiMeshSobMax	96,1747	95,3222	266,0252	18,2659	24,6338	0,3667	0,6535
MaxGainMeshSobMax	96,1875	95,2785	264,4245	18,2972	24,6652	0,3675	0,6545
ExpRoiMaxGainMeshSobMax	96,1905	95,2870	263,9944	18,3075	24,6754	0,3676	0,6547
RotT1 Avg	96,1555	95,3079	268,4780	18,2201	24,5881	0,3644	0,6516
RotT1 Med	96,1613	95,2692	267,6991	18,2317	24,5997	0,3654	0,6522
RotT1 Clo	96,1482	95,2574	269,8336	18,1979	24,5658	0,3646	0,6510
RotT2	96,1492	95,2577	269,3774	18,2033	24,5712	0,3641	0,6511
Magnify	96,0882	94,9105	284,8859	17,9519	24,3190	0,3518	0,6418
Basic Comb	96,1451	95,1959	271,1632	18,1861	24,5569	0,3573	0,6511
ExpRoiMaxGainComb	96,1609	95,1896	269,1784	18,2198	24,5907	0,3586	0,6523
ExpRoiMaxGainMeshSobMax RotT1 Med	96,1908	95,2868	263,9931	18,3075	24,6754	0,3676	0,6547

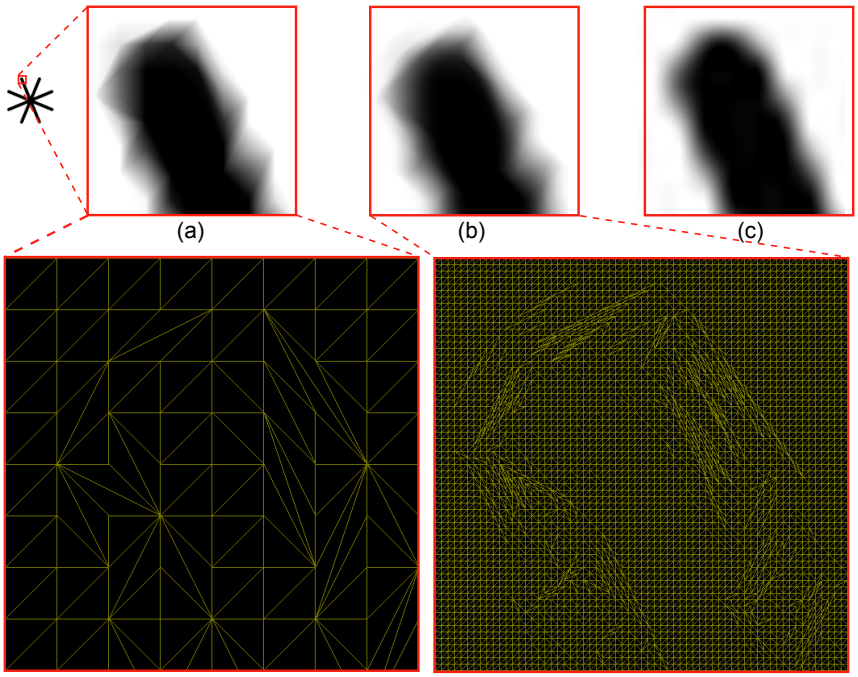
Tabuľka 4.4: Priemerné kvalitatívne výsledky merané perceptuálnymi metrikami pre reálne obrázky.

Algoritmus využívajúci náhodné predspracovanie siete bol vykonaný 300-krát. Z výsledkov bol vybratý medián (usporiadanie na základe cien triangulácií), kde v tabuľke 4.4, 4.5 sú uvedené namerané hodnoty. Rozpätie medzi najlepšou a najhoršou rekonštrukciou bolo do jedného percenta.

Technika využívajúca postupné zväčšovanie obrázkov generuje horšie výsledky ako základný algoritmus. Je to spôsobené kumuláciou chyby, ktorá nastáva pri každom kroku rekonštrukcie. Táto technika má možné využitie pri viacnásobných zväčšeniach (≥ 10), kde klasické DDT techniky produkujú výrazné artefakty vo forme viditeľných trojuholníkov (obrázok 4.18(a)). Technika postupného zväčšovania tieto vysokofrekvenčné oblasti aproximuje väčším počtom trojuholníkov. Na druhej strane je táto oblasť viac rozmazaná, čo je prijateľnejšie pre pozorovateľa (obrázok 4.18(b))

Meranie kvality – vektorová skupina obrázkov							
priemerné hodnoty	**Korelácia%**	**Krížová kor.%**	**MSE**	**SNR(dB)**	**PSNR(dB)**	**UIQI**	**SSIM**
bilinear filter	97,8058	97,1848	347,9350	21,1548	23,5693	0,7315	0,9008
b-spline filter	97,0695	95,8742	507,1912	19,6697	22,0842	0,6226	0,8785
Lanczos filter	98,2447	98,0020	256,9001	22,4676	24,8820	0,6045	0,9088
CPU DDT	98,2257	97,7630	271,7720	22,2660	24,6804	0,7927	0,9194
Basic	98,1522	97,6275	286,916	22,0282	24,4229	0,7757	0,9156
ExpRoi	98,1715	97,6450	283,838	22,0616	24,4563	0,7772	0,9165
MaxGain	98,1570	97,6162	287,492	22,0196	24,4143	0,7760	0,9159
ExpRoiMaxGain	98,1982	97,6887	278,899	22,1660	24,5607	0,7782	0,9179
ExpRoi2MaxGain	98,2057	97,6992	277,238	22,1865	24,5812	0,7785	0,9183
ExpRoi3MaxGain	98,2007	97,6920	278,375	22,1702	24,5650	0,7783	0,9180
MeshCanny	98,1747	97,7063	279,1330	22,1577	24,5722	0,7121	0,9171
MeshSobMax	98,2002	97,7780	272,1485	22,2593	24,6737	0,7133	0,9185
MeshSobAvg	98,1972	97,7638	273,6493	22,2364	24,6508	0,7131	0,9183
MeshLO	98,1198	97,5813	292,6324	21,9156	24,3301	0,7098	0,9142
MeshRand	98,1455	97,6385	286,8905	22,0242	24,4386	0,7111	0,9156
ExpRoiMeshSobMax	98,2232	97,8118	267,3245	22,3284	24,7429	0,7141	0,9196
MaxGainMeshSobMax	98,2597	97,8665	259,1128	22,4350	24,8494	0,7148	0,9213
ExpRoiMaxGainMeshSobMax	98,2763	97,9157	255,4667	22,4962	24,9107	0,7155	0,9222
RotT1 Avg	98,2365	97,6735	284,2416	22,0647	24,4791	0,7099	0,9151
RotT1 Med	98,1492	97,7007	281,0627	22,1171	24,5315	0,7115	0,9161
RotT1 Clo	98,1443	97,6818	283,3571	22,0795	24,4939	0,7109	0,9155
RotT2	98,2410	97,6630	285,5158	22,0473	24,4617	0,7103	0,9148
Magnify	97,9858	97,5888	308,4090	21,7249	24,1393	0,6864	0,9096
Basic Comb	98,1440	97,7198	270,3707	22,2470	24,6464	0,6022	0,9124
ExpRoiMaxGain Comb	98,1712	97,7663	265,7571	22,3342	24,7337	0,6032	0,9138
ExpRoiMaxGainMeshSobMax RotT1 Med	98,2762	97,9157	255,5023	22,4956	24,9101	0,7158	0,9222

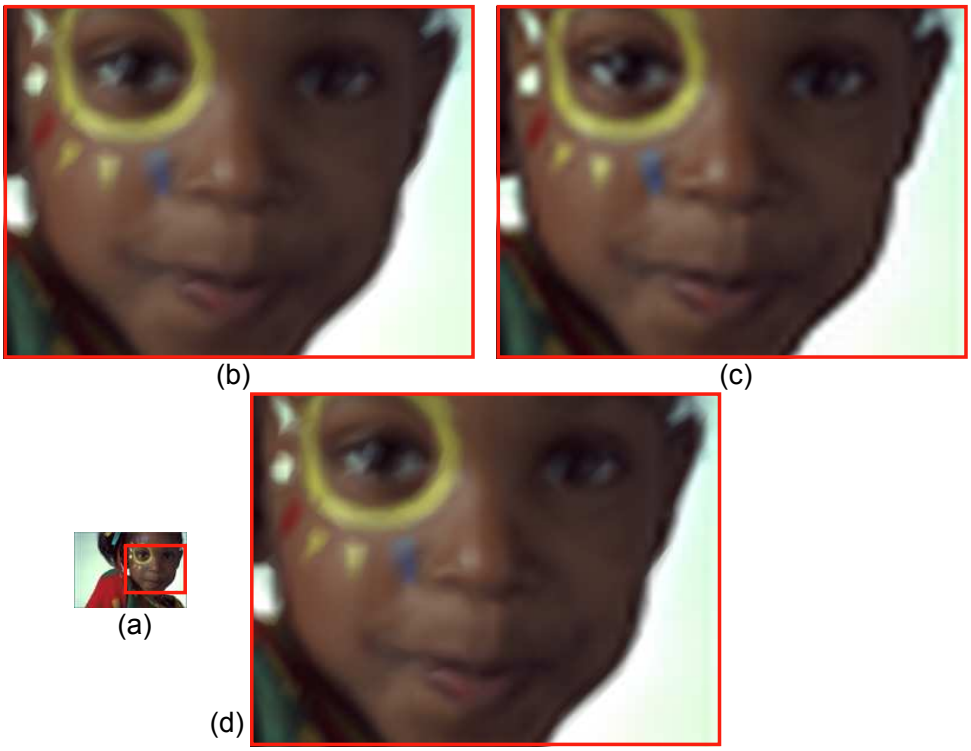
Tabuľka 4.5: Priemerné kvalitatívne výsledky merané perceptuálnymi metrikami pre vektorové obrázky.



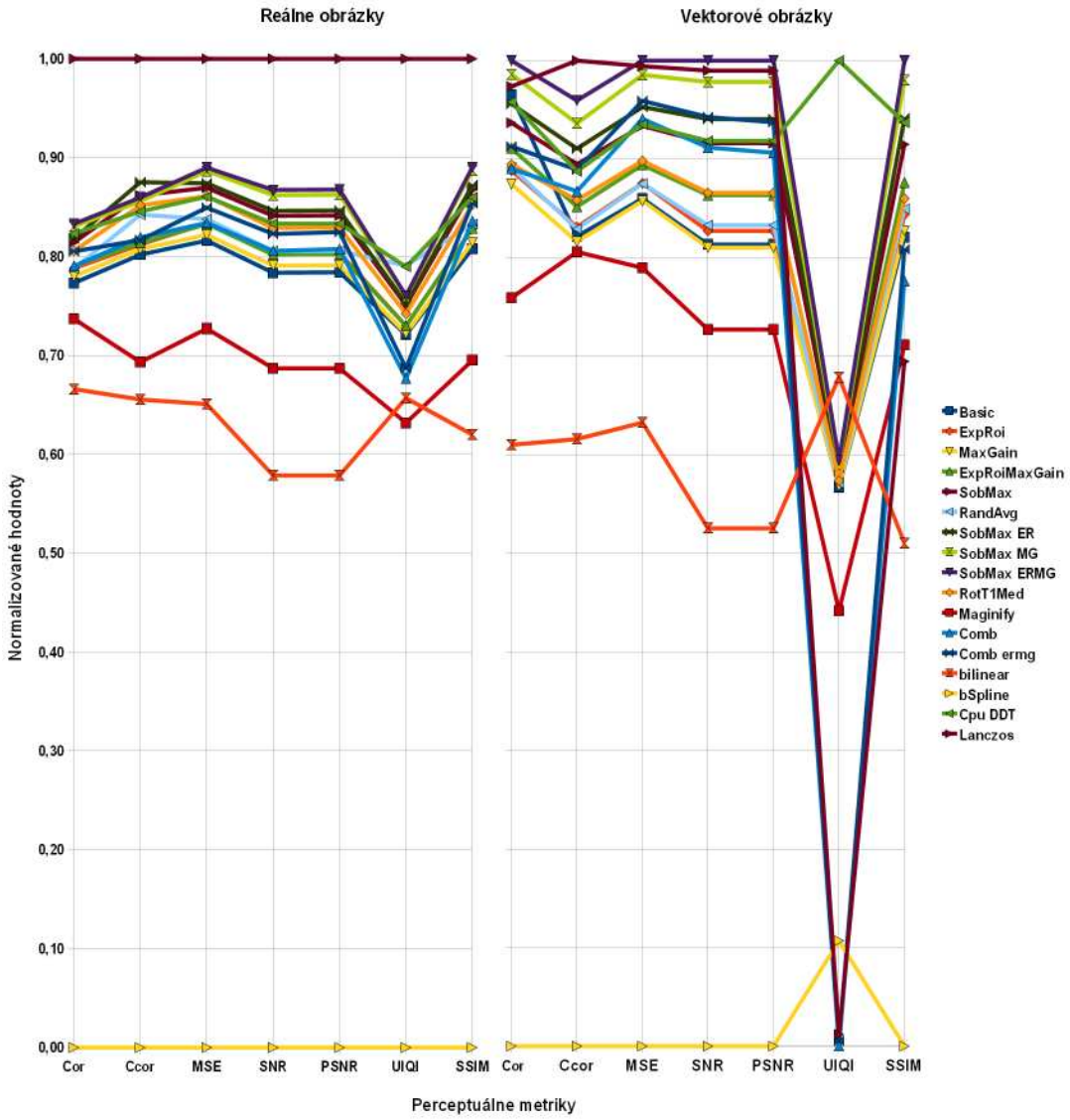
Obrázok 4.18: Zobrazenie 3200% zväčšenia s použitím základnej techniky(a), techniky postupného zväčšovania (b) a Lanczosovho filtra (c). V dolnom riadku sú pre DDT techniky znázornené príslušné triangulácie.

Pri testovaní kvality kombinačnej techniky boli použité rovnaké parametre ako pri testovaní času pre túto techniku. Z testov vyplýva, že rôzne šírky masiek a taktiež použitie DDT masky pri výpočte nemajú veľký vplyv na výslednú kvalitu techniky. Rozdiely medzi jednotlivými kombináciami pre ľubovolné metriky sa líšia do dvoch percent. Aj pri takýchto malých rozdieloch vyšla najlepšie kombinačná technika využívsajúca DDT masku a šírkami 1-1-5 (obrázok 4.19). Výsledky pre túto techniku sú uvedené v tabuľke 4.4 a tabuľke 4.5 pod označením *Comb*. Avšak pri porovnaní kombinačnej techniky s modifikáciou, ktorá bola použitá na DDT rekonštrukciu (porovnanie *Basic* s *Basic Comb*), táto technika priniesla zvýšenie kvality takmer vo všetkých metrikách. Toto správanie bolo očakávané, nakoľko použitý Lanczsov filter v homogénnych oblastiach generuje kvalitnejšiu rekonštrukciu ako po častiach lineárna funkcia založená na DDT prístupe. V ďalšej práci by sme chceli preskúmať využitie v kombinačnej technike aj modifikácie založené na predspracovaní iniciálnej triangulácie.

Nakoľko sme na meranie použili viacero perceptuálnych metrík a je náročné zhodnotiť všetky ich parametre, tak sme na grafe 4.20 znázornili vybrané techniky v prehľadnejšej forme. Znázornené techniky reprezentujú základné techniky a techniky generujúce kvalitnejšie výsledky z pohľadu perceptuálnych metrík. Namerané hodnoty boli normalizované pre každú perceptálnu metriku zvlášť.



Obrázok 4.19: Výsledky kombinačnej techniky pri 800% zväčšení. Originálny obrázok (a), ExpRoiMaxGain rekonštrukcia (b), Lanczos filter (c), výsledný kombinovaný obrázok (d).



Obrázok 4.20: Znázornenie normalizovaných hodnôt perceptuálnych metrík pre vybrané typy modifikácií a techník.

Ohodnotenie kvality obrazu ľudským vizuálnym systémom nie je možné merať pomocou perceptuálnych metrík. Z tohto dôvodu sú na obrázkoch 4.21 a 4.22 zobrazené niektoré výsledky rôznych rekonštrukcií na subjektívne porovnanie pre čitateľa. Pre konvolučné techniky sú viditeľné blokové artefakty v oblastiach vysokých frekvencií.

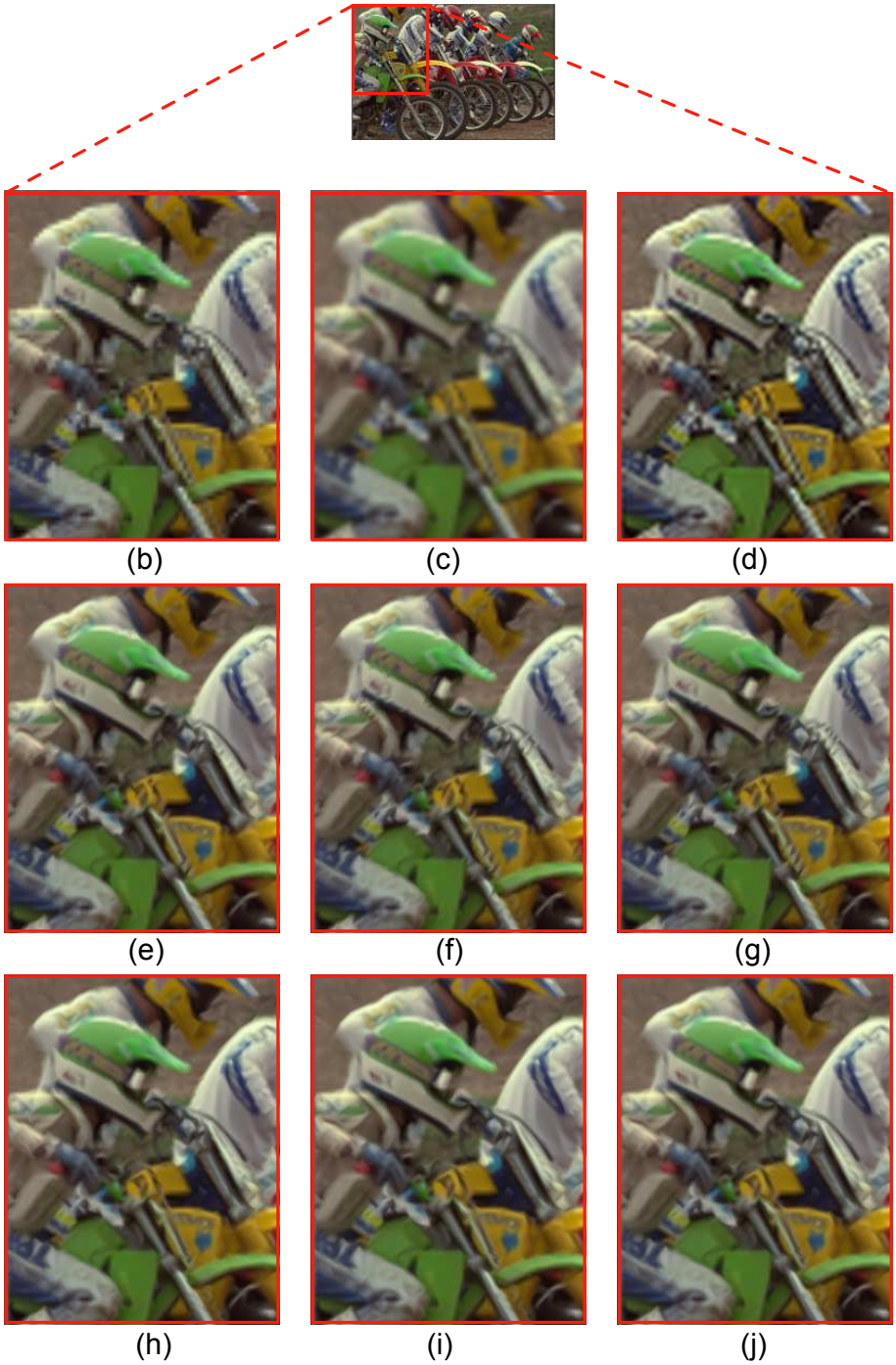
Z vizuálneho hodnotenia a nameraných hodnôt sme medzi techniky s najlepšou rekonštrukciou zaradili nasledujúce: *ExpRoiMeshSobMax*, *MaxGainMeshSobMax*, *ExpRoiMaxGainMeshSobMax*. Z pohľadu efektívnosti je najvýhodnejší algoritmus *MaxGainMeshSobMax*, ktorý generuje kvalitnú rekonštrukciu za prijateľný čas.

4.6 Možné optimalizácie a budúca práca

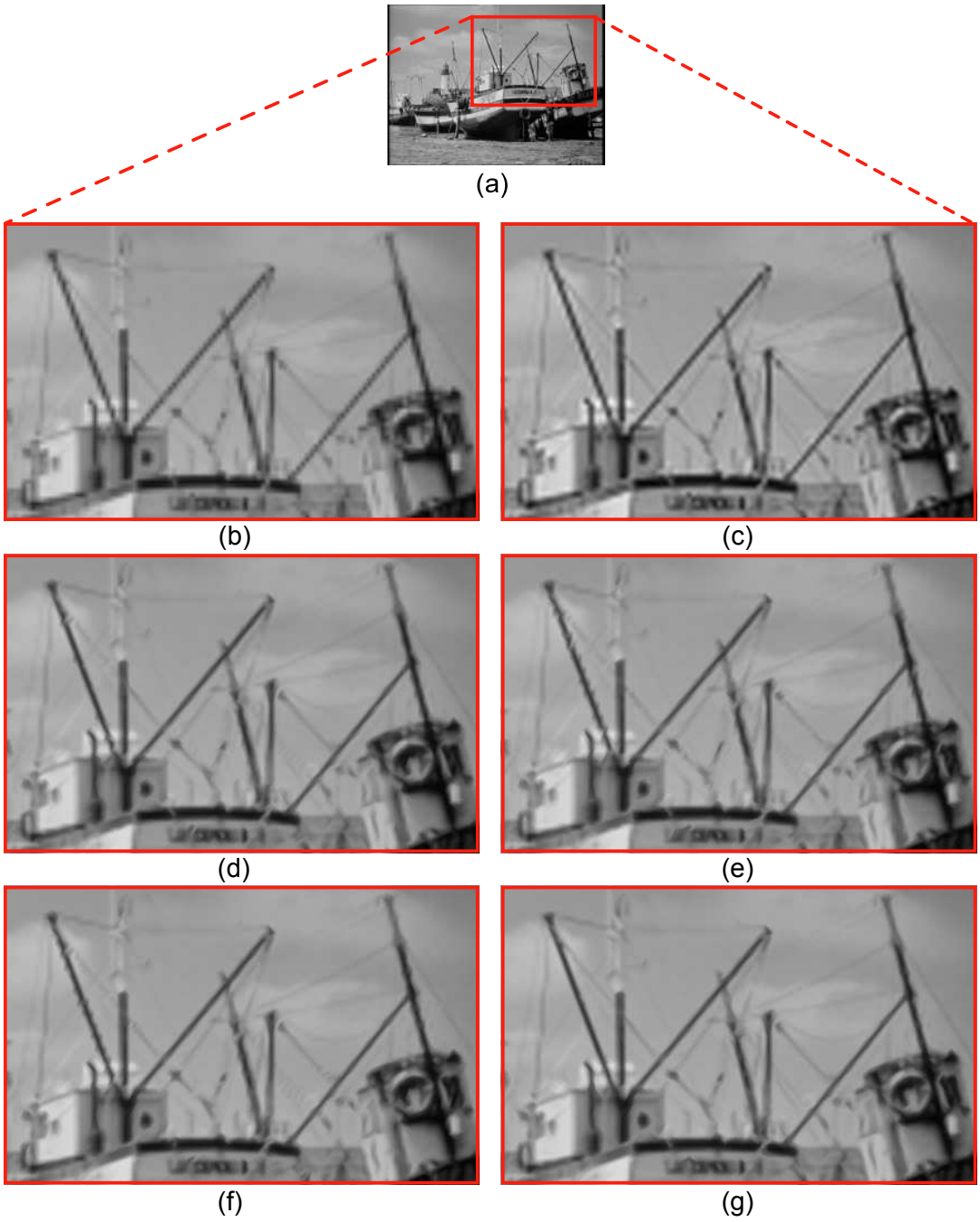
V tejto časti uvedieme možné optimalizácie predošlého základného algoritmu. Tieto optimalizácie môžu priniesť vyššiu efektivitu vo výpočte a tak znížiť výsledný čas. Navrhnuté optimalizácie zasahujú do pôvodného návrhu algoritmu a vyžadujú sofistikovanejšie dátové štruktúry, prípadne preusporiadania v dátových štruktúrach. OpenGL rozhranie nie je príliš vhodné na takéto zmeny kvôli jeho obmedzeným možnostiam správy pamäte. Výhodnejšie je využiť nové rozhranie na programovanie GPU a paralelných systémov ako je OpenCL. Uvedené návrhy nebudú predstavené detailne. Mali by však vyhovovať možnostiam moderných GPU a OpenCL rozhraniu.

Prvá zmena sa týka druhého kroku algoritmu (akceptovanie a zamietanie kandidátov), kde sa každá hrana testuje či je kandidátom a ak je, tak sa prehľadáva jej okolie. Na základe porovnávania identifikátorov sa rozhoduje o ďalšom spracovaní hrany. Tento spôsob výpočtu nie je efektívny hlavne pri malom počte kandidátov a vedie k zbytočnému testovaniu hrán. Pri počiatočných iteráciách optimalizácie sa v triangulácii vyskytuje dostatočný počet kandidátov, ale ich počet rýchlo klesá s nasledujúcimi iteráciami, čo vedie k neefektívnemu spracovaniu hrán. Preto navrhujeme po prvom kroku algoritmu (vytváranie kandidátov) vytvoriť množinu hrán, ktorá bude obsahovať iba hrany označené za kandidátov a ich okolie. Druhý krok algoritmu by spracovával už len túto menšiu množinu, čo by malo priniesť vyššiu efektivitu výpočtu. Samozrejme, vytváranie takejto množiny vyžaduje isté náklady, a preto by sa mohla vytvárať až po istom počte iterácií, respektíve pokiaľ bude triangulácia obsahovať menší počet kandidátskych hrán. Moment kedy je už výhodné vytvárať novú množinu hrán je náročné určiť a závisí od mnohých faktorov. Táto hranica by mohla byť určená vstupným parametrom algoritmu.

Ďalší návrh sa týka operácie preklápania hrany. Pri neparalelnej verzii algoritmu sa pri preklopení hrany zaktualizujú všetky štruktúry v ROI okolí hrany a pokračuje sa ďalšou hranou. Po ukončení iterácie je zrejmé, že miesta kde nebola preklopená hrana sú lokálne optimálne. Pri preklopení hrany sa daná hrana stáva lokálne optimálnou, ale táto zmena má vplyv na okolité hrany z jej ROI. Preto v ďalšej iterácii stačí testovať len hrany z okolia preklopenej hrany z predošlej iterácie. Táto situácia je však pri paralelnej verzii



Obrázok 4.21: Výsledky rekonštrukcie pri 400% zväčšení. Originálny obrázok (a), bilineárny filter (b), b-spline filter (c), Lanczos filter (d), CPU DDT rekonštrukcia (e), Basic (f), ExpRoiMax-Gain (g), SobMax (h), ExpRoiMaxGain MeshSobMax (i), ExpRoiMaxGain MeshSobMax Comb (j) modifikácie.



Obrázok 4.22: Výsledky rekonštrukcie pri 400% zväčšení. Originálny obrázok (a), bilineárny filter (b), Lanczos filter (c), CPU DDT rekonštrukcia (d), Basic (e), SobMax (f), ExpRoiMaxGain SobMax (g) modifikácie.

komplikovanejšia. Keďže hrana na preklopenie sa vyberá z množiny kandidátskych hrán, ktorá môže byť väčšia ako ROI okolie preklopenej hrany, je nedostatočné testovať len jej okolie. V prvom kroku algoritmu (vytváranie kandidátov) je potrebné testovať hrany a ich ROI okolia, ktoré boli v predchádzajúcej iterácii kandidátmi. V tomto prípade nastáva podobná situácia ako v predchádzajúcej optimalizácii. Pri počiatočných iteráciách triangulácia obsahuje veľa kandidátskych hrán a preto vytváranie menšej množiny len s týmito hranami by bolo neefektívne. Po niekoľkých iteráciách však počet kandidátskych hrán výrazne klesá a tak by bolo zrejme výhodné vytvárať špeciálnu množinu len s kandidátskymi hranami a ich okoliami. Táto množina, štruktúra by sa automaticky mohla použiť v predchádzajúcom návrh.

Dodatočným vylepšením pôvodného algoritmu by mohli byť nedeterministické prístupy ako je simulované žíhanie alebo genetické algoritmy. Tieto prístupy by mohli mať vplyv nielen na výpočtový výkon, ale aj na výslednú kvalitu rekonštrukcie. Samozrejme problémom týchto techník je vhodné nastavenie inicializačných parametrov.

Pri implementácii algoritmu a analýze výsledkov sme si všimli situáciu, že iniciálne hrany je možné rozdeliť do troch disjunktných skupín: vodorovné, zvislé a diagonálne hrany. Toto je možné využiť pri úplne novom návrhu algoritmu založenom na farbení. Každá skupina hrán (farba) by mohla byť spracovávaná samostatne, bez nutnosti riešenia kolízií. Po spracovaní danej skupiny by sa zaktualizovali dátové štruktúry a prešlo by sa na spracovávanie nasledujúcej skupiny. Pri všeobecnom prípade rozloženia vrcholov triangulácie môže dôjsť k situácii, že jeden trojuholník bude pozostávať z dvoch alebo troch hrán rovnakého typu (farby). V takejto situácii dochádza ku konfliktu, ktorý je nutné riešiť. Jedným z riešení môže byť prefarbenie grafu, čo ale nie je jednoduchá úloha. Podľa nášho predpokladu a pozorovaní takéto konfliktné situácie nanastávajú ak sú vrcholy triangulácie usporiadané v mriežke (spracovanie obrazu). Samozrejme, že tento predpoklad je potrebné dokázať a podrobnejšie navrhnúť daný algoritmus.

4.7 Zhrnutie

V tejto časti bola uvedená technika pre výpočet paralelnej dátovo závislej triangulácie využívajúcej GPU, špeciálne zameraná na rekonštrukciu obrazu—zväčšovanie. Techniku sme porovnali pomocou perceptuálnych metrík, vizuálnej kontroly s neparalelnou implementáciou na CPU a interpolačnými technikami založenými na konvolúcii. Vizuálne výsledky boli podobné ako pri sériovom prístupe a lepšie ako pri konvolučných technikách. Z pohľadu perceptálnych metrík boli výsledky porovnateľné. Vďaka paralelizácii a GPU implementácii bol výpočtový čas 8-krát rýchlejší ako pri sériovej implementácii. Taktiež boli prezentované viaceré modifikácie, ktoré vedú k zlepšeniu vizuálnej kvality.

Prezentované riešenie výpočtu dátovo závislej triangulácie založené na paralelizovanom Lawsonovom optimalizačnom procese je všeobecné a umožňuje vypočítať lokálne optimálnu trianguláciu s použitím rôznych cenových funkcií na GPU. Tieto algoritmy

môžu byť využité v špeciálnych optimalizčných úlohách a môžu poskytnúť lokálne optimálne siete s rôznymi vlastnosťami aj mimo oblasti spracovania obrazu. Prezentované algoritmy sa dajú využiť pre ľubovoľnú distribúciu dát a ľubovoľné optimalizačné úlohy založené na dátovo závislých trianguláciách.

Obe verzie algoritmov (CPU a GPU) požadujú iniciálnu trianguláciu. Pri probléme rekonštrukcie obrazu je generovanie tejto triangulácie jednoduché a na GPU môže byť vykonané efektívnym a rýchlym spôsobom. V iných aplikačných oblastiach je zrejme výhodnejšie vytvoriť iniciálnu trianguláciu na CPU a potom dátové štruktúry transferovať na GPU pre ďalšiu optimalizáciu.

Kapitola 5

Záver

Moderné GPU ponúkajú výpočtový výkon, ktorý významne prevyšuje výkon bežných procesorov počítača. Funkcionalita, ktorú ponúkajú sa neustále rozširuje a nové možnosti programovacieho rozhrania sa neustále vyvíjajú pre potreby programátorov. Dnešné osobné počítače obsahujú väčšinou moderné GPU, ktoré sú využívané najmä na počítačové hry a inak je ich výkon nevyužitý. V súčasnosti vznikajú rôzne tzv. GPGPU aplikácie, ťažiace z vysokého výkonu GPU. V našej práci sme sa preto zamerali na výpočty s využitím GPU, ktoré nesúvisia primárne so zobrazovaním. Špeciálne sme sa venovali dvom oblastiam a to prúdovému spracovaniu objemových dát a rekonštrukcii obrazu založenej na paralelných dátovo závislých triangu̇láciach.

V prvej oblasti sme navrhli využitie GPU pri spracovaní objemových dát. Dnešné GPU sa v tejto oblasti využívajú hlavne na renderovanie dát. Existujú však algoritmy využívajúce GPU aj na ich spracovanie, ale ich nevýhodou je vysoká pamäťová náročnosť, ktorá obmedzuje použitie týchto algoritmov na bežne dostupných počítačoch. Preto sme sa zamerali na prúdové spracovanie objemových dát, špeciálne na spracovanie po rezoch, kde je v pamäti počítača (GPU) len nevyhnutne potrebné množstvo dát. Tento prístup efektívnejšie využíva pamäťové zdroje počítača a umožňuje spracovanie objemových dát veľkých rozmerov na bežných počítačoch. Navrhli sme jednoduché aplikačné rozhranie, ktoré je univerzálne a je ho možné využiť viacerými dostupnými hardvérovými prostriedkami počítača ako je CPU, prípadne jeho inštrukčná sada SSE a moderné GPU. Ďalej sme uviedli viaceré všeobecné techniky využívajúce funkcionalitu súčasných GPU pri spracovaní objemových dát. Tieto techniky sme využili pri implementovaní algoritmov spracovávajúcich objemové dáta na GPU v prúdovom režime. Jednalo sa o dva filtre, všeobecný filter na výpočet separovateľnej a neseparovateľnej konvolúcie a filter detekujúci tubulárne štruktúry v objemových dátach. Pri implementácii sme použili viaceré techniky, na ktorých sme demonštrovali ich vplyv na výsledný čas spracovania dát. Techniky urýchľovali výpočet oproti základnej implementácii, ktorá nevyužíva efektívne potenciál a funkcionalitu GPU. Tieto techniky sme porovnávali s rovnakými algoritmami využívajúcimi CPU, prípadne SSE. Dosiahli sme výrazne urýchlenie, ktoré záviselo od konkrétnych algoritmov. Tieto urýchlenia sú výraznejšie pri algoritmoch s väčšou zložitosťou. Pre veľké objemové dáta a vysokých nárokoch na výpočet boli tieto urýchlenia

rádovo vyššie oproti CPU algoritmom a vo väčšine prípadov aj oproti SSE optimalizovaným verziám. Musíme však podotknúť, že využitie GPU pri menej zložitých problémoch, alebo menších objemových dátach nie je vždy výhodnejšie, a to z dôvodu potreby inicializácie a prenosu dát na/z GPU. Implementované algoritmy sme využili aj v reálnej aplikácii, v ktorej sa tiež dosiahlo výrazné urýchlenie spracovania dát. V tejto aplikácii sa využívali všetky dostupné zdroje počítača (GPU, SSE), čo sa ukázalo ako výhodné riešenie pri spracovaní objemových dát.

V druhej oblasti sme využili výkon GPU pri výpočte dátovo závislých triangulácií. Pre tento výpočet sme navrhli všeobecný paralelný algoritmus založený na Lawsonovom optimalizačnom procese. Pomocou paralelnej verzie je možné efektívnejšie využiť aj dnešné viacjadrové CPU. Tento paralelný algoritmus je univerzálny, takže pri optimalizácii je možné použiť rôzne cenové funkcie. Pri návrhu tohto algoritmu sme brali do úvahy možnosti a obmedzenia GPU. Samotný algoritmus sme implementovali v OpenGL prostredí využívajúc výkon GPU. Výsledná triangulácia bola využitá pri rekonštrukcii obrazu, konkrétne pri zväčšovaní. Základná verzia algoritmu bola vykonávaná 8-krát rýchlejšie oproti sekvenčnej verzii na CPU. Tento algoritmus sme optimalizovali a navrhli modifikácie, ktoré viedli ešte k rýchlejšiemu výpočtu. Pri rekonštrukcii obrazu sme sa zamerali na kvalitu výslednej rekonštrukcie, najmä na korektnú rekonštrukciu hranových oblastí, na ktoré je pozorovateľ najviac senzitívny. Navrhli sme viaceré modifikácie základného algoritmu s cieľom zvýšiť výslednú kvalitu rekonštrukcie. Tieto modifikácie ovplyvňovali priamo výpočet algoritmu, prípadne spracovávali už rekonštruovaný obraz. Viacerými modifikáciami sme dosiahli výraznejšie zvýšenie kvality rekonštrukcie s malým dopadom na výsledný čas. Niektoré z týchto techník je možné využiť aj pri sekvenčnom prístupe za účelom zvýšenia kvality rekonštrukcie. Na meranie kvality rekonštrukcie sme použili viaceré perceptuálne metriky. Merania preukázali, že predstavený algoritmus a jeho modifikácie sú konkurencieschopné v porovnaní s klasickými rekonštrukčnými algoritmami založenými na konvolučných technikách. Výsledné rekonštruované obrázky boli podľa nášho názoru dokonca kvalitnejšie, s menším výskytom nežiadúcich artefaktov . Taktiež sme navrhli prípadné úpravy predstaveného algoritmu, ktoré by mali zefektívniť jeho výpočet a ktoré by sme radi preverili v budúcnosti. Načrtli sme aj ideu nového algoritmu na rekonštrukciu obrazu, ktorej by sme sa chceli venovať v budúcej práci.

V predkladanej práci sme splnili ciele a uviedli sme teoretické a praktické výsledky využitia GPU na negrafických výpočtoch. Tieto výsledky boli implementované do knižníc a programu, ktoré je možné využívať, či už na ďalší výskum alebo v praxi. Výsledky práce ukazujú, že výskum v oblasti GPGPU je žiaduci a prínosný. Predpokladáme, že GPU sa bude neustále viac využívať v ďalších oblastiach vedy a ich výkon si nájde uplatnenie aj v bežných aplikáciach pre bežných používateľov.

Literatúra

[Ahn07] Song Ho Ahn. OpenGL Pixel Buffer Object (PBO) [Internet]. http://www.songho.ca/opengl/gl_pbo.html, 2007. [cited Apr. 20, 2010].

[AKTD00] L. Alboul, G. Kloostreman, C. Traas, and R. V. Damme. Best data-dependent triangulations. *Journal of Computational and Applied Mathematics*, 119(1-2):1–12, 2000.

[AMD10] AMD. ATI Stream Computing - Technical Overview [Internet]. http://ati.amd.com/technology/streamcomputing/, 2010. [cited Apr. 20, 2010].

[Aur91] F. Aurenhammer. Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[BB08] Adrian Kaehler ByGary Bradski. *Computer Vision with the OpenCV Library*. O'Reilly MediaReleased, 2008.

[BCG+06] Prosenjit Bose, Jurek Czyzowicz, Zhicheng Gao, Pat Morin, and David R. Wood. Simultaneous diagonal flips in plane triangulations. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 212–221. ACM Press, 2006.

[BGM04] L. Battiato, G. Gallo, and G. Messina. SVG Rendering of Real Images Using Data Dependent Triangulation. In *Proceedings of Spring Conference on Computer Graphics 2004*, pages 191–198, 2004.

[BHG+10] I. Buch, Matt J. Harvey, T. Giorgino, D. P. Anderson, and G. De Fabritiis. High-Throughput All-Atom Molecular Dynamics Simulations Using Distributed Computing. In *Journal of Chemical Information and Modeling*, March 2010. doi: 10.1021/ci900455r.

[Bjo04] Kevin Bjorke. High-Quality Filtering. In Randima Fernando, editor, *GPU Gems*, chapter 24. Addison Wesley, 2004.

[Bra99] Ronald N. Bracewell. *The Fourier Transform & Its Applications (3rd ed.)*. McGraw-Hill Science/Engineering/Math, 1999.

[Bro91] J. Brown. Vertex Based Data Dependent Triangulations. *Computer Aided Geometric Design*, 8(3):239–251, 1991.

[Can86] John Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986.

[Cor99] Intel Corporation. Real and Complex FIR Filter Using Streaming SIMD Extensions, order number: 243643-002 [Internet]. www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/245711.htm?page=1, 1999. [cited June 23, 2009].

[Cor10a] Microsoft Corp. DirectX [Internet]. http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx, 2010. [cited Apr. 20, 2010].

[Cor10b] Microsoft Corp. HLSL - High Level Shading Language for DirectX [Internet]. http://msdn.microsoft.com/en-us/library/bb509561(VS.85).aspx, 2010. [cited Apr. 20, 2010].

[Cor10c] NVIDIA Corporation. Cg Homepage [Internet]. http://developer.nvidia.com/page/cg_main.html, 2010. [cited Apr. 20, 2010].

[ČTS+10] Michal Červeňanský, Zsolt Tóth, Juraj Starinský, Andrej Ferko, and Miloš Šrámek. Parallel GPU-based data-dependent triangulations. *Computers & Graphics*, 34:125–135, 2010.

[CUD10] CUDA. *NVIDIA CUDA - Programming Guide, Version 3.0*. NVidia Corporation, 2010.

[DLR90] N. Dyn, D. Levin, and S. Rippa. Data Dependent Triangulations for Piecewise Linear Interpolation. *IMA Journal of Numerical Analysis*, 1(10):137–154, 1990.

[Ebe08] David Eberly. Eigensystems for 3×3 Symmetric Matrices (Revisited). Technical report, Geometric Tools, LLC, March 2008. [cited Apr. 20, 2010].

[f3d10] f3d. f3d volume format and tools [Internet]. http://gerlach.viskom.oeaw.ac.at/f3dtrac, 2010. [cited Apr. 20, 2010].

[FG06] I. Fisher and C. Gotsman. Fast approximation of high order Voronoi diagrams and distance transforms on the GPU. *Journal of Graphics Tools*, 11(4):39–60, 2006.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FNP96] A. Ferko, L. Niepel, and T. Plachetka. Criticism of Hunting Minimum Weight Triangulation Edges. In *Proceedings of the 12th Spring Conference on Computer Graphics*, pages 259–264, 1996.

[FNVV98] Alejandro F. Frangi, Wiro J. Niessen, Koen L. Vincken, and Max A. Viergever. Multiscale vessel enhancement filtering. In *MICCAI'98 - Medical Image Computing and Computer-Aided Intervention*, pages 130 – 137, 1998.

[Fun05] James Fung. Computer vision on the GPU. In M. Pharr, editor, *GPU Gems 2*. Addison Wesley, 2005.

[Gf05] Chris Guy and Dominic ffytche. *An Introduction to The Principles of Medical Imaging*. Imperial College Press, 57 Shelton Street, Covent Garden, London WC2H 9HE, England, 2005.

[GNU10] GNU. GSL - GNU Scientific Library [Internet]. http://www.gnu.org/software/gsl/, March 2010. [cited Apr. 20, 2010].

[GPG10] GPGPU. GPGPU General-Purpose Computation on Graphics Hardware [Internet]. http://www.gpgpu.org, 2010. [cited Apr. 20, 2010].

[GPU10] GPUGRID. GPUGRID.net [Internet]. http://www.gpugrid.net, 2010. [cited Apr. 20, 2010].

[Gro10a] The Khronos Group. The Khronos Group Open Standards for Media Authoring and Acceleration [Internet]. http://www.khronos.org, 2010. [cited Apr. 20, 2010].

[Gro10b] The Khronos Group. OpenGL The Industry's Foundation for High Performance Graphics [Internet]. http://www.opengl.org, 2010. [cited Apr. 20, 2010].

[GW05] Joachim Georgii and Rüdiger Westermann. Mass-Spring Systems on the GPU. *Simulation Modelling Practice and Theory*, 13:693–702, 2005.

[GW06] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[HCK+99] K. E. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Proceedings of ACM SIGGRAPH 1999*, pages 277–286. ACM, 1999.

[HD06] O. Hjelle and M. Dahlen. *Triangulations and Applications, Series: Mathematics and Visualization*. Springer, 2006.

[HE99] Matthias Hopf and Thomas Ertl. Accelerating 3D Convolution using Graphics Hardware. In *VISUALIZATION '99: Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, Washington, DC, USA, 1999. IEEE Computer Society.

[HH04] Shawn Hargreaves and Mark Harris. 6800 Leagues Under the Sea: Deferred Shading [Internet]. http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf, 2004. [cited Apr. 20, 2010].

[HKRs+06] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[HTHG01] M. Hadwiger, T. Theußl, H. Hauser, and M. E. Gröller. Hardware-Accelerated High-Quality Filtering on PC Hardware. In *Proceedings of Vision, Modeling, and Visualization 2001*, pages 105–112, 2001.

[JP93] Mark T. Jones and Paul E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput*, 14:654–669, 1993.

[KF01] I. Kolingerová and A. Ferko. Multicriteria-optimized Triangulations. *The Visual Computer*, 17(6):380–395, 2001.

[KH01] O. Kreylos and B. Hamann. On Simulated Annealing and the Construction of Linear Spline Approximations for Scattered Data. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):17–31, 2001.

[Khr10] Khronos Group. The NV_transform_feedback specification [Internet]. http://www.opengl.org/registry/specs/NV/transform_feedback.txt, 2010. [cited Apr. 20, 2010].

[KP05] Craig Kolb and Matt Pharr. Option pricing on the GPU. In Matt Pharr, editor, *GPU Gems 2*, chapter 45. Addison-Wesley, 2005.

[KW05a] Peter Kipfer and Rüdiger Westermann. Improved GPU Sorting. In Matt Pharr, editor, *GPU Gems 2*, chapter 46. Addison Wesley, 2005.

[KW05b] Jens Krüger and Rüdiger Westermann. A GPU Framework for Solving Systems of Linear Equations. In Matt Pharr, editor, *GPU Gems 2*, chapter 44, pages 703–718. Addison-Wesley, 2005.

[LHK+04] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33. ACM, 2004.

[LSMT99] C. Charles Law, William J. Schroeder, Kenneth M. Martin, and Joshua Temkin. A multi-threaded streaming pipeline architecture for large structured data sets. In *VIS '99: Proceedings of the conference on Visualization '99*, pages 225–232, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[MA03] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[MeV09] MeVisLab. MeVisLab: Medical Image Processing and Visualisation [Internet]. http://www.mevislab.de, 2009. [cited Apr. 20, 2010].

[MR06] W. Mulzer and G. Rote. Minimum weight triangulation is NP-hard. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 1–10. ACM, 2006.

[Ope09] OpenCL. *The OpenCL Specification*. Khronos OpenCL Working Group, 2009.

[ORO05] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate Sum and Dot Product. *SIAM J. Sci. Comput.*, 26(6):1955–1988, 2005.

[PM03] Craig Peeper and Jason L. Mitchell. Introduction to the DirectX® 9 High Level Shading Language. In *ShaderX2: Introductions and Tutorials with DirectX 9.0*, pages 1–61. Wordware Publishing, Inc, November 2003.

[Pod07] Victor Podlozhnyuk. Image Convolution with CUDA. Technical report, NVIDIA Corporation, June 2007. [cited Apr. 20, 2010].

[Ros05] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[RT06] G. Rong and T.-S. Tan. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 109–116. ACM Press, 2006.

[RTCS08] G. Rong, T.-S. Tan, T.-T. Cao, and Stephanus. Computing Two-dimensional Delaunay Triangulation Using Graphics Hardware. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 89–97. ACM Press, 2008.

[SA10] Mark Segal and Kurt Akeley. The OpenGL® Graphics System: A Specification [Internet]. http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf, 2010. [cited Apr. 20, 2010].

[Sch93] L. L. Schumaker. Computing Optimal Triangulations Using Simulated Annealing. In *Selected papers of the international symposium on Free-form curves and free-form surfaces*, pages 329–345. Elsevier Science Publishers B. V., 1993.

[SCI09] SCIRun. SCIRun: A Scientific Computing Problem Solving Environment [Internet]. http://www.scirun.org, 2009. [cited Apr. 20, 2010].

[ŠD02] Miloš Šrámek and Leonid I. Dimitrov. f3d – a file format and tools for storage and manipulation of volumetric data sets. In *In 1-st International Symposium on 3D Data Processing, Visualization and Transmission, Padova, Italy, June 19-21, 2009*, pages 368–372, 2002.

[ŠDSČ04] Miloš Šrámek, Leonid I. Dimitrov, Matúš Straka, and Michal Červeňanský. The f3d tools for processing and visualization of volumetric data. In *Journal of Medical Informatics and Technologies*, pages MIP–71–MIP–79, 2004.

[Shi05] Oles Shishkovtsov. Deferred Shading in S.T.A.L.K.E.R. In Matt Pharr, editor, *GPU Gems 2*, chapter 9, pages 143–166. Addison-Wesley, 2005.

[SL05] Thilaka Sumanaweera and Donald Liu. Medical Image Reconstruction with the FFT. In Matt Pharr, editor, *GPU Gems 2*, chapter 48, pages 765–784. Addison-Wesley, 2005.

[ST04] Robert Strzodka and Alexandru Telea. Generalized Distance Transforms and Skeletons in Graphics Hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym '04)*, pages 221–230, 2004.

[Stu10] ImageMagick Studio. ImageMagick [Internet]. www.imagemagick.org, 2010. [cited Apr. 20, 2010].

[SW04] D. Su and P. Willis. Image Interpolation by Pixel-Level Data-Dependent Triangulation. *Computer Graphics Forum*, 23(2):189–202, 2004.

[TÓ6] Zsolt Tóth. Towards an Optimal Texture Reconstruction. In *CESCG 2000-2005 Best Paper Selection*, pages 197–212. Österreichische Computer Gesellschaft Wien, 2006.

[Vaš05] Anton Vaško. SIMD Optimization in Volume Rendering. Master's thesis, Comenius University, 2005.

[VVS+07] Andrej Varchola, Anton Vaško, Viliam Solčány, Leonid I. Dimitrov, and Miloš Šrámek. Processing of volumetric data by slice- and process-based streaming. In *AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 101–110, New York, NY, USA, 2007. ACM.

[WB02] Z. Wang and A.C. Bovik. A Universal Image Quality Index. *IEEE Signal Processing Letters*, 9(3):81–84, 2002.

[WBSS04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[YBS01] X. Yu, B. S. Bryan, and T. W. Sederberg. Image Reconstruction Using Data-Dependent Triangulation. *Computer Graphics and Applications*, 21(3):62–68, 2001.