VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

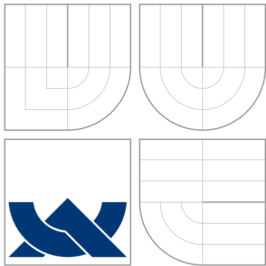
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ GPU PRO AKCELEROVANÉ ZPRACOVÁNÍ OBRAZU

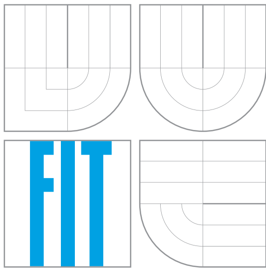
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LADISLAV BAČÍK
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VYUŽITÍ GPU PRO AKCELEROVANÉ ZPRACOVÁNÍ OBRAZU

IMAGE PROCESSING ON GPUS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LADISLAV BAČÍK
AUTHOR

VEDOUCÍ PRÁCE — Doc. RNDr. PAVEL SMRŽ, Ph.D.
SUPERVISOR

BRNO 2008

Zadání diplomové práce

Řešitel **Bačík Ladislav, Bc.**
Obor Počítačová grafika a multimédia
Téma **Využití GPU pro akcelerované zpracování obrazu**
Kategorie Počítačová grafika

Pokyny:

1. Prostudujte základy zpracování obrazu.
2. Analyzujte současné možnosti využití programovatelných GPU pro zpracování obrazu.
3. Vyberte vhodné algoritmy (např. filtrace obrazu, detekce hran, DFT, atd.) a navrhněte metodu jejich akcelerovaného výpočtu pomocí GPU.
4. Experimentujte s vaší implementací a případně navrhněte vlastní modifikace.
5. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje.
6. Vytvořte stručný plakát prezentující vaši diplomovou práci, její cíle a výsledky.

Literatura:

- Dle pokynů vedoucího.
- http://www.gpgpu.org/
- http://developer.nvidia.com/object/cuda.html

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění prvních tří bodů zadání.

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

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

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

Vedoucí **Smrž Pavel, doc. RNDr., PhD.**, UPGM FIT VUT
Konzultant **Španěl Michal, Ing.**, UPGM FIT VUT
Datum zadání 24. září 2007
Datum odevzdání 19. května 2008

Licenční smlouva

Licenční smlouva v kompletním znění je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Výňatek z licenční smlouvy:

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.

2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.

3. Autor souhlasí se zvěřejněním díla v databázi přístupné v mezinárodní síti:
   - ● ihned po uzavření této smlouvy
   - ○ 1 rok po uzavření této smlouvy
   - ○ 3 roky po uzavření této smlouvy
   - ○ 5 let po uzavření této smlouvy
   - ○ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací).

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

Abstrakt

Tato diplomová práce se zabývá moderními technologiemi grafického hardware a jeho využitím pro obecné výpočty. Soustředí se především na architekturu unifikovaných procesorů a implementaci algoritmů pomocí programového rozhraní CUDA. Základem je zvolit vhodný algoritmus demonstrující výkon GPU. Cílem této práce je implementace multiplatformní knihovny, která poskytne vektorizaci diskrétních volumetrických dat. Pro tento proces byl zvolen algoritmus Marching cubes, hledající povrch nasnímaného objektu. V knihovně bude obsažena jak část pro zpracování na grafickém zařízení, tak na CPU. Na závěr obě varianty porovnáme a vyjádříme se k výhodám či nevýhodám těchto přístupů.

Klíčová slova

technologie grafického hardware, obecné výpočty na grafické výpočetní jednotce, GPGPU, CUDA, volumetrická data, Marching cubes

Abstract

This master thesis deals with modern technologies in graphic hardware and using their for general purpose computing. It is primary focused on architecture of unified processors and algorithm implementation via CUDA programming interface. Thesis base is to choose suited algorithm for GPU horsepower demonstration. Main aim of this work is implementation of multiplatform library offering algorithms for discrete volumetric data vectorization. For this purpose was chosen algorithm Marching cubes that is able to find surface of processed object. In created library will be contained algorithm runnable on graphic device and also one runnable on CPU. Finally we compare both variants and discuss their pros and cons.

Keywords

technology of graphics hardware, general purpose computing on graphics processing units, GPGPU, CUDA, volumetric data, Marching cubes

Citace

Ladislav Bačík: Využití GPU pro akcelerované zpracování obrazu, diplomová práce, Brno, FIT VUT v Brně, 2008

Využití GPU pro akcelerované zpracování obrazu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. RNDr. Pavla Smrže, Ph.D.. Uvedl jsem též veškeré literární prameny a publikace, ze kterých jsem čerpal.

....
Ladislav Bačík
15. května 2008

© Ladislav Bačík, 2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**
1.1 Rozvržení textu . . . 4

2 Technologie grafického hardwaru **6**
2.1 Historie . . . 6
2.2 GPGPU . . . 7
2.2.1 Přehled . . . 7
2.3 DirectX 10 . . . 9
2.3.1 Cíle . . . 10
2.3.2 Architektura . . . 10
2.3.3 Vykreslovací řetězec . . . 11
2.4 NVidia GeForce 8 . . . 12
2.4.1 Architektura . . . 12
2.5 Technologie CUDA . . . 13
2.5.1 Architektura . . . 14
2.5.2 Programovací model . . . 14
2.5.3 HW implementace . . . 15
2.5.4 Paměťový model . . . 16
2.5.5 Aplikační programové rozhraní . . . 16

3 Zobrazování volumetrických dat **18**
3.1 Zobrazovací metody . . . 19
3.1.1 Metody hledající povrch . . . 19
3.1.2 Přímé zobrazovací metody . . . 20

4 Návrh **21**
4.1 Motivace a cíle . . . 21
4.2 Vstupní data . . . 22
4.3 Marching cubes . . . 23
4.3.1 Algoritmus . . . 23
4.3.2 Charakteristika . . . 24
4.3.3 Stínování . . . 24
4.4 Výstupní data . . . 26
4.5 Možné vylepšení a modifikace . . . 27
4.5.1 Vyhlazení . . . 27
4.5.2 Decimace . . . 27

5 Praktická realizace **29**
5.1 Vývojové prostředky . . . 29
5.2 Vstupní rozhraní . . . 29
5.2.1 Předzpracování dat . . . 30
5.3 Výstupní rozhraní . . . 31
5.4 Vektorizace objemových dat . . . 32
5.4.1 Základní princip . . . 32
5.4.2 Sdílení vrcholů . . . 34
5.4.3 CPU implementace . . . 36
5.4.4 GPU implementace . . . 37
5.4.5 Multi GPU . . . 41

6 Dosažené výsledky **43**
6.1 Měřená vstupní data . . . 43
6.2 Testovaný hardware . . . 43
6.3 Počet vrcholů a trojúhelníků . . . 44
6.4 Režie volání funkcí na CPU a jader na GPU . . . 45
6.5 Globální a sdílená paměť . . . 46
6.6 Duplicitní vrcholy . . . 47
6.7 Zhodnocení . . . 48

7 Závěr **51**

A Demonstrační aplikace **54**
A.1 Instalace . . . 54
A.2 Spuštění aplikace . . . 54

B Ukázky z vizualizace polygonálních modelů **55**

C Obsah DVD **58**

Seznam obrázků

2.1 Porovnání výkonu CPU a GPU 7
2.2 OpenGL vykreslovací pipeline 8
2.3 Direct3D 10 pipeline 11
2.4 Blokové schéma GeForce 8800 12
2.5 Vrstvy API CUDA 14
2.6 Struktura programovacího modelu 15
2.7 Struktura paměťového modelu 17

3.1 Vizualizace volumetrických dat v medicíně 18

4.1 Struktura knihovny 22
4.2 Mřížka objemových dat 23
4.3 Marching cubes – konfigurace případů 23
4.4 Osvětlení polygonálního modelu 25
4.5 Metody výpočtu normálového vektoru 25
4.6 Konfigurace lokální topologie vrcholu 28
4.7 Decimace vrcholů 28

5.1 Struktura výstupního rozhraní knihovny 31
5.2 Označení vrcholů a hran krychle 33
5.3 Sdílení vrcholů 34
5.4 Proces vektorizace na CPU 36
5.5 Kroky vektorizace na GPU 38
5.6 Paralelní Prefix sum 40
5.7 Vektorizace na více grafických zařízeních 41

6.1 Režie volaných funkcí 46
6.2 Vektorizace objemových dat různých velikostí 48
6.3 Podíl jednotlivých fází vektorizace 48
6.4 Vektorizace polygonální sítě s duplicitními vrcholy 49

B.1 Detailní pohled na strukturu polygonální sítě 55
B.2 Generované modely v závislosti na hodnotě hranice 56
B.3 Orientace normálových vektorů 57
B.4 Spojení částí modelu z více grafických zařízení 57

Kapitola 1

Úvod

Komplexní soustavu hardwarových komponent, nazvanou počítač, sestrojil člověk za účelem zjednodušit a urychlit výpočetní algoritmy používané ve skutečném světě. Již skoro zapomenuté jsou doby, kdy počítač zabral celou místnost ve výzkumných centrech. S postupujícím časem se počítače vyvinuly do podoby velice efektivního a každodenního pomocníka, bez ohledu na obor či účel. Tento vývoj má za následek neustálé rozpínání prostoru použitelnosti. Jelikož se počítače stávají stále výkonnější, paradoxně se také zvyšují nároky člověka.

Prakticky již od počátku byly počítače využívány hlavně na různé komplikované matematické či vědecké problémy, simulace a spousty dalších úkolů, které denně zaměstnávají jejich výpočetní čas. Je zřejmé, že zkrácení výpočetního času bylo vždy hlavní motivací pro pokračující vývoj. Po dlouhou dobu bylo jedinou možností, jak zpracovávat obecné výpočty, využít služeb *CPU* (z angl. Central Processing Unit) nebo specializovaných elektronických obvodů. Ovšem trh si žádá vysoké výkony i od ostatních hardwarových komponent. V průběhu několika posledních let se jednou z nejprogresivnějších komponent staly grafické karty, na čemž má svůj podíl hlavně tlak současného herního průmyslu. S příchodem programovatelných či unifikovaných procesorů, tvořících architekturu dnešních *grafických procesorových jednotek* (zkr. GPU), byla myšlenka využití GPU pro obecné a náročné výpočty přenesena z papírových návrhů do reality.

V této práci jsme se zabývali aplikací vybraného „problematického" algoritmu na grafické zařízení, k čemuž jsme využili programové rozhraní CUDA. CUDA poskytuje programové prostředky, díky nimž se vývoj aplikací využívající GPU více sjednotil s klasickými programovými zvyky a postupy. Cílovým algoritmem se stal *Marching cubes*, který umožňuje vektorizaci diskrétních volumetrických dat do podoby polygonálních sítí. Algoritmus tvoří základ námi vyvíjené multiplatformní knihovny, která představuje výstup praktické části této práce.

1.1 Rozvržení textu

V kapitole 2 je hlouběji popsána problematika *GPGPU* a s ní spojené technologie ze světa grafiky. Současně si osvojíme teoretický základ programového modelu rozhraní CUDA a charakteristických vlastností podporovaných grafických zařízení.

Kapitola 3 poskytuje čtenáři zásadní informace spojené s volumetrickými daty, které jsou doménou moderní medicíny. Jsou zde vyjmenovány používané metody umožňující vizualizaci těchto dat uživateli.

Další kapitoly se již zabývají praktickou částí naší práce. Od fáze návrhu, kde přiblížíme princip Marching cubes a rozhraní vyvíjené knihovny, se dostáváme k praktické realizaci, která objasňuje rozdílné kroky, jež bylo nutné provést ve variantách pro CPU a GPU. Závěrem může čtenář shlédnout dosažené výsledky této práce a porovnat tak, zda nám volba či aplikace algoritmu na grafické zařízení přinesla výkonové výhody či spíše řadu problémů.

Kapitola 2

Technologie grafického hardwaru

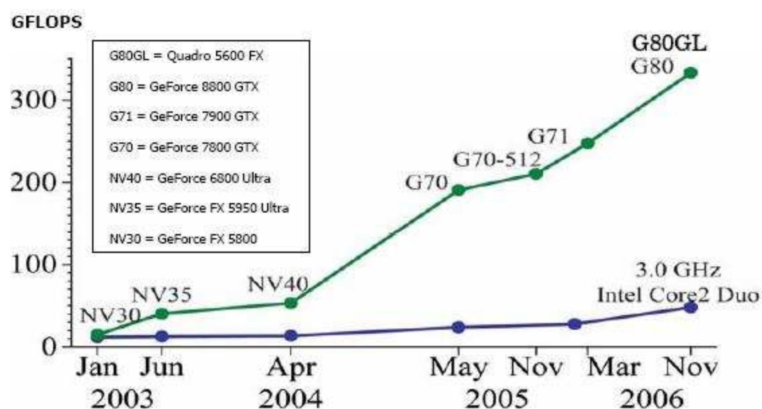
V této kapitole se budeme podrobněji seznamovat s technologiemi grafických karet, které jsou významné vzhledem k tématu obecných výpočtů na GPU.

2.1 Historie

Vývoj grafických procesorů můžeme rozdělit do několika postupných fází [8]:

1. generace (1999) – grafické adaptéry poskytovaly rasterizaci trojúhelníků, geometrické transformace ovšem musel stále provádět procesor.
   - Grafické karty: NVidia GeForce 256, ATI Radeon 7500
   - Grafické knihovny: DirectX 6, OpenGL 1.1
2. generace (2000) – na GPU přibyla specializovaná jednotka pro geometrické transformace a výpočet osvětlení (T&L unit), uživatel mohl ovlivňovat nastavení renderingu pomocí rozsáhlejší konfigurace.
   - Grafické karty: NVidia GeForce 256, ATI Radeon 7500
   - Grafické knihovny: DirectX 7, OpenGL 1.2
3. generace (2001) – GPU umožňovaly programátorský zásah do vykreslovacího řetězce, objevily se první tzv. *shader modely* (verze 1.1), které ovšem byly funkčně i velikostně značně omezené
   - Grafické karty: NVidia GeForce 3 a 4, ATI Radeon 8500
   - Grafické knihovny: DirectX 8.0/8.1, OpenGL 1.4
4. generace (2003) – tato generace poprvé využívala plně programovatelných jednotek pro zpracování vrcholů a pixelů (vertex a pixel shadery) a teprve tehdy vznikla myšlenka využití GPU i pro obecné výpočty.
   - Grafické karty: NVidia GeForce FX, ATI Radeon 9700
   - Grafické knihovny: DirectX 9.0, OpenGL 1.5/2.0
5. generace (2006) – nejnovější generace grafických karet, přišla s úplně novou architekturou unifikovaných procesorů. Výrobci zároveň dali programátorům do rukou silnou zbraň v podobě API pro obecné využití GPU.

- Grafické karty: NVidia GeForce 8, ATI R600
- Grafické knihovny: DirectX 10, OpenGL 2.1 (3.0)



Obrázek 2.1: Porovnání výkonu [5]

2.2 GPGPU

GPGPU (z angl. General-purpose computing on Graphics Processing Units) je moderní trend využívající GPU pro obecné „negrafické“ výpočetní operace namísto CPU (Central processing unit). Na obrázku 2.1 si nelze nevšimnout, jak moderní grafické karty mohutně překonávají výkon procesorů. Je ale třeba vzít na vědomí, že porovnávacím měřítkem jsou operace s desetinnými čísly. Procesory grafických karet jsou pro práci s desetinnými čísly přímo konstruovány a optimalizovány, což je nutné z důvodu přesnosti hodnot (souřadnic vrcholů, vektorů) potřebné při renderování scén.

2.2.1 Přehled

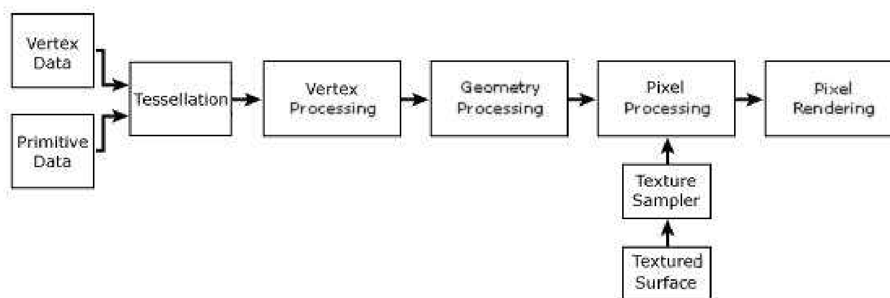
V této části si přiblížíme způsob využití programovatelných částí vykreslovacího řetězce v GPGPU.

Během roku 2001 (viz 3) došlo k malé revoluci na poli grafických technologií, po které již nějakou dobu volali sami vývojáři grafických aplikací. Do této doby byl průběh vykreslovacího řetězce GPU dán architekturou a implementací zabudovaných funkcí, analogicky pojmenovaný jako *vykreslovací řetězec pevně daných funkcí* (z angl. fixed function pipeline). Vývojář tak neměl možnost zasahovat či měnit jakoukoliv část pipeline (viz obrázek 2.2), různé efekty (např. vržených stínů apod.) musely být předpočítány na CPU. Zmíněnou revolucí je myšleno zpřístupnění částí pipeline tak, aby vývojář mohl ovlivnit její funkční logiku. Konkrétně se jedná o část zpracovávající vstupní vrcholy primitiv, tzv. *vertex shadery*, a části pracující s jednotlivými fragmenty výsledného obrazu, tzv. *pixel shadery* (někdy se také používá výraz *fragment shader*). Např. vertex shader nebyl ničím jiným, než do té doby využívanou jednotkou pro výpočet transformace geometrie a osvětlení (angl. Transform and Lighting Unit) [8].

Shadery nahrazovaly pevně danou funkcionalitu grafického řetězce programovatelnou logikou definovanou vývojářem. Nové možnosti programovatelných procesorů okamžitě zaujaly výsadní postavení v herním průmyslu, kde vzhled scény a celkový dojem ze hry poskočil

o několik kroků blíže k realitě. Je však třeba zdůraznit, že první verze shader modelu byly velice omezené, instrukční sada čítala jen několik málo instrukcí a limitován byl i počet instrukcí ve zdrojovém kódu shader programů (to platilo především pro pixel shadery), ovšem s každou vyšší verzí se určitá omezení postupně odstraňovala (viz [8]).

Na hardwarové úrovni lze považovat vertex i pixel procesory za nezávislé výpočetní jednotky. Každá má svoji instrukční sadu (s postupným vývojem se čím dál více sjednocují), mají přístup (přímý, nepřímý) do určitých částí paměti, ze které mohou číst i do ní zapisovat. Funkce, která zastupuje postup zpracování v shaderu, nazýváme tzv. *kernel* (z angl. jádro). Jelikož máme možnost číst data z paměti, provést s nimi určité operace a výsledky zapisovat zpět do paměti, začalo čím dál více vývojářů využívat GPU pro obecné výpočetní operace. Tak vznikl nový směr využití schopností GPU, které se mohli pyšnit propracovaným systémem paralelního zpracování a rychlého přístupu do lokální paměti. Techniky pro „negrafické“ výpočty ovšem byly většinou dost krkolomné, právě kvůli architektuře pipeline určené k renderování grafického výstupu [8]. V kapitole 2.4 o nejnovější řadě grafických karet GeForce 8 společnosti NVidia nastíníme, jaký má příchod moderních GPU vliv na vnitřní architekturu GPU a její zvýhodnění při použití pro obecné výpočty.



Obrázek 2.2: Klasická OpenGL vykreslovací pipeline

Krátce zmíníme i programovací jazyky určené k implementaci shaderů. Jak je uvedeno výše, první shader modely měly velmi malé množství instrukcí. Kód shaderu se psal pouze v jazyce assembler, chyběla podpora specializovaných programovacích jazyků vyšší úrovně. Vývoj programovatelných jednotek GPU znamenal velký úspěch na poli počítačové grafiky a tak, jak již bylo uvedeno, novější verze shader modelů na sebe nenechaly dlouho čekat. S novou modelovou verzí se vždy jednalo o rozšíření instrukčních sad procesorů a tím také rostla nutnost vytvoření vyšších programovací jazyků určených pro psaní shaderů, které by usnadnily jejich tvorbu. Společnost Microsoft a výrobce grafických karet NVidia proto začaly spolupracovat na vývoji jazyka, který je ve své podstatě velice blízký jazyku C. Microsoft jej vydal pod názvem *HLSL* (zkr. High Level Shader Language) jako nativní programovací jazyk shaderů pro svůj DirectX, NVidia jazyk s drobnými úpravami vydala pod označením *Cg* (zkr. C for graphics). Díky předchozí spolupráci je tak HLSL z více než 90 % shodný s *Cg* od firmy NVidia. Cg však není vázaný na žádnou grafickou knihovnu a lze jej tedy použít jak v OpenGL tak v DirectX, což může být v určitých situacích výhodné. Také ARB (zkr. Architecture Review Board), která spravuje specifikace grafické knihovny OpenGL, vydala svůj jazyk vyšší úrovně pojmenovaný jako *GLSL* (zkr. OpenGL Shading Language), který překvapil některými svými inovativními prvky. Např. možností použít celočíselný datový typ, který před příchodem DirectX 10 nebyl na GPU skutečně implementován. Zmíněný pojem DirectX 10 pak přinesl do počítačové grafiky další a zároveň

nemalý posun kupředu. I z toho důvodu se jím budeme zabývat v následující kapitole.

Příklady oblastí, kde se GPGPU využívá [10]:

- počítačové clustery jako např. HPC (superpočítače) nebo Grid (distribuované výpočty)
- simulace v oblasti fyziky (dynamika kapalin, simulace pohybu šatů, atd.)
- zobrazování objemových dat
- rychlá Fourierova transformace
- mapování a zobrazení zvuku
- zpracování audio signálu (digitální, analogový)
- segmentace 2D a 3D obrazu
- zpracování videa (hardwarová akcelerace kódování a dekódování)
- raytracing
- globální osvětlení (radiozita, photon mapping)
- detekce kolizí a počítání geometrie
- vědecké výpočty (předpověď počasí, výzkum klimatu a oteplování, molekulární modelování, kvantová či mechanická fyzika)
- bioinformatika
- počítačové vidění
- neuronové sítě
- databáze (řazení a vyhledávání)
- kryptografie

2.3 DirectX 10

Vývoj rozhraní *DirectX 10* (dále DX10) začal přibližně v roce 2002 (jedná se o uzavřený systém, tak to nelze říci přesněji) a u jeho vzniku stály vedle společnosti Microsoft také přední firmy zabývající se vývojem grafických čipů (NVIDIA, ATI, Intel). Důležitou roli v procesu vzniku hráli také vývojáři 3D aplikací (téměř výhradně počítačových her), jejichž časté požadavky na chybějící funkcionalitu moderního grafického HW tvoří podstatnou část novinek zavedených v nově definovaném API. První beta specifikace DX10 byla představena v roce 2005.

2.3.1 Cíle

Cílem vývoje nového Direct3D API bývalo v předchozích verzích zpřístupnění nové funkcionality nejmodernějších GPU na trhu. V případě Direct3D 10 byly při návrhu brány v potaz také další faktory:

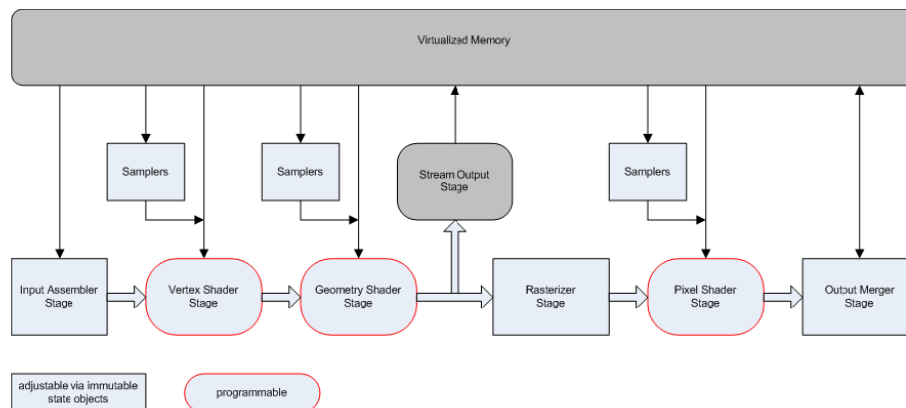
- jednotné API z hlediska používání – zde hrály významnou roli herní konzole, kde existuje unifikovaný hardware napříč všemi uživateli
- zlepšení komunikace HW a SW – požadavek vycházel především od výrobců GPU a autorů systémových ovladačů
- nová funkcionalita – požadavek od vývojářů her, kteří již léta volali po schopnostech GPU, které dosud nebyly k dispozici
- využití hrubého výpočetního výkonu GPU – aktuální rychlý vývoj GPU byl motivací k návrhu nového 3D API jako obecnějšího nástroje pro práci s grafickým hardwarem

2.3.2 Architektura

Rozhraní DX10 nedefinuje pouze přístup k funkcionalitě GPU, ale také definuje přístup k operačnímu systému (resp. prostředí), na kterém běží. Nové DX10 je mnohem více svázané se systémem a vzhledem ke specifickým požadavkům vyžaduje OS typu MS Windows Vista (nebo pozdější). Nutnost použití tohoto systému vyplývá především z tzv. virtualizace paměti na GPU právě skrze operační systém, který umí spravovat paměť nejen typu RAM, ale také paměť spravovanou GPU a jeho ovladači. Ta bývá obecně rozdělena na paměť lokální (umístěná přímo na grafické kartě) a nelokální (jedná se o část paměti RAM adresovatelnou pomocí GPU).

Z hlediska architektury došlo v DX10 oproti klasickému OpenGL vykreslovacímu řetězci k několika významným změnám. Tou nejdůležitější je především tzv. *Geometry Shader* (GS), k němu připojený *Stream Output* (SO) a dále propojení programovatelných jednotek s virtualizovanou pamětí bez nutnosti používat pro načítaní dat texturovací jednotky. Tím je umožněno načítat do programovatelných jednotek data téměř libovolného typu. Další významné změny jsou např.:

- odstranění fixního řetězce – v DX10 je všechna tato činnost ponechána na programátorovi, který ji definuje v shaderech.
- odstraněny bity schopností (CAPabilitieS bity) – pokud chce dané GPU splňovat specifikaci DX10, pak musí splňovat všechny parametry.
- neměnné stavové objekty – DX10 definuje snadné nastavení celé grafické pipeline pomocí pouhých pěti stavových objektů. Každý z nich obsahuje různá nastavení pro některou část grafické pipeline.
- *Shader Model 4.0* a *HLSL 4.0* přímo v DX10 – dřívější verze DirectX obsahovaly jazyk HLSL pouze v pomocné knihovně D3DX. Ve verzi DX10 je HLSL přímo zabudováno. Obsahuje plnou podporu pro Shader Model 4.0 včetně unifikace shader programů (tj. podpory všech instrukcí v libovolném typu programovatelné jednotky) a podpory *geometry shaderu*.



Obrázek 2.3: Direct3D 10 pipeline

2.3.3 Vykreslovací řetězec

Pipeline tvoří uzavřený tok dat proudících přes virtualizovanou paměť, což je hlavní rozdíl oproti předchozím verzím D3D. Na obrázku 2.3 jsou znázorněny programovatelné jednotky a také jednotky, které lze nastavit pomocí stavových objektů.

Jednotlivé části:

- Input Assembler – jedná se o vstupní část celého řetězce. Hlavním úkolem této části je vytvářet z paměťových bufferů vertexy a posílat je do řetězce.
- Vertex Shader – aplikuje zadaný shader program na všechny vstupní vertexy. Má přímý nebo nepřímý (přes textury) přístup do virtualizované paměti, což umožňuje téměř libovolnou práci s daty.
- Geometry Shader – úkolem je zpracovávat celé geometrické útvary (trojúhelníky, úsečky, body) složené z vertexů již transformovaných předchozím vertex shaderem. Jednotka obdrží geometrický útvar vždy jako pole vertexů a může tedy provádět např. takové operace jako je výpočet normály trojúhelníku nebo výpočet délky úsečky. Stejně tak je možné příchozí vertexy rušit nebo dokonce nové přidávat.
- Stream Output – jediným úkolem je zápis dat přímo do zadaného paměťového bufferu(ů). Použitím této jednotky můžeme snadno zapsat zpracovaná data do bufferu a následně je znovu vložit na vstup procesorů pro další zpracování.
- Rasterizátor – v DX10 téměř identický s běžným rasterizátorem předchozího D3D (tedy převádí primitiva na posloupnost pixelů). Další důležitou novinkou rasterizátoru je možnost jeho úplného vypnutí. V takovém případě je celý zbytek grafické pipeline deaktivován, což na HW s unifikovanými shadery (který se dnes používá) znamená, že všechen výpočetní výkon může být soustředěn na předchozí části pipeline (IA,VS,GS,SO). Tohoto chování se využívá především při použití GPU jako obecného výpočetního zařízení (GPGPU).
- Pixel Shader – jednotka, která doznala nejmenšího počtu změn oproti předchozí verzi D3D. Aplikuje příslušný shaderový program na příchozí pixely.

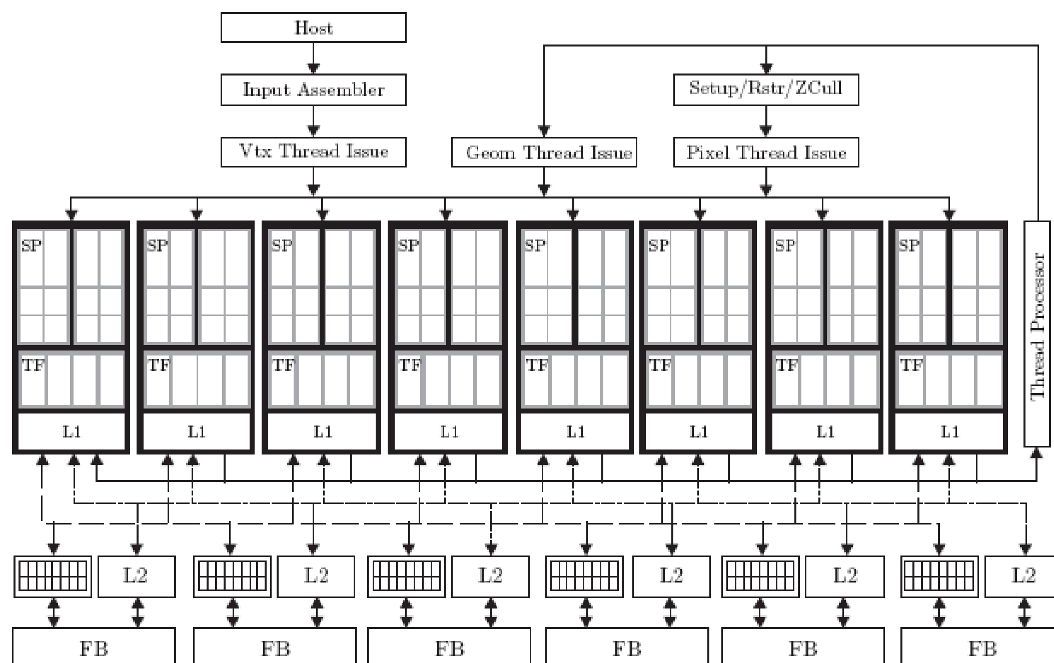
- Output Merger – jednotka nastavitelná pomocí dvou stavových objektů. Základním úkolem je práce s pixely, provádění per-pixel testů a jejich zápis do výstupních bufferů (render-target buffer). Mezi nejpoužívanější testy patří test hloubky (depth test) a stencil test.

Více informací o této grafické knihovně lze nalézt v [4].

2.4 NVidia GeForce 8

Společnost NVidia v roce 2006 přišla na trh s novou převratnou sérií grafických karet – osmou řadou GeForce (přesněji s čipem G80). Až do této doby zpracovávaly všechny GPU osobních počítačů data víceméně shodným způsobem – „sekvenčně“, tedy seriovým spojením několika částí grafické pipeline. I přes skutečnost, že programovatelné shadery daly vývojářům velkou volnost a nové možnosti (zmíněno v kapitole 2.2), stále platila specifikace pevných kroků vykreslovacího řetězce. HW implementace původní pipeline (viz obrázek 2.2) obsahující programovatelné jednotky (vertex a pixel shadery) předchozích generací grafických karet byla nahrazena flexibilnější architekturou (viz blokové schéma GeForce 8 2.3) poskládanou výhradně z tzv. *unifikovaných stream procesorů* [8].

2.4.1 Architektura



Obrázek 2.4: Blokové schéma GeForce 8800 [8]

Z hlediska vývoje čipu GPU je zcela jistě jednodušší mít více typů specializovaných procesorů než jeden komplexní procesor. Nicméně z pohledu výkonu architektury je výhodnější vytvořit procesory v GPU jako komplexní jednotky, které jsou schopny provádět libovolné operace, což umožňuje maximalizovat vytížení celého GPU dynamickým přidělováním jednotlivých výpočetních jednotek v GPU tam, kde je právě potřeba. To byl právě problém

předchozí generace grafických čipů, kdy některé části GPU byly přetížené a jiné naopak málo využívané. Grafické karty obsahovaly pevně daný počet vertex procesorů a pixel procesorů, tedy pokud se např. zpracovávaly vrcholy ve vertex procesoru, schopností pixel procesorů se k urychlení výkonu v daný moment nevyužilo, zapojily se do výpočtu až na ně v pipeline přišla řada. Naopak unifikované stream procesory poskytují svůj výpočetní výkon kdekoliv je zrovna třeba, ať už se jedná o vertex, pixel či nový geometry shader nebo pro obecné výpočty. Navíc tyto moderní karty obsahují velice komplexní vnitřní řízení zdrojů a přidělování stream procesorů potřebným částem pipeline. Např. pokud nastane situace, že pixel shader čeká na data z texturovací jednotky a jeho běh je pozastaven, mohou být mezitím prostředky přiděleny jiné části, např. vertex shaderu [8].

Se stále větší oblibou aplikace shaderů na obecné výpočty, se zvyšoval i počet algoritmů, které častěji prováděly operace se skalárními hodnotami namísto vektorových. Doménou stream procesorů v architektuře G80 je tedy práce se skalárními hodnotami, přičemž vektorové operace lze na paralelní procesory jednoduše namapovat. Důvod, který vedl architekty k tomuto kroku, bylo především zmenšení komplexnosti jednotek, ovšem při zachování flexibility a univerzálnosti. Stream procesory používají stejnou architekturu zpracování jako předchozí shadery – *SIMD* (z angl. Simple Instruction, Multiple Data), které dokáží v jednom cyklu multiprocesoru vykonat jednu instrukci na několika procesorech obsahující různé data [8].

Více informací o grafických kartách řady GeForce 8xxx lze nalézt v [6].

2.5 Technologie CUDA

S příchodem nové architektury hardware se na scéně objevila další technologie, řekněme z pohledu obecného využití GPU doslova revoluční a vývojáři dlouho očekávaná, která dostala název *CUDA* (z angl. Computed Unified Device Architecture).

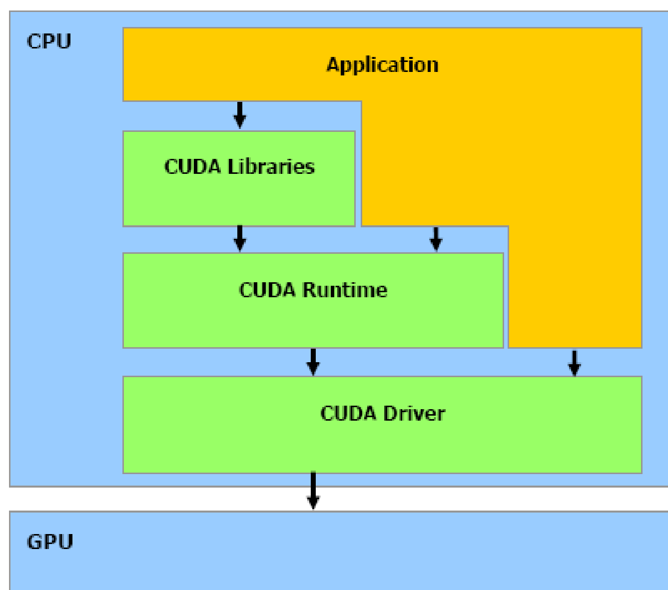
Jak jsme již uvedli v kapitole o GeForce 8 (2.4), předchozí grafické karty měly pevně danou vykreslovací pipeline. Jedinou možností, jak programátorsky ovlivnit způsob zpracování v takové pipeline, tedy jsou shadery, jejichž pomocí lze měnit pouze dvě části pipeline (vertex a pixel shader). Programátor tedy musel znát grafické API, jehož pomocí mohl přenášet obecná data na GPU v podobě dvou nebo třídimenzionálních textur (navíc omezeny datovým typem). V těle shaderu jsme s daty vykonali požadovanou operaci a výsledek se „vykreslil“ do frame bufferu [1], ze kterého se výsledná data mohla číst. Tato technika se nám může zdát nesmyslná v porovnání s technologií CUDA, která svým vydáním způsobila již zmíněnou revoluci ve využití moderních grafických karet. Je třeba uvést, že technologie je dostupná pouze na moderních produktech výrobce NVidia [8]. Co se týče hardwarové podpory, CUDA je dostupná na grafických kartách řady GeForce 8, určených výhradně pro hry, dále v sérii karet Quadro zaměřených pro profesionální grafiku (CAD apod.) a také ve výpočetních systémech Tesla, které se využívají v oblasti náročných specifických výpočetních úloh (HPC [2]).

V několika následujících podkapitolách se s touto technologií seznámíme podrobněji (čerpáno z [5, 8]).

[1] buffer, jehož obsah se zobrazí na monitoru.
[2] High Performance Computing

2.5.1 Architektura

CUDA je zcela nová hardwarová a softwarová architektura pro obecné výpočty na GPU. Využívá GPU jako obecný paralelní výpočetní prostředek, ovšem s tím rozdílem, že již není třeba znát grafické API stejně jako principy vykreslovací pipeline. Kompletní zastoupení této technologie v HW a SW je rozdělena do několika vrstev. Podpora je již implementována v samotném hardwaru a také ve specializovaných ovladačích grafické karty, bez kterých nelze CUDA aplikaci i na podporované grafické kartě spustit. Z pohledu vývojáře jsou významné runtime knihovny a na nejvyšší úrovni položené knihovny API. Závislost a umístění v jednotlivých vrstvách demostruje obrázek 2.5.

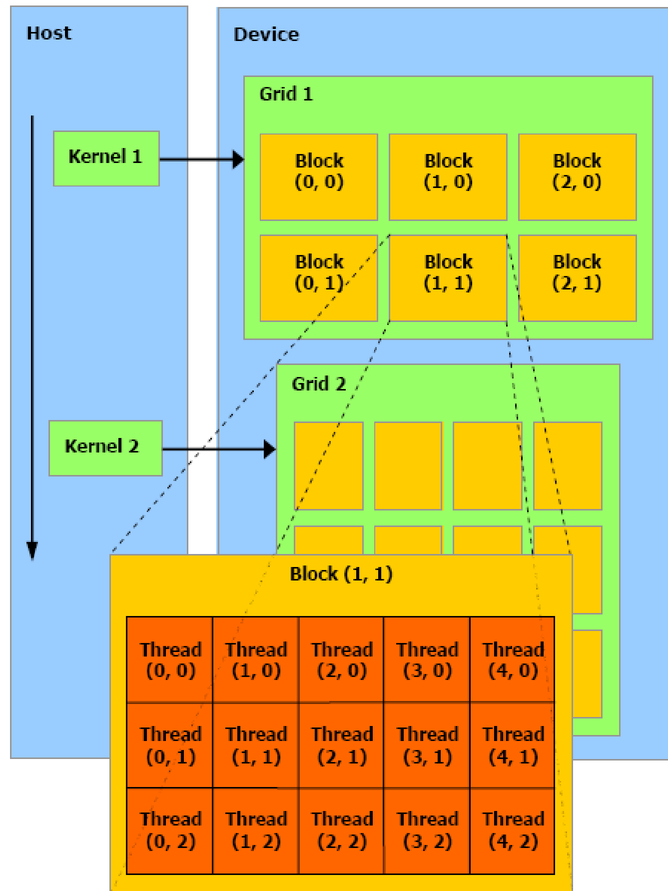


Obrázek 2.5: Vrstvy programového vybavení CUDA [5]

2.5.2 Programovací model

Základem programovacího modelu architektury je chápání GPU jako výpočetního *zařízení* (*device*) schopného paralelně provádět velké množství *vláken* (*threads*). Vzhledem k hlavnímu procesoru CPU, který je označován za *hostitele* (*host*) řídícího celý proces, plní GPU funkci koprocesoru. Posloupnost instrukcí, které chceme vykonávat na grafickém procesoru a využít tak vícevláknového zpracování, kompilujeme do podoby tzv. *jader* (*kernels*). Skupina vláken, která je spouštěna v rámci jednoho jádra, je rozdělena do *bloků* (*blocks*). Bloky nabízí možnost sdílení paměti (*shared memory*) mezi vlákny, které danému bloku patří, což u shaderů nebylo možné. Navíc nám tento způsob organizace umožňuje efektivní komunikaci mezi vlákny či jejich synchronizaci v podobě synchronizačních bodů uvedených v kódu jádra. Dokud všechna vlákna nedosáhnou synchronizačního bodu, jsou ostatní pozastavena. Každé vlákno má svůj identifikátor (*thread ID*), podle kterého jej můžeme jednoznačně adresovat. Jelikož adresace může být v různých případech velice složitá a nepřehledná, lze bloky specifikovat i jako dvou či třídimenzionální pole. Skutečný identifikátor vlákna se pak vypočítává z jednotlivých složek.

V současnosti je blok limitován maximálním počtem vláken (512), které může obsahovat. Z toho důvodu lze bloky (stejných velikostí a dimenzí, a zároveň patřící do jednoho jádra) sdružovat do *mřížek* (*gridů*). Avšak bloky, resp. vlákna v blocích, mezi sebou nemohou komunikovat, ani provádět synchronizaci. Stejně jako v případě vláken, adresování bloků probíhá také pomocí jednoznačného identifikátoru (*block ID*), ze stejného důvodu jako v případě bloků lze mřížku specifikovat i jako dvoudimenzionální. Kompletní strukturu popsanou v této části lépe ilustruje obrázek 2.6.



Obrázek 2.6: Struktura programovacího modelu [5]

2.5.3 HW implementace

Zařízení je implementováno jako množina multiprocesorů a chová se jako paralelní SIMD zařízení (viz také v kapitole o GeForce 8 2.4). Každý multiprocesor obsahuje čtyři typy pamětí:

- na každém procesoru je sada 32-bitových registrů,
- sdílená paměť pro všechny procesory multiprocesoru,
- cache pro konstantní paměť zařízení (pouze pro čtení),
- cache pro paměť textury (pouze pro čtení).

Jeden nebo více bloků je namapováno na jeden multiprocesor. Platí i další podmínka, kdy jeden blok je vždy zpracován na jednom multiprocesoru, což umožňuje sdílet paměťový prostor implementovaný na čipu a zároveň zajišťuje rychlou a efektivní komunikaci na lokální úrovni.

2.5.4 Paměťový model

V části programovacího modelu jsme si uvedli jako nejnižší element vlákno, proto je třeba objasnit, jak může vlákno pracovat s pamětí. Vlákno vykonávané v rámci jádra má přístup pouze do lokální paměti čipu nebo do videopaměti GPU. V seznamu uvedeme výčet operací, které může vlákno provádět s pamětí:

- čtení/zápis do registrů vlákna,
- čtení/zápis do lokální paměti vlákna,
- čtení/zápis do sdílené paměti bloku,
- čtení/zápis do globální paměti gridu,
- čtení z konstantní paměti gridu,
- čtení z paměti textur gridu.

Zároveň je třeba zmínit, že z (do) globální, konstantní a texturové paměti může hostitel číst (zapisovat).

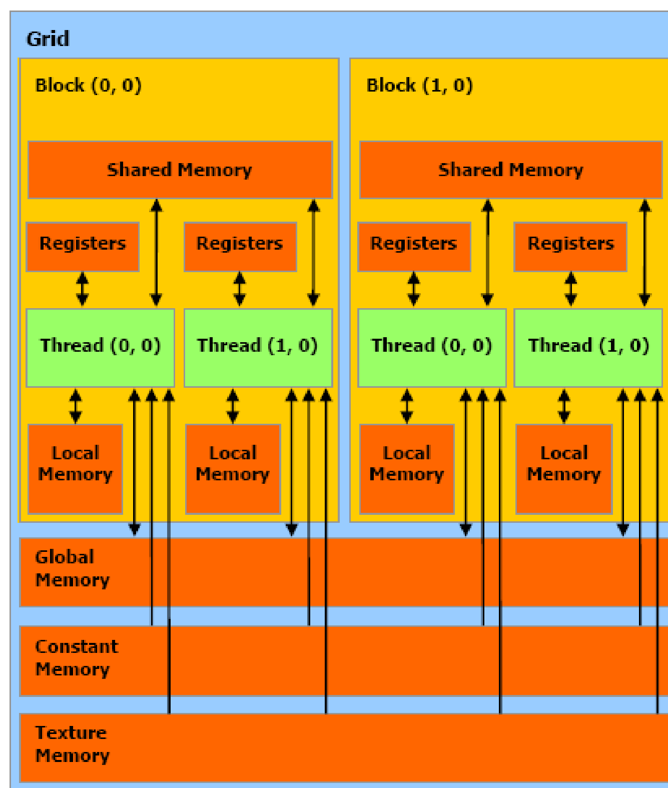
2.5.5 Aplikační programové rozhraní

Programové rozhraní CUDA poskytuje relativně snadnou cestu programátorům, kteří mají jisté zkušenosti s programovacím jazykem C. CUDA rozšiřuje jazyk C o následující konstrukce:

- funkční kvalifikátory – rozlišují funkce, které jsou vykonávány na hostiteli nebo zařízení, nebo zda je lze volat z obou míst
- typové kvalifikátory – specifikují umístění proměnných v konkrétní části paměti zařízení
- direktivy – definují, jakým způsobem se jádro na zařízení spustí
- čtyři vestavěné proměnné – dimenze gridu a bloku, indexy bloků a vláken

Samotná runtime knihovna je rozdělena do tří částí podle toho, jestli jsou funkce spouštěné na hostiteli, na zařízení nebo zda mohou být spouštěné na obou místech.

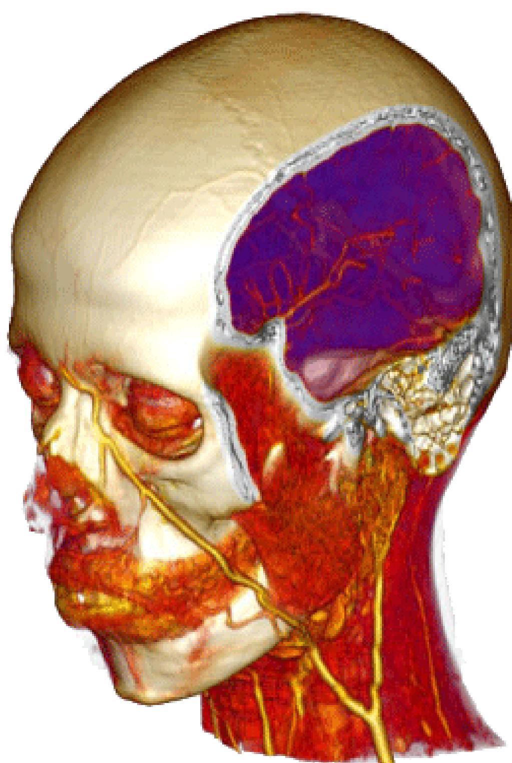
Každý zdrojový kód, který obsahuje výše uvedená rozšíření jazyka C, musí být přeložen do binární podoby pomocí speciálního kompilátoru *NVCC*, jehož hlavní funkcí je oddělit kód určený pro zařízení. Je možné používat na zařízení pouze ty prostředky standardní knihovny jazyka C, které jsou podporovány runtime knihovnou.



Obrázek 2.7: Struktura paměťového modelu [5]

Kapitola 3

Zobrazování volumetrických dat



Obrázek 3.1: Ukázka vizualizace volumetrických dat v medicíně

Volumetrická nebo-li objemová data a jejich požadovaná grafická reprezentace jsou velice mocným nástrojem pro vizualizaci prostorových objektů, aniž bychom znali jejich polygonální strukturu. Svět moderní medicíny by bez nich byl velice omezující a nenabízel by takové možnosti jako s nimi. Volumetrická data reprezentují povrchy nebo objemové struktury, umožňují tak transformovat skutečné objekty do digitální podoby. Bohužel zisk těchto dat v praxi není zcela jednoduchou záležitostí, odpovídá tomu komplexnost a také vysoké pořizovací ceny různých typů snímacích zařízení. V dnešní době patří v medicíně mezi nejznámější a nejrozšířenější *CT* (z angl. Computed Tomography) a *MRI* (z angl. Magnetic Resonance Imaging).

Volumetrická data ve většině případů představují sadu dvourozměrných obrázků (tzv. ře-

zů), mezi kterými je velice malá vzdálenost, výsledkem je tedy trojrozměrná prostorová mřížka. Jelikož prostorové objekty nejsou reprezentovány vrcholy, nelze využít klasický vykreslovací řetězec grafických karet a tedy je třeba zvolit vhodné a výpočetně přijatelné algoritmy s ohledem na kvalitu výstupu. Takových zobrazovacích metod je známo několik, ovšem všechny jsou v jistém smyslu závislé na schopnostech a výpočetním výkonu hardwaru. Tato skutečnost byla po dlouhou dobu velice limitující. Pokud bylo třeba dosahovat velice kvalitních výsledků, bylo zapotřebí zapojit sílů superpočítačů, které byly a jsou taktéž velice nákladné. V ostatních případech se musí přistupovat k problematice zobrazování s jistým kompromisem mezi vykreslovací rychlostí a mírou kvality zobrazení výsledku, ovšem s nástupem moderního grafického hardwaru a technologií GPGPU se tato problematika posunula významným krokem kupředu. Zároveň musíme být obezřetní ke skutečnosti, že volumetrická data jsou velice náročná na velikost potřebné paměti. Grafické karty se sice v tomto směru stále vyvíjejí, ale toto potenciální omezení nelze při volbě zobrazovací metody zcela ignorovat. Obecně lze data do paměti grafické karty ukládat dvěma způsoby, buď jako 2D textury, které ovšem mají vyšší nároky na velikost paměti, nebo v podobě 3D textury, které jsou v moderních grafických kartách běžně podporovány.

3.1 Zobrazovací metody

Zobrazovacích metod je známo několik, v praxi z nich volíme s ohledem na hlavní požadavky výsledné vizualizace. S ohledem na rychlý vývoj technik GPGPU a grafických karet, jejichž paralelizace se účinně využívá, se ve většině případů volí přímé zobrazovací metody. Rozhodujícím faktorem je také vyšší kvalita dosažených výsledků.

3.1.1 Metody hledající povrch

Tato skupina algoritmů nezajišťuje přímé zobrazení objemových dat. Jsou určitým předstupněm jejich zobrazení, protože techniky představují možnost, jak rekonstruovat objekty geometrickými primitivy a polygonální sítí. Jsou tak vhodné i pro následnou archivaci dat v 3D podobě.

- *Contour connecting* – mezi jednotlivými řezy se určí segmenty vyznačením jejich hranic. Tyto související segmenty se poté spojí povrchy polygonů.
- *Opaque cubes* – voxely (element volumetrických dat) jsou chápány jako plné kostičky. Po nalezení hranice objektu se daný obrys vyplňuje těmito kostičkami. Získanou „kostkovitou“ strukturu ovšem nelze vhodně vyhladit, proto se tato metoda používá pro rychlé zobrazení dat, ovšem za ceny nízké kvality.
- *Marching cubes* – jeden z nejrozšířenějších algoritmů, je podrobněji popsán v 4.3.
- *Dividing cubes* – řeší problém pomalé rasterizace obrovského množství ploch. Metoda generuje povrchové body s normálovým vektorem, ovšem velikost těchto bodů je tak malá, že každý z nich odpovídá velikosti obrazového bodu. Díky normálám lze snadno počítat stínování při pohybu objektem, naopak nevýhodou je nemožnost příblížení tělesa.

Více lze nalézt v [1].

3.1.2 Přímé zobrazovací metody

Narozdíl od předchozích technik zobrazování objemových dat odpadá při přímém zobrazení nutnost převodu dat do povrchové reprezentace, skalární data se tedy zobrazují přímo. S velkou oblibou se při takovém způsobu zobrazování využívá shader programů, které jsou díky HW architektuře procesorů schopny zpracovávat pixely paralelně (více v [12]).

- *Volume Ray Casting* – princip spočívá ve vysílání světelných paprsků z pozice pozorovatele skrz body průmětny. Paprsek postupně projde objemovou strukturou, kde zjišťuje hodnoty jednotlivých vrstev objektu a pomocí přenosové funkce z nich akumuje hodnotu výsledného bodu průmětny (pixelu obrazu).

- *Splatting* – technika vykreslování spočívá ve vykreslení velmi malých plošek, které jsou natočené kolmo k pozorovateli. Barva a průsvitnost se diametrálně mění v Gaussově rozložení.

- *Shear warp* – spočívá v transformaci pohledu tak, aby se jednotlivé vrstvy zarovnaly s osou. Každá takto získaná vrstva má jednotný poměr velikosti promítaného bodu a voxelu. Jakmile máme takto nastavené vrstvy, můžeme postupně renderovat do bufferu, který ve výsledku zdeformujeme do roviny pohledu.

- *Texture mapping* – tento způsob využívá schopnosti velice rychlé práce grafické karty s texturami. Principem metody je transformace vrstev 3D textury do nových zobrazovacích vrstev, které jsou kolmé k pozorovateli. Vrstvy procházejí původní 3D texturou a pomocí interpolace se zjišťují hodnoty texelů nových vrstev, které jsou představovány 2D texturami. Vykresleny jsou pak v pořadí od nejvzdálenější k nejbližší.

Kapitola 4

Návrh

V této kapitole nejdříve uvedeme zvolenou praktickou část této práce, následně si rozebereme kroky postupné implementace.

4.1 Motivace a cíle

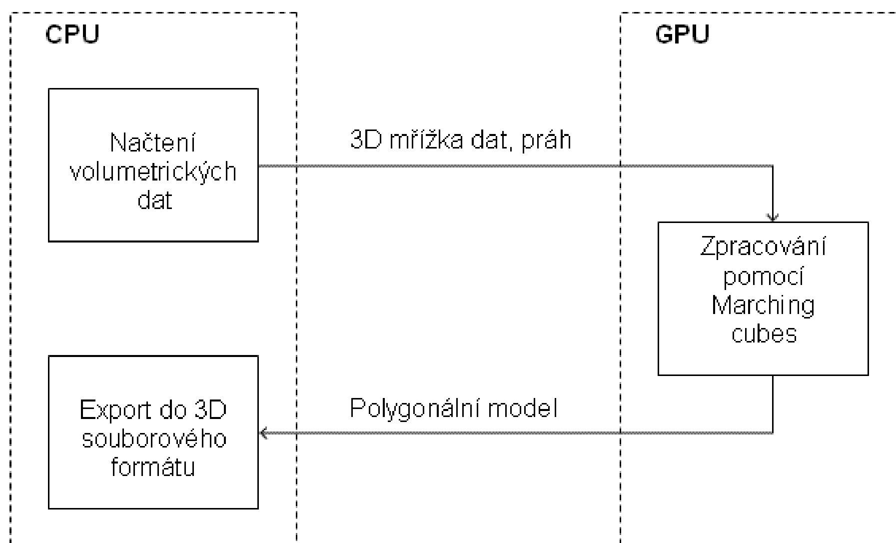
Motivací této práce byla přímá konfrontace současných CPU a GPU a jejich srovnání ve výpočetně náročných oblastech. Pro implementaci na straně GPU jsme zvolili nejnovější technologii CUDA (viz kapitola 2.5) a její aplikační programové rozhraní s použitím nejmodernějšího grafického hardwaru GeForce 8. Požadovaným výsledkem bude demonstrační aplikace využívající GPU a následné srovnání výkonu mezi CPU a GPU implementacemi zvoleného algoritmu. Volba algoritmu musí splňovat určitá kritéria, abychom co nejvíce využili paralelizmu procesorů grafických karet, ale také musí respektovat omezení, kterými se GPU vyznačují (např. velikostí paměti).

Srovnávacím algoritmem je metoda Marching cubes, která provádí vektorizaci volumetrických dat do podoby polygonálních sítí, tvořenou geometrickými primitivy. Zvolení této výpočetní metody má dva hlavní důvody. Prvním z nich je vysoká výpočetní náročnost algoritmu i pro současné CPU a z toho vyplývající požadavek na urychlení tohoto procesu. Druhý důvod vyplývá již z principu algoritmu, který pracuje s jednotlivými skupinami voxelů nezávisle na ostatních a lze jej tedy velice dobře paralelizovat. Tento výběr také podpořila myšlenka vytvořit demonstrační implementaci v podobě knihovny, kterou bude možno zakomponovat do dalších projektů. Navíc pátrání po hotovém řešení Marching cubes vytvořeném pomocí API CUDA bylo neúspěšné (GPU akcelerace algoritmu byla dostupná pouze pomocí shaderů) a tedy bylo možné říci, že výsledná implementace bude obecně prospěšná.

Kompletní knihovna bude tvořena ze tří částí (viz obrázek 4.1) – načtení volumetrických dat, zpracování algoritmem Marching cubes a exportování do 3D souborového formátu.

S myšlenkou budoucího využití definujeme v rámci knihovny pevné, neměnné části, kterými jsou veškeré výpočty prováděné na GPU a rozhraní mezi CPU a GPU. Výhoda takového přístupu je hlavně pro budoucí uživatele knihovny, protože nebude nutná žádná znalost API CUDA. Jedinou podmínkou, kterou musí uživatel splňovat, je mít grafickou kartu podporující technologii CUDA a potřebné ovladače.

Obě rozhraní mezi CPU a GPU částí knihovny jsou navržena obecně, představují jediné přístupové body ke zpracování volumetrických dat pomocí Marching cubes na GPU. Vstupní rozhraní se omezuje pouze na přenos volumetrických dat do paměti grafické karty,



Obrázek 4.1: Struktura knihovny

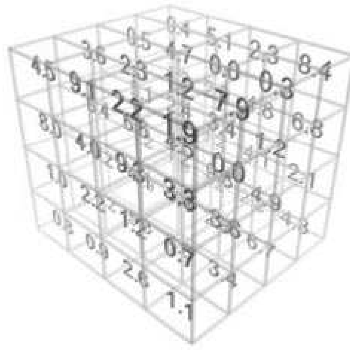
nastavení prahové hodnoty a spuštění transformačního algoritmu. Výstupní rozhraní obsahuje seznam trojúhelníků včetně jejich normálových vektorů, které tvoří výsledný polygonální model prostorového objektu. Pro demonstraci funkčnosti budou také vytvořeny části pro načtení volumetrických dat a následný export výsledného polygonálního modelu do 3D souborového formátu, které si včetně Marching cubes podrobněji přiblížíme v následujících kapitolách.

4.2 Vstupní data

Vstupní volumetrická data uvažujeme v podobě multiobrázkového souboru, kde jednotlivé obrázky představují řez objemovým tělesem. Souborových formátů pro přenos volumetrických dat existuje několik, většina aplikací pro vizualizaci objemových dat používá svůj vlastní formát. Formáty se mezi sebou liší většinou pouze hlavičkou, která obsahuje informace o dimenzích prostorové mřížky, skutečný rozměr jednoho voxelu, datový typ hodnoty voxelu a další doplňující informace.

Pro demostraci načítáme v naší implementaci „surová“ data z RAW souboru. Tento soubor obsahuje data v binární formě, v jehož hlavičce jsou uvedeny rozměry prostorové mřížky a skutečné velikosti voxelů. Rozsahy hodnot voxelů jsou voleny s ohledem na rozlišovací schopnost snímacího zařízení (nejčastěji 16-bitová celá čísla). V případě čistě obrázkových vstupních souborů bychom k výpočtu použili jasovou složku obrazu.

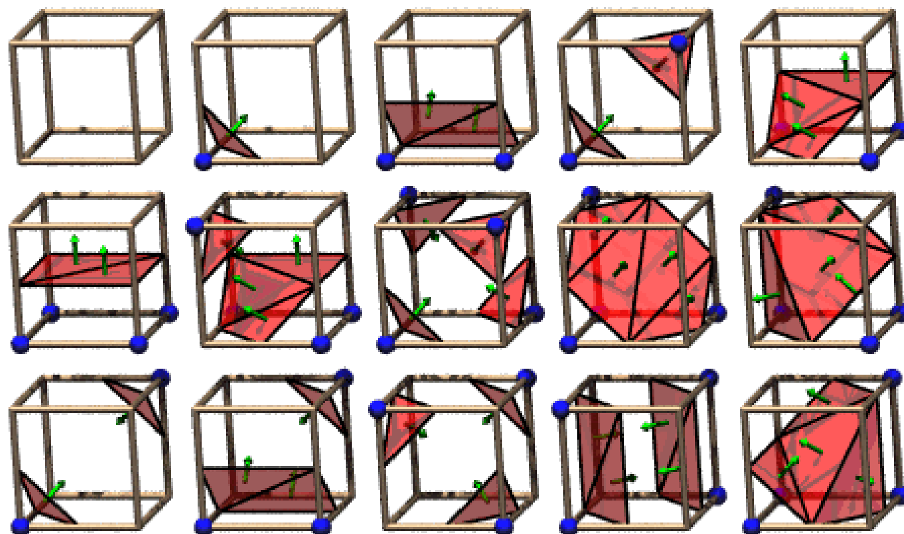
Velikost datové mřížky v paměti se pohybuje v řádu desítek megabajtů, což možnosti grafické karty NVidia GeForce 8 s velkou rezervou splňují. V krajních případech mohou paměťové nároky převýšit kapacitu grafické karty, v tom případě je třeba datovou mřížku upravit na maximální povolenou velikost podvzorkováním či oříznutím volumetrických dat.



Obrázek 4.2: Mřížka objemových dat

4.3 Marching cubes

Lorensen a Cline přišli v roce 1987 s algoritmem, jenž dokázal z volumetrických dat získaných tomografem rekonstruovat původní objekt. Algoritmus pojmenovali s ohledem na způsob výpočtu – *Marching cubes* (přeloženo jako pochodující kostky). Principem techniky je nalézt izoplochy z prostorové mřížky a pokrýt objekt sítí trojúhelníků. Výslednou síť již lze zobrazit klasickým postupem vykreslovacího řetězce grafického akcelerátoru. Přesněji řečeno, na rozdíl od zobrazovacích metod přímých, tento algoritmus je vykonán pouze jednou na objemových datech a při vizualizaci již pracujeme pouze s jejich polygonální reprezentací [1].



Obrázek 4.3: Základní konfigurace všech možných případů

4.3.1 Algoritmus

Postup algoritmu pro jednu buňku, jehož vstupem je ortogonální prostorová mřížka a prahová konstanta a výstupem síť trojúhelníků včetně jejich normálových vektorů, je uveden

v následujícím sledu kroků [1]:

1. *Sestavení vrcholů* – z datové mřížky vybereme krychli, kterou tvoří čtveřice vzorků sousedních řezů.
2. *Ohodnocení vrcholů* – hodnoty ve vrcholech jsou porovnávány se zadanou vstupní prahovou konstantou. Výsledky rozhodují, jestli jednotlivé vrcholy krychle jsou vnitřní nebo vnější z pohledu objemového tělesa. Je-li všech osm vrcholů krychle ohodnoceno shodně, dalšími kroky již algoritmus nepokračuje.
3. *Sestavení indexů do tabulky případů* – ohodnocení vrcholů dává dohromady osmibitový index do tabulky možných případů. Základní počet různých konfigurací je 15 (viz obrázek 4.3), další do celkového počtu 256 vznikly rotací či inverzí. Všeobecně se doporučuje používat kompletní tabulku konfigurací, která více eliminuje vznik děr na povrchu.
4. *Nalezení seznamu hran, které plocha protíná* – z tabulky možných případů získáme seznam hran krychle, na nichž leží vrcholy generovaných trojúhelníků.
5. *Interpolace souřadnic vrcholů trojúhelníků na hranách* – zjištěné trojúhelníky pak musíme se sousedními trojúhelníky(resp. jejich vrcholy) tzv. interpolovat. Tento proces vede k získání přesné polohy vrcholů, ovšem je závislý na použité metodě interpolace (lineární, kvadratická atd.).
6. *Výpočet normál ve vrcholech trojúhelníků* – normálové vektory ve vrcholech trojúhelníků dopočítáme interpolací, stejně jako v předchozím kroku. Tentokrát interpolujeme jednotlivé složky normál ve vrcholech krychle.

4.3.2 Charakteristika

Výhody:

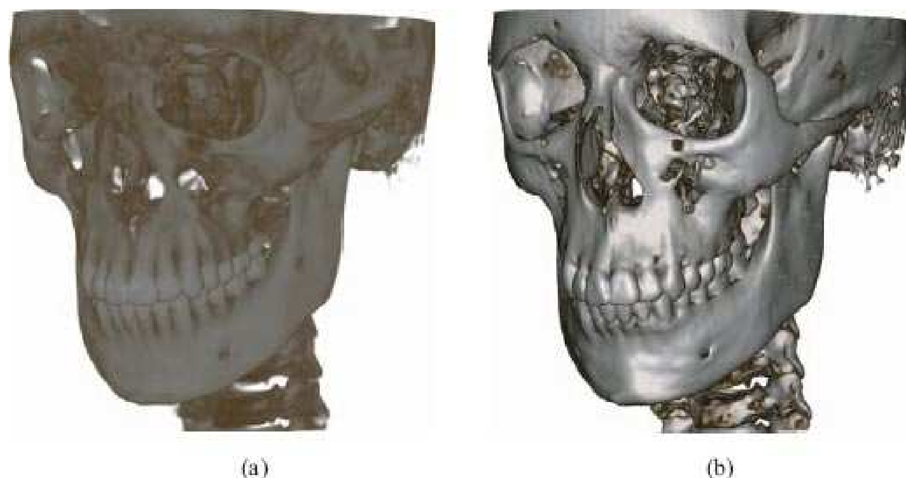
- Získáme polygonální model objektu.
- Proces transformace je proveden pouze jednou.
- Rychlé zobrazení využitím klasického vykreslovacího řetězce grafických akcelerátorů.

Nevýhody:

- Výsledkem bývá často velké množství trojúhelníků.
- Mohou nastat případy, které zapříčiňují vznik ploch obsahující díry.

4.3.3 Stínování

I když vektorizací volumetrických dat získáme obstojně identifikovatelné jednotlivé části dat, tak stínováním lze přinejmenším vylepšit celkovou vizuální kvalitu výstupu a umožnit tak detailnější představu o prostorovém rozložení dat. Cílem je hlavně zdůraznění tvarů a detailů struktury objektu. Toho je dosaženo simulací světelných efektů vzniklých dopadem světla na povrch objektu. Stínování je tedy uskutečněno pomocí světel ve scéně, kdy v každém bodě (v ideálním případě) objektu známe normálový vektor, na který aplikujeme zvolený osvětlovací model (rozdíl můžete porovnat na ilustračním obrázku 4.4). Vizualizací jsme se v naší aplikaci přímo nezabývali, ovšem museli jsme potřebné informace ke stínování poskytnout ostatním vizualizačním nástrojům.



Obrázek 4.4: a) bez použití stínování, b) použité stínování [2]

Výpočet normálového vektoru

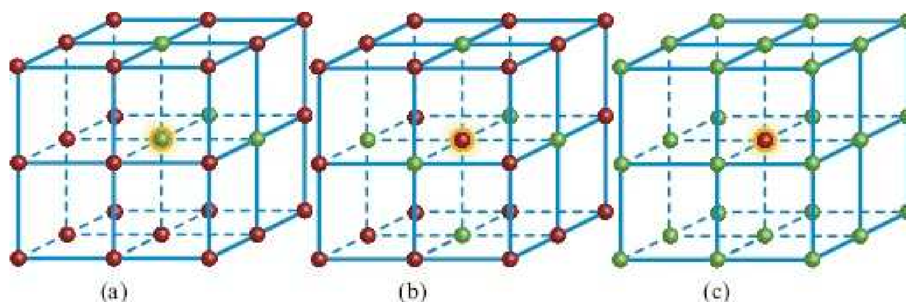
V polygonálním modelu lze jednotlivé normálové vektory vrcholů či trojúhelníků dopočítat, ovšem objemová data jsou čistě skalární hodnoty. Objemová data jsou ve většině případech získána snímáním spojitých objektů a po ukončení snímání neexistuje žádná informace o normálových vektorech. Jediným možných řešením je tedy prozkoumání okolí daného voxelu. Kvalita zobrazení objektu silně závisí na metodě provádějící výpočet normálových vektorů. Normálu pro libovolný bod objemových dat získáme výpočtem gradientu v tomto bodě. Gradientem se rozumí parciální derivace prvního řádu skalární hodnoty $I(x, y, z)$ definovaná jako

$$\nabla I = (I_x, I_y, I_z) = (\frac{\delta}{\delta x}I, \frac{\delta}{\delta y}I, \frac{\delta}{\delta z}I) \tag{4.1}$$

a velikost vektoru lze pak získat pomocí vztahu

$$||\nabla I|| = \sqrt{I_x^2 + I_y^2 + I_z^2}. \tag{4.2}$$

Nyní si představíme několik metod výpočtu gradientu, které se v praxi řadí mezi nejpoužívanější. Grafické znázornění lze vidět na obrázku 4.5.



Obrázek 4.5: a) Intermediate difference gradient, b) Central difference gradient, c) Neumann gradient. Zeleně označené voxely jsou použité k výpočtu [2].

Intermediate difference gradient přijímá jako vstup tři sousední voxely. Gradient ∇ daného voxelu V na pozici (x, y, z) se zjistí jako:

$$\begin{aligned}\nabla_x &= V_{x+1,y,z} - V_{x,y,z} \\ \nabla_y &= V_{x,y+1,z} - V_{x,y,z} \\ \nabla_z &= V_{x,y,z+1} - V_{x,y,z}\end{aligned} \tag{4.3}$$

Central difference gradient přijímá jako vstup šest voxelů. Gradient ∇ daného voxelu V na pozici (x, y, z) se zjistí jako:

$$\begin{aligned}\nabla_x &= V_{x+1,y,z} - V_{x-1,y,z} \\ \nabla_y &= V_{x,y+1,z} - V_{x,y-1,z} \\ \nabla_z &= V_{x,y,z+1} - V_{x,y,z-1}\end{aligned} \tag{4.4}$$

Neumann gradient přijímá jako vstup 26 sousedních voxelů. Obecně je tato metoda spíše teoretickým rozhraním založeném na lineární regresi. Více o této metodě je napsáno např. v [2].

Navíc se v [2] uvádí jako hlavní výhoda Intermediate difference možnost detekovat detaily o vysoké frekvenci. Ovšem celkový vizuální dojem je horší, pokud jsou vykreslována silně zašuměná data. Naopak Central difference tyto vysokofrekvenční detaily odfiltruje, ale tento operátor někdy může vynechat velmi úzké struktury, které mohou být ve výsledku podstatné.

4.4 Výstupní data

Výstupem Marching cubes je datová struktura obsahující seznam vrcholů, normálových vektorů a z nich složených trojúhelníků. Stejně jako v případě vstupního rozhraní bude i výstupní rozhraní umožňovat jedinou možnou cestu, jak získat výsledná data z grafické karty. Kvalitu výsledných dat budeme demonstrovat exportováním polygonálního modelu do 3D souborového formátu *Obj*.

Obj je textový, řádkově orientovaný formát (binární verze *mod*) pro archivaci statických polygonálních modelů. Důvodem volby formátu je hlavně jeho jednoduchost, neukládá totiž žádné (pro náš případ) nadbytečné informace jako materiály, a zároveň je vhodný pro vizuální kontrolu výsledku. Soubory obj mají též podporu CAD/CAM aplikací v podobě pluginů, čehož při práci využijeme. V následujícím seznamu si uvedeme výčet značek, které vždy uvozují řádek v souboru:

- *#* – znaky uvedené za tímto znakem jsou při čtení ignorovány, jedná se o komentář.
- *v* – definice vrcholu jeho třemi souřadnicemi v prostoru.

  ```
  v x y z
  ```

- *vt* – souřadnice textury v intervalu od 0 do 1.

  ```
  vt u v [w]
  ```

- *vn* – normálový vektor vrcholu, který některé aplikace automaticky počítají z normál okolních polygonů.

```
vn x y z
```

- *f* – definice obecného polygonu specifikovaného seznamem indexů vrcholů, příp. normálových vektorů a texturovacích souřadnic.

```
f v1[/vt1][/vn1] v2[/vt2][/vn2] v3[/vt3][/vn3]
```

- *g* – pojmenování skupiny primitiv, které následují za tímto příkazem.

```
g name
```

Výhodou formátu je jeho otevřenost, lze doplňovat vlastní značky, také máme možnost jednoduše načíst pouze požadované informace. Naopak mezi nevýhody patří větší velikost souboru v porovnání s ostatními 3D souborovými formáty a také pomalé načítání struktury modelu [11].

4.5 Možné vylepšení a modifikace

V této části uvedeme některé modifikace, které souvisí s danou tématikou. Jelikož vektorizace volumetrických dat pomocí metody Marching cubes vrací velký počet výsledných trojúhelníků, používají se techniky pro vylepšení polygonálního modelu. A to jak z pohledu kvality vizualizace, tak vysokého počtu trojúhelníků, aniž bychom se markantně vzdálili od původní geometrie modelu.

4.5.1 Vyhlazení

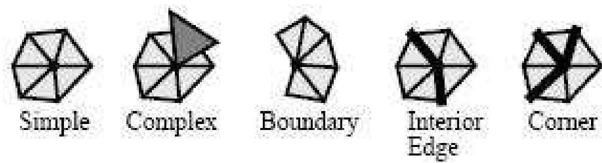
Metoda Marching cubes (obecně metody vektorizace diskrétních dat) vrací výsledné polygonální modely poněkud hranaté (vrstevnaté), z toho důvodu se používají různé metody pro vyhlazení modelu. Nejjednodušší vyhlazovací technikou je aplikace tzv. *Laplaceova operátoru*, jedná se o velice snadný iterační algoritmus. Mezi jeho největší nevýhodu patří vlastnost smršťovat vyhlazovaný objekt.

Algoritmus postupně prochází polygonální sítí a pro každý vrchol spočítá novou pozici jako průměr jeho přímých sousedních vrcholů. Aditivní metody tohoto algoritmu mohou jednotlivým vrcholům navíc definovat váhu jejich vlivu na okolní vrcholy nebo naopak míru ovlivnění původní pozice novou pozicí vrcholu.

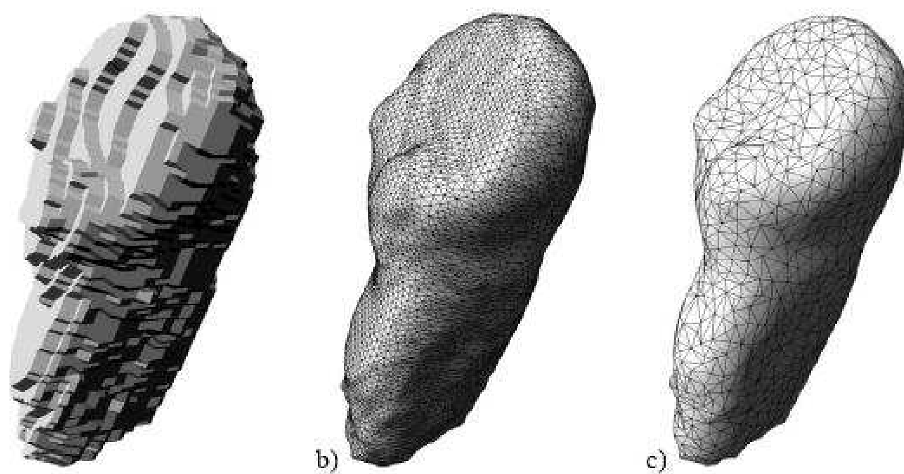
4.5.2 Decimace

Jak jsme již zmínili v úvodu této podkapitoly, algoritmus Marching cubes se potýká s problémem generování velkého počtu trojúhelníků, kdy některé modely mohou obsahovat až miliony trojúhelníků, z čehož následně vyplývá i vysoké zatížení grafické karty při vykreslování modelu. Techniky, které redukují počet trojúhelníků, se zároveň musí snažit zachovat původní geometrii objektu. Chyba se ve většině případů pohybuje okolo 10 %, což nemá zásadní vliv na kvalitu modelu.

Mezi nejznámější metody patří *Schroederův iterační algoritmus* decimace vrcholů modelu. V každé iteraci decimačního procesu je vybrán vrchol sítě a určí se jeho lokální topologie z pěti konfigurací (viz obrázek 4.6). Pokud splní decimační kritéria (dle ohodnocení), je následně (spolu se všemi přilehlými trojúhelníky) odstraněn a vzniklá díra se vyplní lokální triangulací. Tato metoda je velmi efektivní jak z časového, tak z paměťového hlediska.



Obrázek 4.6: Konfigurace lokální topologie vrcholu [7]



Obrázek 4.7: Marching cubes, vyhlazený model, decimovaný model [3]

Kapitola 5

Praktická realizace

V této chvíli již čtenář ví, co to jsou volumetrická data a v jakých situacích či oborech se jich využívá. Také získal jistou představu, co bude náplní implementační části této práce. V této kapitole postupně popíšeme vývoj knihovny pro vektorizaci diskrétních objemových dat a probereme techniky a postupy, které podpořily dosažení vytyčeného cíle. Na počátku bylo nutné stanovit si určité vývojové prostředky, které jsme po celou dobu implementace využívali.

5.1 Vývojové prostředky

Zásadním rozhodnutím byla volba vhodného programovacího jazyka. Programová část knihovny, určená pro běh na grafickém zařízení, byla postavena na současně nejmladší verzi programového rozhraní CUDA 1.1. Jelikož CUDA je rozšířením programovacího jazyka C (viz 2.5.5) a navíc jsme při návrhu necítili silnou potřebu objektově orientovaných technik programování, zvolili jsme i pro zbylé části knihovny jazyk C. Stanovením výstupních podmínek implementace knihovny, zejména nezávislost na platformně, jsme se omezili pouze na standardní knihovny jazyka C.

V závislosti na volbě programovacího jazyka jsme taktéž vybírali vývojové prostředí. Zcela jistě nejvhodnějším nástrojem je pro programování podobných aplikací Visual Studio 2005 od společnosti Microsoft a to z několika důvodů. Prostředí především disponuje velice kvalitním nástrojem pro ladění (angl. debugger) a zároveň dovoluje měnit kód aplikace přímo v režimu ladění. Ovšem největší výhoda je možnost definovat si vlastní profil překladu programu. Kompilátor NVCC (viz 2.5.5) umožňuje překlad CUDA aplikací jak do režimu pro běh na grafickém zařízení, tak i do emulovaného režimu, ve kterém lze ladit kód pro grafickou kartu. Při vývoji jsme tedy mohli jednoduše přepínat mezi jednotlivými profily a navíc kombinovat kód standardního jazyka C a CUDA, což značně urychlilo práci.

5.2 Vstupní rozhraní

Před samotným převodem objemových dat na polygonální síť je třeba příslušná data načíst a zpracovat. Ta mohou být uložena v různých formátech, které jsou charakteristické svými výhodami či naopak nevýhodami, které je vzhledem ke způsobu zpracování nutné vyřešit.

Jak jsme již uvedli v kapitole 4, naše implementace podporuje RAW formát vstupních souborů. Tento typ datového souboru jsme zvolili s ohledem na jednoduchost formátu a na dostupnost několika nasnímaných modelů v tomto datovém formátu. Data obsahují pouze

Popis	Datový typ
Velikost mřížky v ose X	znaménkové 32-bitové celé číslo
Velikost mřížky v ose Y	znaménkové 32-bitové celé číslo
Velikost mřížky v ose Z	znaménkové 32-bitové celé číslo
Minimální hodnota	bezznaménkové 16-bitové celé číslo
Maximální hodnota	bezznaménkové 16-bitové celé číslo
Mřížka objemových dat	beznaménková 16-bitová celá čísla
Skutečná vzdálenost mezi voxely v ose X	64-bitové desetinné číslo
Skutečná vzdálenost mezi voxely v ose Y	64-bitové desetinné číslo
Skutečná vzdálenost mezi voxely v ose Z	64-bitové desetinné číslo

Tabulka 5.1: Struktura RAW formátu

intenzitu jednotlivých bodů, veškerá informace o snímaném bodu je tedy uložena pouze v jedné 16-bitové složce. Tento 16-bitový rozsah je dán rozlišovací schopností snímacího zařízení, které ve většině případů disponuje 12-bitovou hloubkou. Data tedy nelze uložit bez ztráty informace do 8-bitových hodnot a proto využíváme nejbližší vyšší násobek 8.

RAW soubor s volumetrickými daty obsahuje hlavičku, následovanou mřížkou nasnímaných dat a formát je ukončen doplňujícími informacemi. Strukturu formátu zobrazuje tabulka 5.1.

V rámci definice obecného rozhraní knihovny byla vytvořena struktura, zapouzdřující vstupní objemová data. Tato struktura byla odvozena z formátu vstupního souboru a parametrů algoritmu Marching cubes. Struktura je zároveň jediný možný způsob, jak předat objemová data funkcím pro transformaci na polygonální model. Navíc její použití je napříč celou knihovnou implementované zcela obecně, lze ji rozšířit o další možné položky, aniž bychom narušili funkčnost ostatních částí knihovny.

5.2.1 Předzpracování dat

Volnost návrhu a následné realizace knihovny byla omezena několika důležitými faktory. Referenční grafické zařízení, které jsme při vývoji používali, sice disponovalo celkovou pamětí 768 MB, ovšem samotný algoritmus Marching cubes a další podpůrné algoritmy jsou velice paměťově náročné. Jak již bylo uvedeno, objemová data jsou uložena jako 16-bitová celá čísla, ale využívají pouze 12-bitový prostor. Abychom postupně co nejvíce snížily nároky na paměť, tak intenzitní hodnoty jednotlivých voxelů normalizujeme do rozsahu $\langle 0, 255 \rangle$, což odpovídá bezznaménkovému celému číslu o velikosti 1 B. Touto procedurou snížíme paměťové nároky vstupních dat bezmála na polovinu.

Největší volumetrická data, která při testování použijeme, mají rozměr mřížky $512 \times 512 \times 147$. Tento rozměr byl v rámci práce považován za referenční. Jeho použitím je zaručeno, že vektorizace se provede úspěšně se zmíněnou velikostí video paměti 768 MB.

$512 \times 512 \times 147 = 38\ 535\ 168$ voxelů
velikost datové mřížky v souboru (1 voxel × 2 B) = 77 070 336 B
velikost datové mřížky po normalizaci (1 voxel × 1 B) = 38 535 168 B
hlavička datové mřížky (rozměr, vzdálenost mezi voxely) = 24 B

---

celková velikost vstupních dat = 38 535 192 B, tj. cca 40 MB

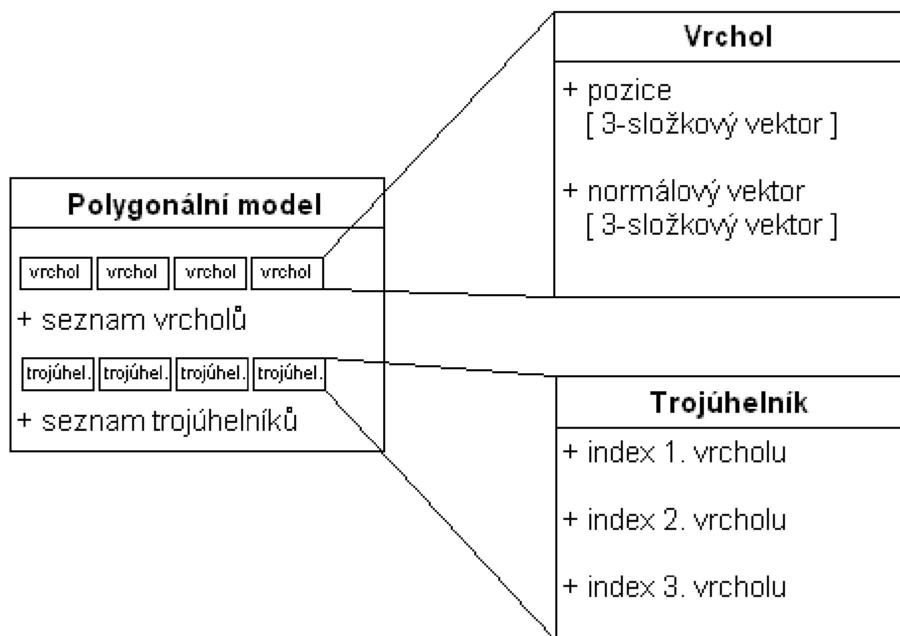
Nabízela se možnost předzpracování normálových vektorů jednotlivých voxelů, tedy vrcholů zpracovávaných krychlí. Získali bychom podstatnou časovou úsporu při výpočtech a odstranilo by se zpoždění při přístupu do paměti, zejména pak na grafické kartě. Ovšem předzpracování vektorů s sebou nese spíše více nevýhod.

Při převodu dat na polygonální síť se generují trojúhelníky, které rekonstruují povrch nasnímaného objektu. Ovšem hledaný povrch je ve většině případů reprezentován nízkým počtem krychlí vzhledem k celkovému počtu v datové mřížce, takže bychom využili pouze zlomek předpřipravených normálových vektorů.

Z pohledu několikrát zmíněného šetření paměti je předzpracování taktéž nepraktické. Normálový vektor se skládá ze tří složek, což jsou 32-bitová desetinná čísla. Pokud tedy velikost struktury vektoru (12 B) vynásobíme počtem voxelů, dostaneme se na hodnotu 463 MB, což je pro naše paměťové limity nepřípustné. Normálové vektory tedy budeme počítat v průběhu vektorizace a to jen u potřebných voxelů.

5.3 Výstupní rozhraní

Stejně jako máme definované vstupní rozhraní knihovny, bylo také nutné definovat jednotné výstupní rozhraní. Metoda, kterou používáme ke konverzi objemových dat na polygonální síť, vytváří povrch objektu reprezentovaný vrcholy a trojúhelníkovými polygony a z tohoto předpokladu jsme vycházeli i při návrhu potřebných struktur. Výstupní rozhraní (viz obrázek 5.1), představující kompletní polygonální model objektu, je složeno ze seznamů vrcholů a trojúhelníků, mezi kterými existuje určitá vazba.



Obrázek 5.1: Struktura výstupního rozhraní

Vrchol je tvořen dvěma 3-složkovými vektory, kde jsou všechny složky reprezentovány 32-bitovými desetinnými čísly. První vektor uchovává pozici vrcholu v 3D prostoru a druhý obsahuje normalizovaný normálový vektor, který využívají vizualizační nástroje k výpočtu osvětlení povrchu objektu. Vrchol je z pohledu geometrie a polygonálního modelu nezávislým

elementem, naopak trojúhelníkový polygon je definován vrcholy, tudíž je na vrcholech existenčně závislý. Struktura trojúhelníku byla navržena tak, aby neobsahovala přímo instance vrcholů, ale pouze se na ně odkazovala do seznamu. Takový přístup má výhodu v tom, že vrchol, který definuje polohu více trojúhelníkům zároveň (což je běžný případ), může být sdílen, takže v paměti bude existovat pouze jedna unikátní instance. Citelně tak snížíme paměťové nároky výsledného polygonálního modelu a navíc můžeme oddělit proces generování vrcholů a trojúhelníků.

Otázkou zůstává, jakým způsobem se budeme na dané vrcholy z trojúhelníku odkazovat. V podstatě jsme měli dvě možnosti:

- Trojúhelník bude obsahovat přímo adresu struktury vrcholu v paměti.
- Trojúhelník bude obsahovat index do seznamu vrcholů.

Obě možnosti mají společnou vlastnost a tou je velikost paměti potřebné pro uložení trojúhelníku. Ukazatel na místo v paměti (32-bitového procesoru) potřebuje 4 B, stejně jako bezznaménková celočíselná hodnota indexu (pozn. maximální délka seznamu je omezena na $2^{32} - 1 = 4\,294\,967\,295$ položek, což bylo dostačující). Nesmíme však opomenout, že proces vektorizace bude probíhat také na grafickém zařízení, které disponuje vlastní pamětí. Pokud bychom vytvořili sadu trojúhelníků, které by obsahovaly ukazatele do paměti, pak by nám při přenosu dat z grafické karty do operační paměti byly ukazatele k ničemu, protože adresa umístění dat se bude zcela jistě lišit. Indexy ovšem zůstávají stále stejné, pouze je nutné navrhnout takové řešení, které bude zajišťovat korektnost dat a jejich vazeb a také jasně specifikovat, jak získat či dopočítat index požadovaného vrcholu při vytváření trojúhelníku. Tato problematika bude objasněna v následujících kapitolách.

5.4 Vektorizace objemových dat

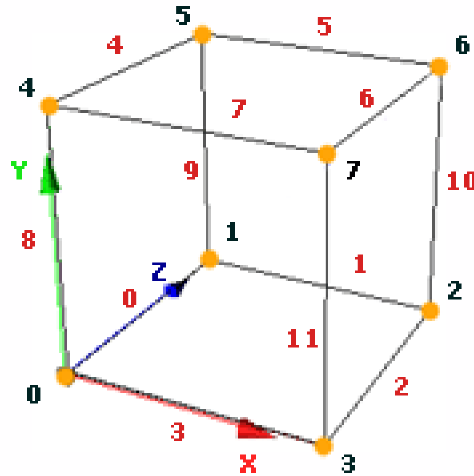
V této fázi implementace již máme připravené datové typy a funkce, které tvoří základ knihovny, a také vstupně-výstupní rozhraní. V následujících sekcích se budeme zabývat hlavní podstatou řešené problematiky – vektorizací objemových dat. Zpočátku si stanovíme, jak budeme s jednotlivými voxely pracovat a definujeme si nové pojmy, které nám usnadní popis implementačního postupu. Následně popíšeme princip vektorizace zpracovávané pomocí CPU, která byla ideálním prostředkem pro praktické seznámení s algoritmem Marching cubes. Na závěr této kapitoly se čtenář dozví, v čem se liší způsob zpracování na grafickém zařízení od verze pro CPU.

5.4.1 Základní princip

Algoritmus Marching cubes, představený v kapitole 4.3, je založen na ohodnocení a zpracování jednotlivých krychlí, což je patrné i z názvu algoritmu. Tato krychle reprezentuje skupinu voxelů, které spolu sousedí v datové mřížce a vytváří tak základní element algoritmu. Vrchol krychle je ohodnocen intenzitou odpovídajícího voxelu, který získáme dle souřadnic krychle ze dvou sousedících řezů nasnímaných dat.

Počáteční fáze algoritmu se snaží získat informaci o tom, zda je krychle protnuta hledaným povrchem objektu. Povrch je určen *hranicí*, která je jedním ze vstupních parametrů. Hodnota vstupního parametru hranice se pohybuje v normalizovaném intervalu $\langle 0, 1 \rangle$, který je při výpočtu transformován do intervalu určeného nejmenší a největší intenzitou vstupních dat. Informaci, zda krychle protíná povrch objektu, reprezentujeme *indexem konfigurace.*

Při implementaci bylo třeba specifikovat číselné označení jednotlivých vrcholů krychle (viz obrázek 5.2). Důvodem je to, že jednotlivé bity indexu konfigurace odpovídají vrcholům krychle a značí, zda se vrchol na daném indexu nachází vně nebo uvnitř objektu. Jelikož má krychle osm vrcholů, k reprezentaci nám postačí 8-bitové číslo.



Obrázek 5.2: Označení vrcholů a hran krychle

Index konfigurace získáme postupným porovnáváním vstupního parametru hranice s hodnotami vrcholů krychle. Pokud je intenzita vrcholu větší než hodnota hranice, považujeme tento vrchol za vnitřní bod objektu a nastavíme příslušný bit indexu konfigurace na hodnotu 1, v opačném případě jej nastavíme 0. Index konfigurace tedy představuje 256 kombinací jak mohou být vrcholy krychle ohodnoceny. Lze si všimnout, že existují dvě varianty z celkového počtu kombinací, kde krychle nemá žádný průnik s povrchem objektu. Jedná se o indexy konfigurace s hodnotou 0 nebo 255. Všechny bity indexu jsou nastaveny buď na hodnotu 0 (resp. 1), což znamená, že krychle je zcela vně (resp. uvnitř) objektu. Hlavní jádro tohoto přístupu spočívá ve zjištění dvojic vrcholů krychle, které jsou spojeny hranou. Jeden vrchol se vyskytuje mimo objekt a druhý uvnitř, což znamená, že hrana protíná povrch objektu v bodě, který je jedním z hledaných vrcholů polygonální sítě.

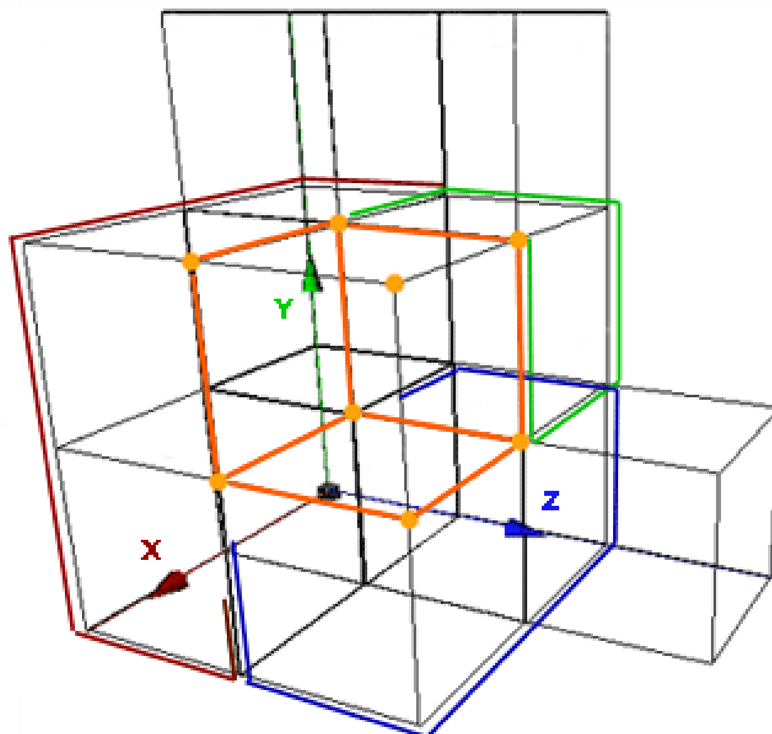
Index konfigurace je z pohledu algoritmu Marching cubes zcela klíčová informace, kterou využíváme i v dalších krocích vektorizace. Abychom vytvořili vrcholy (resp. trojúhelníky) polygonálního modelu, potřebujeme zjistit, které hrany protínají objekt a tedy mezi kterými vrcholy krychle budeme hledat vrchol. Z toho důvodu jsme definovali konstantní seznam 12-bitových hodnot (pomocí 16-bitových hodnot s využitím pouze 12 bitů) o délce 256 položek, jejichž index odpovídá indexu konfigurace. Tyto hodnoty (podobně jako v případě indexu konfigurace) obsahují na jednotlivých bitových pozicích příznak, zda se na hraně daného indexu vyskytuje vrchol povrchu objektu (číslování hran je naznačeno v obrázku 5.2). Pokud ano, tak provedeme lineární interpolaci krajních bodů hrany řízenou hodnotami hranice a daných vrcholů, čímž získáme nový vrchol povrchu.

Při vektorizaci využíváme ještě další konstantní pole, jehož položky jsou také adresovatelné indexem konfigurace (stejně jako v tabulce hran). Tentokrát se jedná o trojice indexů

hran.

5.4.2 Sdílení vrcholů

V daném momentu již známe princip generování výsledného modelu. Pokud se čtenář zamyslí nad obsahem několika předchozích odstavců, jistě si uvědomí, že popisovaný způsob vytvoří duplicitní vrcholy sítě. Krychle, které mají společnou hranu protínající objekt, mohou vrchol hrany sdílet. Bylo nutné nalézt způsob, jak zamezit generování kopie vrcholu, který již figuruje v seznamu vrcholů, ale přitom zachovat nezávislost zpracování jednotlivých krychlí datové mřížky.



Obrázek 5.3: Sdílení vrcholů

Na obrázku 5.3 můžeme shlédnout výřez z datové mřížky volumetrických dat, kde žluté vrcholy označují zpracovávanou krychli. Při sdílení vrcholů vycházíme z předpokladu, že dříve (z pohledu souřadnic) vektorizovaná krychle má vytvořené všechny hledané vrcholy. V ilustrované situaci krychle sdílí hrany vyznačené oranžovou barvou. Červeně zvýrazněné krychle patří do skupiny krychlí, které mají shodnou Z-ovou souřadnici (o 1 menší než aktuálně zpracovávaná krychle), dále nazýváme tuto 2D mřížku *slice*. Modře jsou vyznačeny krychle, které mají společnou Y-ovou souřadnici, jedná se o řádek krychlí generovaných před aktuálním řádkem, dále budeme nazývat *row*. A konečně zelená krychle značí předchozí krychli z pohledu souřadnice osy X, nazýváme *cube*. Aktuální krychle má devět společných hran se třemi, dříve zpracovávanými krychlemi. Výjimku tvoří krychle, které mají alespoň

jednu složku souřadnice nulovou – sdílí buď méně hran nebo žádnou. Tabulka 5.2 naznačuje sdílení hran se třemi možnými krychlemi podle souřadnic krychle.

X	**Y**	**Z**	**Cube**	**Row**	**Slice**
$= 0$	$= 0$	$= 0$	$\times$	$\times$	$\times$
$= 0$	$= 0$	$\neq 0$	$\times$	$\times$	$\surd$
$= 0$	$\neq 0$	$= 0$	$\times$	$\surd$	$\times$
$= 0$	$\neq 0$	$\neq 0$	$\times$	$\surd$	$\surd$
$\neq 0$	$= 0$	$= 0$	$\surd$	$\times$	$\times$
$\neq 0$	$= 0$	$\neq 0$	$\surd$	$\times$	$\surd$
$\neq 0$	$\neq 0$	$= 0$	$\surd$	$\surd$	$\times$
$\neq 0$	$\neq 0$	$\neq 0$	$\surd$	$\surd$	$\surd$

Tabulka 5.2: Paměť sdílených vrcholů

Výsledkem implementace těchto poznatků do stávajícího procesu generování vrcholů byla znatelná optimalizace a to jak po stránce výkonu eliminací výpočtů duplicitních vrcholů, tak po stránce paměťových nároků díky snížení celkového počtu vrcholů polygonální sítě. Při vytváření trojúhelníku se ovšem nevyhneme potřebě znát skutečné indexy vrcholů v jejich seznamu. Tento problém jsme vyřešili stanovením přesných pravidel pro zisk požadovaných indexů. V tabulce 5.3 uvádíme výčet možných kombinací souřadnic krychle (pro nás je rozhodující faktor, jestli souřadnice obsahuje nulovou složku souřadnice) a k tomu odpovídající skupinu krychlí, odkud získáme index existujícího vrcholu.

Vycházíme-li z číslování hran dle obrázku 5.2 a výše uvedeného textu, vyvozujeme z toho následující poznatky ke sdílení vrcholů při vektorizaci objektu (viz 5.3).

Číslo hrany	**Generujeme nový vrchol při**	**Index existujícího vrcholu je v**
0	X = 0 a Y = 0	Cube, Row
1	Y = 0	Row
2	Y = 0	Row
3	Y = 0 a Z = 0	Row, Slice
4	X = 0	Cube
5	vždy	-
6	vždy	-
7	Z = 0	Slice
8	X = 0 a Z = 0	Cube, Slice
9	X = 0	Cube
10	vždy	-
11	Z = 0	Slice

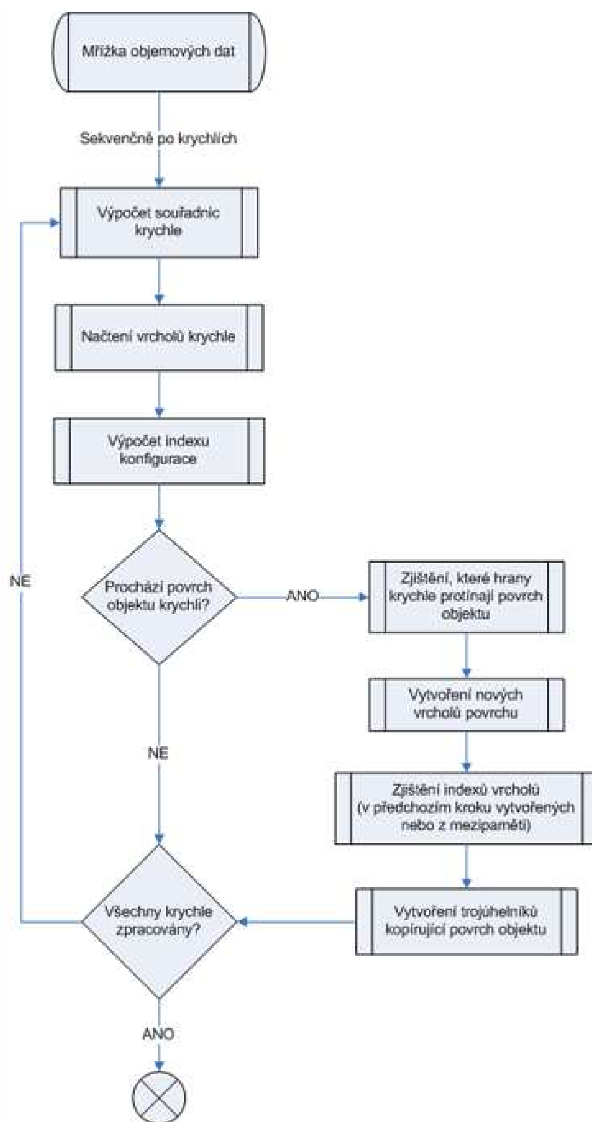
Tabulka 5.3: Pravidla vytváření sdílených vrcholů

CPU a GPU implementace se liší ve způsobu získání indexů existujících vrcholů, ovšem princip zůstává stejný. Rozdíly jsou dány především odlišným přístupem zpracování krychlí a výhodnější variantou po stránce výkonu.

5.4.3 CPU implementace

I když považujeme vektorizaci pomocí Marching cubes za jednoduchou a přímočarou, přesto existuje několik různých variant algoritmu, jak dosáhnout očekávaného výsledku. Výběr nejvhodnějšího postupu ovlivňuje především hardwarová vybavenost výpočetního zařízení a s tím spojené omezující podmínky.

Při návrhu postupu zpracování na CPU jsme žádné podstatné omezení nepocítili. Referenční počítač, použitý při vývoji a následném testování, byl vybaven operační pamětí, která splňovala kapacitní požadavky. Pokud bychom aplikaci spustili na stroji s nedostatkem volné paměti, docházelo by při výpočtu k tzv. „swapování“ (z angl. prohození) dat mezi operační pamětí RAM a diskovým úložištěm, což by vedlo k výraznému zpomalení celého procesu vektorizace.



Obrázek 5.4: Proces vektorizace na CPU

Zvolený přístup CPU verze je čistě sekvenční (jednovláknový), postupně tedy procházíme

jednotlivé krychle datové mřížky (viz 5.4). Seznamy vrcholů a trojúhelníků se chovají díky definovaným rozšiřujím funkcím jako dynamické struktury. To je výhodné z toho důvodu, že předem neznáme konečný počet vygenerovaných vrcholů, ani trojúhelníků polygonální sítě. V každé krychli přidáváme nové vrcholy a trojúhelníky na konec seznamu. Seznamy se v případě dosažení maximální kapacity automaticky zvětší o definovaný, konstantní počet položek. Konstantu jsme zvolili optimálně nejen z pohledu příliš častých operací s pamětí, ale také z pohledu zabránění alokacím zbytečně velkých úseků dynamické paměti.

Z textu předchozích kapitol by měla být většina procesů v obrázku 5.4 jasná a pochopitelná. Ovšem je třeba objasnit, jakým způsobem získáme indexy vygenerovaných vrcholů při vytváření trojúhelníkových polygonů v situaci, kdy chceme sdílet vrcholy mezi sousedními krychlemi. Jelikož nové vrcholy pouze přidáváme na konec seznamu vrcholů, nelze zpětně zjistit, který vrchol v seznamu patří sousední krychli. Jediná vlastnost vektorizace, na kterou se můžeme spolehnout, je procházení krychlemi sekvenčně. To znamená, že v době zpracování dané krychle víme, že její sousední krychle, které tu aktuálně zpracovávanou předcházely, již mají všechny své vrcholy vytvořené. Toho lze využít k implementaci tzv. „mezipaměti“, která zcela kopíruje struktury slice, row a cube z kapitoly 5.4.2. Slice představuje 2D mřížku krychlí předchozího řezu, row řádek krychlí pod aktuální krychlí a cube krychli předchozí na stejném řádku. Každá krychle v mezipaměti uchovává pouze indexy čtyř společných hran. Pokud nastane situace, že aktuální krychle by měla vytvořit vrchol na hraně, která je společná s dříve zpracovávanou krychlí, proces vytvoření krychle se přeskočí a z mezipaměti načteme pouze index vrcholu v seznamu vrcholů, stejně jako nám definuje tabulka 5.3. Po zpracování každé krychle mezipaměť aktualizujeme novými indexy vrcholů.

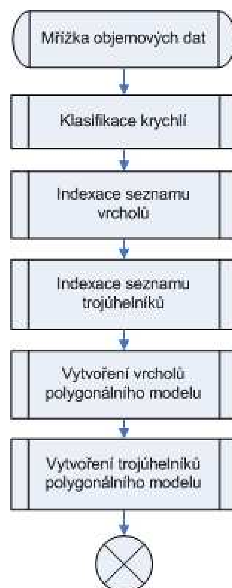
5.4.4 GPU implementace

V posledním bodě implementační části se budeme zabývat tím, jak vektorizační algoritmus vhodně aplikovat na grafické zařízení. Tato fáze potřebuje hlubší zamyšlení nad výhodami, ale také omezeními, která jsou spojena s programováním pro GPU.

Ve srovnání s CPU variantou je princip zpracování zcela odlišný. Na GPU budeme pracovat s jednotlivými krychlemi datové mřížky zvlášť, pro každou vytvoříme vlastní výpočetní vlákno, které poběží paralelně se všemi ostatními. Před spuštěním jakéhokoliv funkčního jádra (funkce prováděná na GPU) je třeba specifikovat, v kolika vláknech jádro poběží a jak budou vlákna organizována. V každém vlákně potřebujeme danou krychli nějakým způsobem identifikovat, k tomu využijeme možnosti sdružovat skupiny vláken do bloků a bloky následně do gridů (viz 2.5.2). Jádru, které pracuje paralelně přes všechny krychle, vytvoříme bloky organizované v dvoudimenzionálním gridu. Tedy jedna dimenze odpovídá počtu krychlí v ose X a druhá dimenze počtu krychlí v ose Y datové mřížky. Zbývající počet v ose Z mapujeme na počet vláken jednoho bloku (také z důvodu omezení max. počtu vláken na 512 – osa Z bývá nejmenší), čímž jsme pokryli kompletní vstupní objemová data. Naopak dimenze gridu je omezena na 65 535 bloků v každé ose, což je nadmíru dostačující pro všechna dostupná data.

V následujícím textu si přiblížíme omezující faktory, které nás vedli k podstatně rozdílnému návrhu. V samotném kódu jádra nelze alokovat dynamickou paměť ve videopaměti grafické karty. Je vždy nutné postupovat tak, že před spuštěním alokujeme dynamickou paměť požadované velikosti, spustíme výpočetní jádro, které nám např. výsledky uloží do námi vytvořené paměti, a po ukončení jádra (tedy opět v kódu zpracovávaném na CPU) výsledná data přeneseme z GPU do operační paměti. Protože v jádře nelze používat prin-

cip dynamických polí a bufferů, bylo nutné algoritmus rozfázovat tak, abychom již před samotným generováním vrcholů a trojúhelníků polygonální sítě znali počty těchto elementů a mohli tak alokovat dostatečnou část videopaměti. Postupné kroky zobrazuje obrázek 5.5, podrobněji se jim věnujeme v následujících kapitolách.



Obrázek 5.5: Kroky vektorizace na GPU

Klasifikace krychlí

Před spuštěním klasifikace je nutné přenést vstupní objemová data do paměti grafické karty. Stejně jako v CPU implementaci si budeme v procesu vektorizace pomáhat konstantními tabulkami s bitovými příznaky protínajících hran krychle a definicemi trojúhelníků. Tyto tabulky je vhodné umístit do texturovací paměti, která disponuje rychlou vyrovnávací pamětí.

Význam klasifikačního kroku spočívá v zisku informací o počtech vrcholů a trojúhelníků, které algoritmus z objemových dat vytvoří. Získané informace jsou důležité pro alokaci potřebné paměti před generováním elementů polygonální sítě. Dále potřebujeme zajistit vazbu mezi vrcholy a trojúhelníky. K tomuto účelu byla definována pomocná pole indexů (dále indexační pole), které velikostí odpovídají počtu krychlí datové mřížky. Jedna položka pole je reprezentována 32-bitovým číslem (rozsah čísla je dostačující, počet generovaných elementů bývá v extrémních případech maximálně v desítkách miliónů), které představuje index do seznamu vrcholů, resp. trojúhelníků. Tato položka je jednoznačně identifikovatelná pomocí souřadnic krychle, čímž zaručíme nezávislost na pořadí zpracování krychlí.

Přestože jsme vytvořili pomocná pole kvůli indexům do seznamů, přímo k výpočtu indexů v této fázi nedochází. Jak bylo již několikrát zmíněno, nemůžeme se spolehnout na pořadí zpracovávaných vláken a tedy nemůžeme v dané chvíli zjistit, kolik vrcholů či trojúhelníků budou předchozí krychle generovat. Jediné, co v této fázi provedeme, je výpočet skutečného počtu vrcholů a trojúhelníků, které zpracovávaná krychle bude generovat, a zápis této informace na odpovídající místo v indexačních polích. Pro jistotu zdůrazníme, že ukládáme počet „skutečně“ vytvářených elementů, tzn. pokud sdílíme vrcholy mezi krychlemi, řídíme se podle pravidel stanovených v tabulce 5.3.

Vzhledem k celkové paměťové náročnosti vektorizace na GPU jsou indexační pole nejvíce zatěžující struktury. Referenční objemová mřížka má rozměry $512 \times 512 \times 147$, tzn. že počet krychlí je $511 \times 511 \times 146 = 38\ 123\ 666$. Při velikosti jedné položky 4 B (32-bitové číslo) se dostáváme na cca 153 MB paměti, a jelikož používáme indexační pole dvě (pro vrcholy a trojúhelníky) tak spotřebujeme 306 MB.

Indexace

Klasifikací jsme zjistili počty elementů v jednotlivých krychlích. Abychom tyto indexační pole mohli transformovat na číselné odkazy do seznamů vrcholů a trojúhelníků, musíme provést operaci, která na pozici položky vloží součet všech předchozích položek seznamu obsahující počty elementů. Tento proces se nazývá *Prefix scan* nebo také *Prefix sum* (přeloženo jako suma předchozích hodnot).

Obecně je Prefix scan definován jako operace s binárním, asociativním operátorem $\oplus$ nad polem s n položkami.

$$[a_0, a_1, ..., a_{n-1}].$$

Prefix scan vrátí výsledné pole

$$[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1}]$$

Metoda Prefix scan nabízí několik modifikací, které mírně mění základní myšlenku algoritmu. Lze měnit směr průchodu, binární operátor (místo sčítání můžeme násobit, apod.), ale také existují dvě varianty určující operandy každého kroku algoritmu. První varianta je naznačena v předchozí definici a nazývá se *inkluzivní.* Hodnota na dané pozici je vypočtena binární operací nad všemi předcházejícími položkami včetně té aktuální. Naopak *exkluzivní* skenování provede binární operaci pouze s předcházejícími položkami. V případě pole s n položkami

$$[a_0, a_1, ..., a_{n-1}],$$

pak exkluzivní Prefix scan vrátí

$$[0, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2}].$$

Proces Prefix scan s konkrétními hodnotami v poli demonstruje následující příklad. Mějme pole hodnot

$$[3, 1, 7, 0, 4, 1, 6, 3],$$

potom inkluzivní varianta vrátí pole

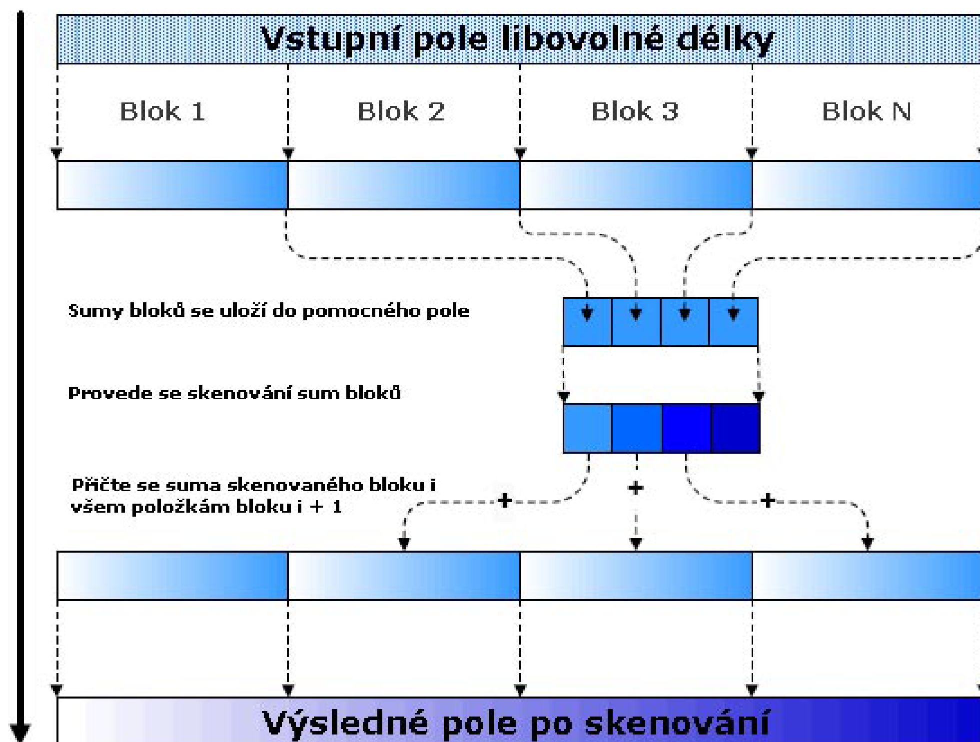
$$[3, 4, 11, 11, 14, 16, 22, 25]$$

a exkluzivní vrátí pole

$$[0, 3, 4, 11, 11, 14, 16, 22].$$

Z naznačených příkladů bylo zřejmé, že pro náš případ jsme potřebovali exkluzivní metodu Prefix scan. Naivním přístupem šlo daný algoritmus implementovat jako sekvenční průchod polem. Jelikož indexační pole figurovaly pouze v paměti grafické karty (nikdy nejsou potřeba v systémové RAM), bylo vhodné Prefix scan implementovat v podobě funkčního jádra přímo na GPU. Tato situace ovšem vyžadovala zamyšlení se nad tím, jak daný algoritmus paralelizovat a urychlit tak celkový čas zpracování využitím výkonného paralelního zpracování na GPU.

Účinný způsob bylo podělit oblast pole do několika oddělených bloků o vhodném počtu položek. V každém bloku spočítat sumu všech položek a tuto hodnotu uložit v nezměněném pořadí bloků do pomocného pole. Potom provést Prefix scan nad těmito sumami a výsledky na odpovídajích pozicích poté přičíst ke všem položkám jednotlivých bloků. Navržený způsob je naznačen na obrázku 5.6.



Obrázek 5.6: Paralelní Prefix sum

Popsaná metoda paralelního algoritmu Prefix scan je obsažena ve volně dostupné knihovně *CUDPP* (zkr. CUDA Data Parallel Primitives Library). Knihovna je velice dobře optimalizovaná pro práci s rychlou sdílenou pamětí jednotlivých multiprocesorů grafické karty. Rozhodli jsme se použít ji i v naší implementaci za účelem dosažení maximálního výkonu (viz [9]). Jedinou větší nevýhodou byla potřeba alokovat paměť pro výsledné pole, což při velikosti jednoho indexačního pole cca 150 MB znamenalo spotřebu další podstatné části paměti.

V okamžiku provedení Prefix scan máme původní a nové indexační pole, potřebujeme tedy dopočítat celkový počet generovaných elementů polygonální sítě. To lze uskutečnit sečtením posledních položek obou indexačních polí, protože původní pole obsahuje počet elementů v každé krychli a pole po prefixu obsahuje na pozici krychle vždy součet elementů všech předchozích krychlí. Jelikož indexační pole existují pouze v paměti GPU, bylo pro tuto akci implementováno jednovláknové jádro.

Vytvoření polygonální sítě

V této fázi vektorizace již známe všechny nezbytné informace k vytvoření polygonální sítě reprezentující hledaný povrch objektu. Generování vrcholů i trojúhelníků probíhá na stejném principu jako v části pro CPU. Abychom nemuseli znovu přistupovat do paměti a načítat

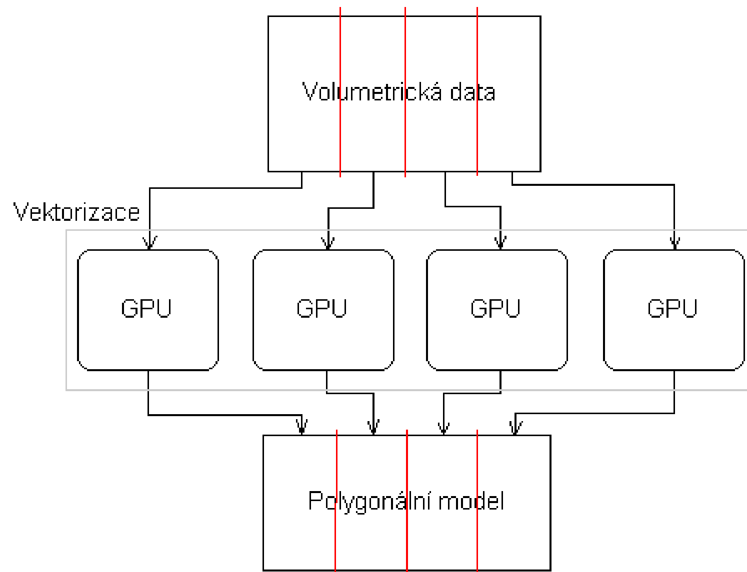
hodnoty vrcholů krychle, bylo v průběhu klasifikace vytvořeno pomocné pole obsahující index konfigurace, což urychlí zjištění protínajících hran a indexů vrcholů jednotlivých trojúhelníků. Indexační pole obsahuje pro danou krychli index pouze prvního vytvářeného elementu výsledného seznamu, další elementy v rámci krychle automaticky řadíme do výsledného seznamu za první element. V případě vrcholů odpovídá pořadí ukládaných elementů pořadí protínajících hran (algoritmus prochází hrany od 0. po 11., viz 5.2). V případě trojúhelníků je pořadí definované konstantní tabulkou trojúhelníků.

Vygenerované vrcholy a trojúhelníky nakonec přesuneme z paměti GPU do operační paměti RAM a vektorizace objemových dat končí.

5.4.5 Multi GPU

Z pohledu programového rozhraní CUDA jsou grafické karty považovány za obecná výpočetní zařízení. V aplikacích musíme vždy před spuštěním funkčních jader zvolit jedno z dostupných zařízení, které se použije k běhu výpočetního kódu. Pokud počítačová sestava disponuje více grafickými kartami, lze vyvíjené aplikace navrhnout tak, aby výpočty probíhaly na všech dostupných GPU. Implementace knihovny dokáže vektorizační proces rozdělit mezi více grafických zařízení.

Jelikož je nutné zvolit zařízení před spuštěním funkčního jádra, museli jsme rozdělit aplikaci již na úrovni CPU kódu. Na začátku vektorizační funkce zjistíme dostupná zařízení a podle počtu vytvoříme vlákna, kde každé bude komunikovat s jednou grafickou kartou. Před námi byl úkol, jak vhodně rozdělit objemová data, a také jak následně výstupní data spojit v jeden polygonální model, jak nastiňuje obrázek 5.7.



Obrázek 5.7: Multi GPU vektorizace

Vektorizační proces probíhá mezi grafickými kartami paralelně, nezávisle na ostatních. Vstupní volumetrická data jsme rozdělili řezy (v obrázku 5.7 označené červeně) poměrem k počtu GPU. Každé grafické zařízení pracuje se svojí částí vstupní datové mřížky. Jelikož neexistuje žádná interakce mezi GPU, tak i výstupní data tvoří samostatné, dílčí celky.

První problém nastal při vytváření nových vrcholů povrchu, přesněji řečeno jejich souřadnic. Do jednotlivých vláken bylo třeba přidat informaci o posunu generovaných vrcholů podle toho, na jak široké úseky jsme data rozdělili. Druhý problém vznikl v případě vytvořených trojúhelníků, které se odkazují do seznamu vrcholů. Spojením seznamů vrcholů z jednotlivých GPU do výsledného seznamu se změnily i indexy vrcholů, a ty je nutné v trojúhelnících aktualizovat. Aktualizaci provedeme přičtením hodnoty, která odpovídá počtu vrcholů ve výsledném seznamu před připojením vrcholů z aktuální GPU.

Popsaná metoda bohužel generuje vedlejší efekt – artefakt na výsledném polygonálním modelu. V místě řezu objemových dat vzniknou duplicitní vrcholy (model bude tvořen z více vrcholů, počet trojúhelníků zůstává stejný), které navíc mají rozdílné normálové vektory. Při vizualizaci lze taková místa řezů jednoduše identifikovat. S tímto jsme se nakonec smířili, protože větší hodnotu pro nás v danou chvíli mělo srovnání výkonů implementací.

Technologie, která se nazývá *SLI* (zkr. Scalable Link Interface), spojuje více grafických karet na úrovni ovladačů. Jejich úkolem je pak rozdělit si části renderovaného výstupu, jejichž složením vznikne výsledný obraz. Pokud je technologie SLI aktivní, tak se skupina grafických karet v aplikacích CUDA tváří jako jedno zařízení. Ovšem na úrovni ovladačů se výpočty nedělí jako při renderování scény, vše se provádí pouze na jedné grafické kartě. Vysvětlení je v podstatě jednoduché. V aplikacích používáme ukazatele a často pracujeme přímo s adresami paměti, tzn. že by paměťové prostory jednotlivých GPU museli tvořit jednotný, virtuální, paměťový prostor, což v současné době není technologicky zajištěné.

Kapitola 6

Dosažené výsledky

V implementační části této práce se nám podařilo dosáhnout uspokojivého výsledku. Aplikace generuje (jak v CPU, tak GPU části) shodné polygonální modely s optimalizovaným počtem vrcholů.

6.1 Měřená vstupní data

Objemová data, která jsme použili pro měření časů vektorizačního procesu a výkonu testovacího hardwaru, reprezentují nasnímanou lidskou hlavu dospělého člověka. Originální rozměr datové mřížky vstupních dat byl $512 \times 512 \times 147$ voxelů, tento rozměr jsme stanovili jako referenční model pro fázi vývoje a následných testů.

Abychom mohli porovnávat také objemová data s menším počtem voxelů, byla originální mřížka zmenšena podvzorkováním, čímž byly vybírány jen vzorky odpovídající nové vzorkovací frekvenci. Tu lze získat pomocí vztahu

$$f_n(d) = T(d)/N(d) \tag{6.1}$$

kde $f_n(d)$ odpovídá nové vzorkovací frekvenci ve směru d, $N(d)$ rozměru originálních vstupních dat ve směru d a $T(d)$ požadovaným rozměrem ve směru d. Tento způsob je velice snadný na implementaci, ale v datech se objevují artefakty, zejména při mnohonásobně menších rozměrech. V našem případě není důležitý obsah, ale počet voxelů vstupních dat a generovaných vrcholů či trojúhelníků, tedy vlastnosti, které prověří rychlost zpracování.

6.2 Testovaný hardware

Po dobu vývoje a měření jsme měli k dispozici dvě počítačové sestavy podporující programové rozhraní CUDA. Následující seznamy obsahují hardwarové komponenty, které přímo ovlivňují čas průběhu vektorizace.

Procesor:	Intel Core 2 Duo E6550 2,33 GHz
Operační paměť:	2 GB
Grafická karta:	NVidia GeForce 8800 GTX 768 MB

Tabulka 6.1: Sestava č. 1

Procesor:	Intel Core 2 Duo E6850 3,00 GHz
Operační paměť:	3 GB
Grafická karta:	2× NVidia GeForce 8800 GT 512 MB

Tabulka 6.2: Sestava č. 2

Sestavu č. 2 jsme použili pro testování běhu aplikace na více grafických zařízeních a zároveň části pro CPU. Paměť o velikosti 512 MB není dostačující pro vektorizaci objemových dat maximálního rozměru, pro variantu běhu aplikace na jedné grafické kartě jsme použili pouze sestavu č. 1.

6.3 Počet vrcholů a trojúhelníků

Předtím než jsme začali s časovým měřením algoritmů knihovny, získali jsme informace o počtech vrcholů a trojúhelníků vybraných testovacích sad. Počty generovaných elementů jsou samozřejmě závislé na hraniční hodnotě vektorizace a na komplexnosti povrchu cílového objektu. Pro samotné testování jsme vybrali tři sady vstupních dat, které zastupují data ve třech odlišných rozměrových skupinách.

	$25 \times 25 \times 7$			
Hranice	Vrcholy (single)	Vrcholy (multi)	Vrcholy (all)	Trojúhelníky
0,0	972	1 074	3 254	1 602
0,1	1 009	1 117	3 444	1 716
0,2	1 102	1 214	3 834	1 898
0,3	1 447	1 551	5 114	2 392
0,4	1 154	1 238	4 034	1 764
0,5	815	874	2 810	1 168
0,6	247	277	884	316
0,7	0	0	0	0
0,8	0	0	0	0
0,9	0	0	0	0
1,0	0	0	0	0

Tabulka 6.3: Počet generovaných elementů ($25 \times 25 \times 7$)

V tabulkách 6.3, 6.4 a 6.5 jsou uvedeny vybrané varianty vstupních dat, které jsme v průběhu měření střídali. Počet vrcholů ve sloupci „single“ odpovídá výsledku vektorizace provedené na jedné GPU (sestava č. 1), „multi“ značí aplikaci na více grafických zařízení (sestava č. 2). „All“ je počet vrcholů vytvořených na jedné grafické kartě (sestava č. 1), ale vrcholy nejsou mezi krychlemi sdíleny, každá krychle vytvoří unikátní vrcholy. Vzhledem k vysokému počtu trojúhelníků by bylo nadmíru vhodné implementovat decimační algoritmus (viz 4.5.2), který by počet snížil, ale zachoval přitom tvar polygonálního modelu. Z časových důvodů decimace není zanesena do této práce, zmiňujeme to pouze jako možné budoucí rozšíření knihovny.

	128 × 128 × 36			
Hranice	Vrcholy (single)	Vrcholy (multi)	Vrcholy (all)	Trojúhelníky
0,0	49 389	50 308	190 348	92 718
0,1	37 538	38 119	146 218	73 260
0,2	41 949	42 564	163 684	81 916
0,3	65 052	66 238	255 346	127 868
0,4	56 525	57 497	221 996	110 708
0,5	47 782	48 530	187 568	92 910
0,6	25 395	25 839	99 960	47 892
0,7	92	92	368	136
0,8	0	0	0	0
0,9	0	0	0	0
1,0	0	0	0	0

Tabulka 6.4: Počet generovaných elementů (128 × 128 × 36)

	512 × 512 × 147			
Hranice	Vrcholy (single)	Vrcholy (multi)	Vrcholy (all)	Trojúhelníky
0,0	1 508 820	1 514 679	5 968 402	3 000 234
0,1	650 578	652 811	2 586 054	1 292 966
0,2	714 497	716 825	2 841 716	1 421 060
0,3	1 171 216	1 175 916	4 663 692	2 332 054
0,4	1 010 927	1 014 975	4 026 396	2 013 744
0,5	882 361	885 755	3 513 820	1 756 692
0,6	600 356	602 682	2 392 136	1 193 656
0,7	3 446	3 446	13 784	6 784
0,8	0	0	0	0
0,9	0	0	0	0
1,0	0	0	0	0

Tabulka 6.5: Počet generovaných elementů (512 × 512 × 147)

6.4 Režie volání funkcí na CPU a jader na GPU

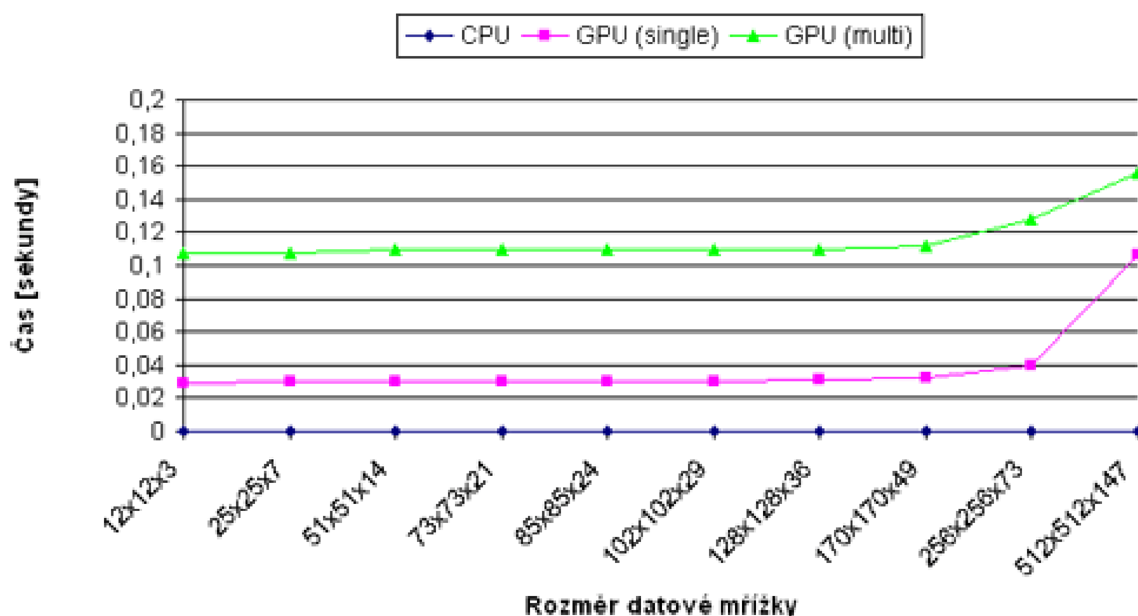
V předchozí části jsme získali přehled, z kolika elementů se výsledná polygonální síť skládá. Než ukážeme přímé srovnání CPU a GPU vektorizací s testovacími daty, je třeba poukázat na časové úseky, které stráví zařízení během volání funkcí či jader implementací. Tyto režie jsme se při vývoji samozřejmě snažili co nejvíce omezit, ovšem ve většině případů se jim bohužel vyhnout nelze. Měření probíhalo tím způsobem, že jsme hlavní funkce (CPU) a jádra (GPU) jednotlivých částí knihovny tzv. „zaslepily“. Tzn., že při měření probíhala vektorizace ve stejných krocích jako při „ostrém“ běhu, ovšem hlavní funkce nic nezpracovávaly.

Nelze si nevšimnout řádově mnohonásobně vyšších naměřených časů, které grafická zařízení strávila režijními úkony. Algoritmus vektorizace při zpracování na GPU s sebou navíc nese přesuny dat z operační paměti na GPU a zpět, což je hlavní úzké místo při práci s GPU. Menší, v porovnání s kopírováním dat ovšem zanedbatelná, část režie patří

Rozměr mřížky	CPU [s]	GPU single [s]	GPU multi [s]
$12 \times 12 \times 3$	0,000001	0,02887	0,107332
$25 \times 25 \times 7$	0,000004	0,029714	0,107751
$51 \times 51 \times 14$	0,000006	0,029799	0,109217
$73 \times 73 \times 21$	0,000007	0,029878	0,10968
$85 \times 85 \times 24$	0,000008	0,029868	0,109178
$102 \times 102 \times 29$	0,000009	0,030175	0,109342
$128 \times 128 \times 36$	0,000013	0,030697	0,109437
$170 \times 170 \times 49$	0,000016	0,032647	0,111603
$256 \times 256 \times 73$	0,000015	0,039668	0,127498
$512 \times 512 \times 147$	0,000017	0,106343	0,156144

Tabulka 6.6: Režie volaných funkcí vektorizace

organizaci spouštění a řízení výpočetních vláken jednotlivých jader.



Obrázek 6.1: Režie volaných funkcí

V případě více grafických zařízení ještě musíme započítat dělení vstupních dat a spojení dat výstupních (i když výstupem v tomto případě je prázdná polygonální síť, tak kontroly vygenerovaných seznamů elementů proběhnout musí). Žádné z vyjmenovaných úkonů se v části pro CPU nekonají, pouze se sekvenčně projde datová mřížka.

6.5 Globální a sdílená paměť

Tato kapitola srovnává výkon algoritmu při použití standardní globální paměti s optimalizovaným přístupem do paměti v podobě sdílené paměti multiprocesoru. Instrukce používané za účelem čtení (zápisu) dat z globální paměti stráví zhruba 400 cyklů procesoru. Naopak

zpoždění při práci se sdílenou pamětí multiprocesoru se rovná 4 cyklům. Toto zjištění nám dalo další prostor ke zrychlení práce s pamětí.

Před započetím vektorizace musíme data vždy přenést do globální paměti grafického zařízení. Každý požadavek získat hodnotu voxelu objemových dat čtením přímo z globální paměti s sebou nese již zmíněné zpoždění. Při velkém počtu vláken toto zpoždění kompenzuje plánovač spouštění vláken – pokud nějaké vlákno čeká na data, plánovač zatím spustí jiné vlákno, takže výkon výrazně neklesne. I přesto se doporučuje využívat sdílenou paměť a to tím způsobem, že na počátku nahrajeme data z globální paměti do sdílené, po celou dobu zpracování pracujeme se sdílenou pamětí a na konci se výsledky uloží zpět do globální paměti.

Sdílená paměť má ovšem určitá omezení, která jsme museli vzít na vědomí již při návrhu. Uvědomíme-li si, že vlákna jednoho bloku jsou vždy vykonána na jednom multiprocesoru a mohou tedy sdílet data mezi sebou, lze tak zmenšit počet přístupů do globální paměti. Využijeme toho, že sousední krychle mají společné čtyři vrcholy a vlákna jsou vždy v datové mřížce organizovány po řádcích v ose Z. Tím snížíme počet čtení z globální paměti téměř na polovinu. Je však třeba vždy zkontrolovat, že potřebná velikost sdílené paměti není větší než její celková kapacita 4 096 B. Právě z tohoto důvodu nemůžeme v naší implementaci vektorizovat vstupní objemovou mřížku přesahující počet 256 voxelů v ose Z. Sdílená paměť se nám časově vyplatí pouze ve fázi klasifikace voxelů, kdy vždy pracujeme se všemi krychlemi řádku objemových dat.

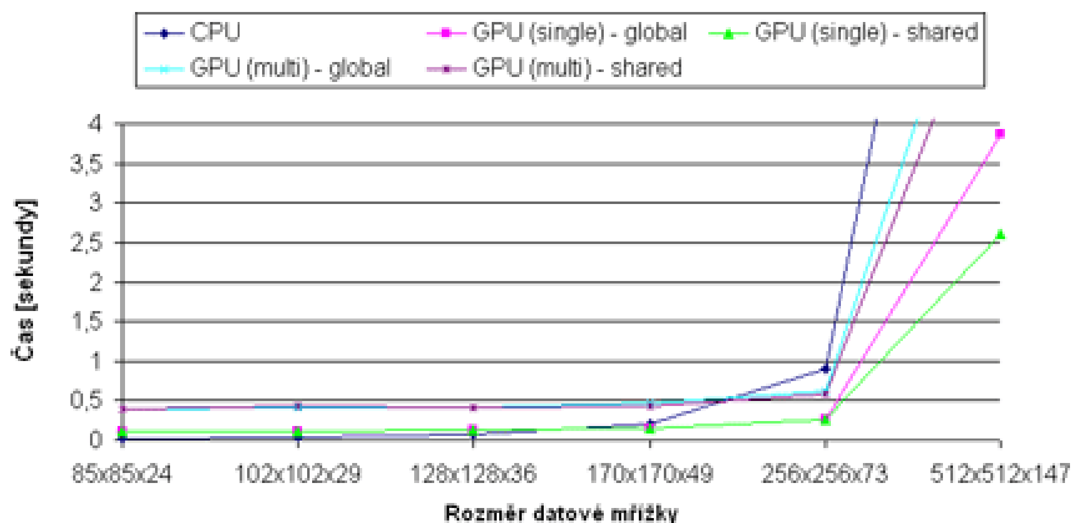
Rozměr mřížky	CPU [s]	GPU global	GPUs global	GPU shared	GPUs shared
$12 \times 12 \times 3$	0,000092	0,106819	0,424038	0,107559	0,387961
$25 \times 25 \times 7$	0,000704	0,108916	0,382013	0,109441	0,382559
$51 \times 51 \times 14$	0,005193	0,110585	0,38535	0,11142	0,421057
$73 \times 73 \times 21$	0,014731	0,113187	0,434412	0,114555	0,389397
$85 \times 85 \times 24$	0,022804	0,114987	0,391687	0,116626	0,399702
$102 \times 102 \times 29$	0,039881	0,118583	0,419951	0,120479	0,431178
$128 \times 128 \times 36$	0,080711	0,12845	0,408142	0,130446	0,408572
$170 \times 170 \times 49$	0,214489	0,145665	0,46466	0,15022	0,432257
$256 \times 256 \times 73$	0,893563	0,265192	0,615434	0,271462	0,579279
$512 \times 512 \times 147$	11,699359	3,860263	7,18812	2,608847	6,212149

Tabulka 6.7: Naměřené časy vektorizace v různých konfiguracích. Vektorizace byla provedena se stejnou hodnotou hranice 0,3 generující nejvyšší počet vrcholů.

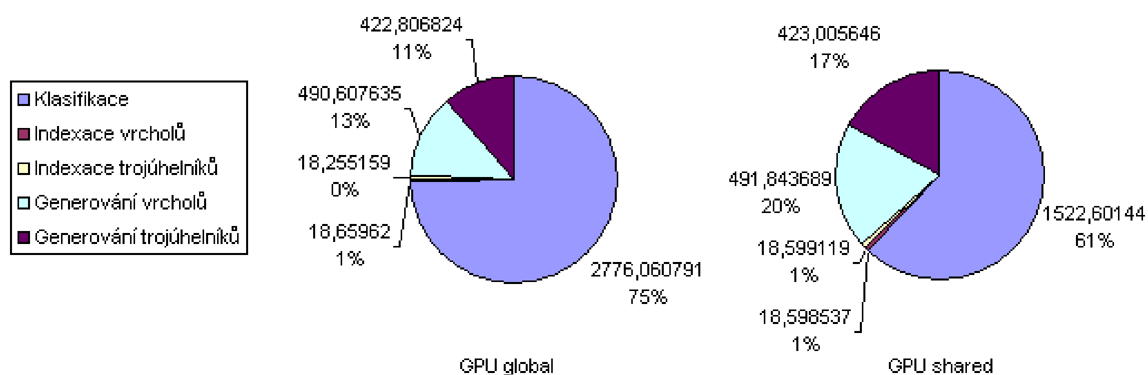
Z pohledu srovnání globální a lokální paměti jsme dosáhli očekávaného výsledku. Verze se sdílenou pamětí je při vektorizaci dat o největším počtu voxelů dokonce o více než jednu sekundu rychlejší, což nám vylepšilo i poměr výkonu v porovnání s CPU implementací.

6.6 Duplicitní vrcholy

Generování polygonálních sítí s duplicitními vrcholy je pro další využití modelu absolutně nepraktické. Ale z pohledu testování implementací to byl vhodný způsob, jak více zatížit grafické karty a procesor. Tímto krokem zvýšíme poměr výpočetních úkonů vzhledem k počtu přístupů do paměti, což více prověří skutečnou výpočetní sílu obou HW komponent.



Obrázek 6.2: Vektorizace objemových dat různých velikostí



Obrázek 6.3: Podíl jednotlivých fází vektorizace, časové srovnání globální a sdílené paměti (čas v milisekundách).

V tabulce 6.8 je výčet naměřených hodnot. Pokud se ve výsledcích zaměříme na větší rozměry objemových dat, lze si všimnout drtivého propadu procesoru v porovnání s grafickými kartami. Vektorizace maximální datové mřížky byla na grafické kartě GeForce 8800 GTX provedena téměř 19× rychleji než na CPU, což v plné míře ukázalo hlavní doménu využití paralelních grafických procesorů.

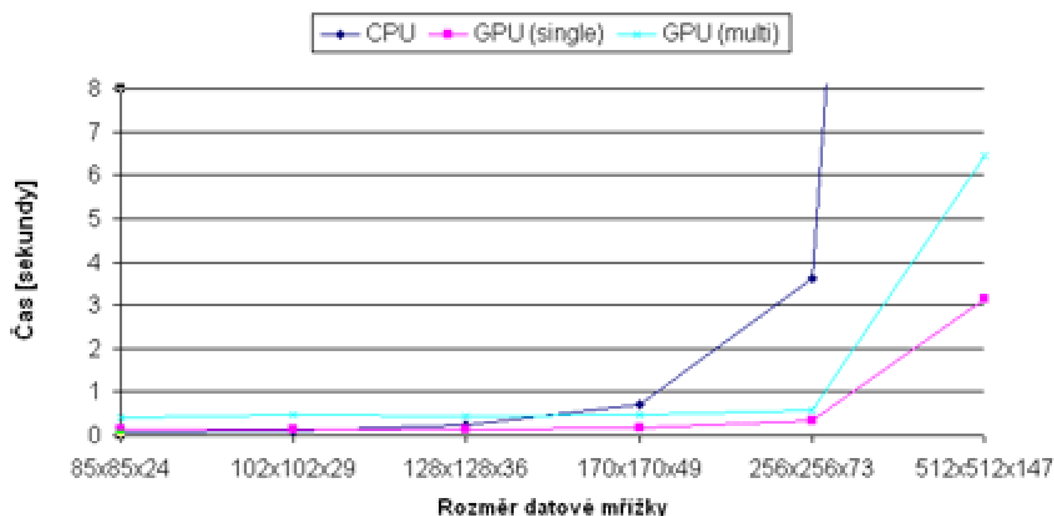
6.7 Zhodnocení

Požadavkem této práce bylo zvolit si vhodný algoritmus, který je výpočetně náročný, a podrobit CPU a GPU varianty implementací vzájemnému srovnání. Obě verze vektorizačního algoritmu Marching cubes jsme zasadili do rozhraní knihovny, kterou lze použít v dalších projektech, čímž jsme splnili požadavky stanovené v návrhu.

V průběhu vývoje jsme postupně přicházeli na různá vylepšení a optimalizace. Také jsme museli přizpůsobit jednotlivé kroky vektorizace prováděné na grafickém zařízení různým

Rozměr mřížky	CPU [s]	GPU (single) [s]	GPU (multi) [s]
$12 \times 12 \times 3$	0,000159	0,106129	0,412740
$25 \times 25 \times 7$	0,001337	0,108615	0,380772
$51 \times 51 \times 14$	0,009901	0,112307	0,421727
$73 \times 73 \times 21$	0,029831	0,118066	0,426471
$85 \times 85 \times 24$	0,051622	0,124768	0,395229
$102 \times 102 \times 29$	0,100032	0,128281	0,462402
$128 \times 128 \times 36$	0,236309	0,145093	0,444508
$170 \times 170 \times 49$	0,703601	0,172342	0,488649
$256 \times 256 \times 73$	3,619947	0,347354	0,582794
$512 \times 512 \times 147$	58,786299	3,149072	6,449651

Tabulka 6.8: Naměřené časy vektorizace generující duplicitní vrcholy sítě. Vektorizace byla provedena se stejnou hodnotou hranice 0,3.



Obrázek 6.4: Srovnání vektorizace polygonální sítě s duplicitními vrcholy

omezením daným architekturou GPU. Výsledná knihovna také umožňuje využít výkonu počítačových sestav, které obsahují i více než jednu grafickou kartu podporující CUDA.

V průběhu experimentálního měření jsme zjistili, že naše implementace vektorizačního algoritmu pro GPU je výkonnější v případě větších rozměrů datové mřížky (viz 6.7). Přeci jen přesun dat mezi pamětí grafické karty a operační paměti vyžaduje určitou časovou prodlevu, čemuž se v CPU implementaci zcela vyhneme. Navíc pokud využíváme pro vektorizaci více grafických karet zároveň, je proces zpomalován dělením vstupních dat a následným spojením dat výstupních. Pokud jsme srovnali výsledky měření vektorizace provedené na jedné nebo více grafických kartách, vždy byla výhodnější verze s jednou grafickou kartou.

Pokud srovnáme GPU verzi, která je nejvýrazněji optimalizovaná (tedy používá sdílenou paměť a sdílení generovaných vrcholů), dosáhli jsme oproti CPU verzi až 4,5-násobnému zrychlení, což lze považovat za velice slušný výsledek. Připomeňme, že se tak stalo při vektorizaci největší datové mřížky o rozměrech $512 \times 512 \times 147$.

Při profilování verze pro GPU bylo zjištěno, že grafické multiprocesory jsou během

vektorizace využívány maximálně z 33 %. Určitý potenciál pro další optimalizace knihovny by tu tedy byl. Domníváme se ovšem, že tato nízká vytíženost procesorů je způsobena častým přístupem do globální paměti. Také určitý vliv mělo strukturování jednotlivých gridů provádějící funkční jádra. Jelikož jsme v naší knihovně chtěli zachovat rezervu pro rozměry vstupních objemových dat, mapovali jsme nejmenší dimenzi datové mřížky (osa Z) na vytvářená vlákna. Tzn., že se vytvořily bloky s maximálním počtem 146 vláken, což je vzhledem k maximální možné hodnotě 512 (grafické karty GeForce 88xx) nízké číslo.

Kapitola 7

Závěr

Tato práce se zabývala novými technologiemi moderního grafického hardwaru, které lze s jistotou aplikovat i na obecné výpočetní algoritmy.

V první kapitole byl podán ucelenější úvod do problematiky GPGPU a souhrn možností využití GPU, které byly dostupné během několika posledních let, až po nejmodernější architektury a technologie současnosti, jejichž prezentace byla prvotním cílem.

V další kapitole byla stručně uvedena problematika objemových dat a metody jejich grafické vizualizace. V našem případě bylo požadavkem demonstrovat výpočetní schopnosti grafické karty, proto jsme z principu zvolili výpočetní metodu hledající povrch objektu místo přímých zobrazovacích technik.

Cílem práce bylo prostudovat možnosti dnešních GPU a následně získané znalosti zúročit při implementaci knihovny pro vektorizaci diskrétních hodnot volumetrických dat. Jinak řečeno, výsledkem se stala knihovna určená pro rekonstrukci objektu z prostorové mřížky obsahující nasnímaná objemová data do podoby polygonálního modelu, který lze klasickou technikou renderování scény vykreslit na grafický výstup. Výsledná knihovna se skládá ze tří částí – načtení objemových dat, vektorizace metodou Marching cubes a následný export modelu do 3D souborového formátu Obj. Právě algoritmus Marching cubes je pro svou výpočetní náročnost zcela přesunut na GPU. Knihovna si umí také poradit s hardwarovým prostředím, které má k dispozici více grafických karet podporující rozšíření CUDA. Jedním ze stěžejních výstupů práce bylo přímé srovnání implementace metody na CPU a GPU, kde jsme očekávali zasádní výkonostní rozdíly. Naše očekávání se v průběhu testování také naplnilo, ovšem pouze při větších rozměrech datové mřížky vstupních dat.

Hlavním požadavkem implementace bylo pokusit se maximálně využít výkonu grafické karty a zároveň dosáhnout co nejlepších výsledků, bez ohledu na kvalitu výstupu. Algoritmus Marching cubes generoval výstupní polygonální modely s velkým počtem vrcholů a trojúhelníků, jakožto další pokračování projektu bych navrhoval decimaci polygonální sítě pomocí známých decimačních technik.

Tato práce mi umožnila hlouběji vstoupit do problematiky dnešních grafických karet a rozšířit tak své znalosti v oboru grafiky o další podstatný kus. Již v průběhu práce jsem dostával nejrůznější nápady, jak bych mohl tuto technologii využít i v budoucnu, což bylo mým osobním cílem.

Literatura

[1] Beneš, B., Žára, J., Sochor, J., aj.: *Moderní počítačová grafika*. Computer Press, 2004, iSBN 80-251-0454-0.

[2] Grimm, S.: *Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware*. 2005, vedoucí disertační práce Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller.

[3] Kršek Přemysl, K. P.: *Problematika 3D modelování tkání z medicínských obrazových dat. Neurologie pro praxi*, 2005.

[4] Microsoft: *MSDN Direct3D 10 SDK*. 2007, dostupné z WWW: http://msdn2.microsoft.com/en-us/library/bb205066.aspx, [cit. 2007-12-02].

[5] NVIDIA: *CUDA Programming Guide 1.1*. 2007, dostupné z WWW: http://developer.nvidia.com/object/cuda.html, [cit. 2008-05-08].

[6] NVIDIA: *NVidia GeForce 8800 Architecture technical brief*. 2007, dostupné z WWW: http://www.nvidia.com/object/IO_37100.html, [cit. 2007-12-12].

[7] Schroeder, W. J., Zarge, J. A., Lorensen, W. E.: *Decimation of Triangle Meshes*. SIGGRAPH 1992, 1992.

[8] Vacata, J.: *GPGPU – provádění vědeckých výpočtů prostřednictvím grafických karet osobních počítačů*. 2007, vedoucí diplomové práce Ing. Tomáš Oberhuber.

[9] WWW: *CUDPP*. http://www.gpgpu.org/developer/cudpp/, [cit. 2008-3-20].

[10] WWW: *GPGPU*. http://en.wikipedia.org/wiki/GPGPU, [cit. 2007-11-24].

[11] WWW: *OBJ File Format*. http://www.royriggs.com/obj.html, [cit. 2007-12-20].

[12] WWW: *Volume Rendering*. http://en.wikipedia.org/wiki/Volume_rendering, [cit. 2007-12-14].

Seznam příloh

- Dodatek A – Demonstrační aplikace
- Dodatek B – Ukázky z vizualizace polygonálních modelů
- Dodatek C – Obsah DVD

Dodatek A

Demonstrační aplikace

Protože cílem práce bylo vytvoření spíše vektorizační knihovny než kompletní aplikace, je program dodán pouze pro experimentální testování a také jako menší ukázka použití knihovních funkcí. Aplikace obsahuje tedy jen to nejnutnější k úspěšnému převodu volumetrických dat na polygonální síť.

A.1 Instalace

Instalace aplikace je velice snadná a spočívá v pouhém zkopírování spustitelného souboru z média na zvolené místo na disku. Ovšem aplikace vyžaduje instalaci ovladačů grafické karty, které podporují programové rozhraní CUDA 1.1. Abychom zaručili plnou funkčnost, musí konfigurace cílové počítačové sestavy splňovat alespoň minimální velikost dostupné paměti grafických zařízení. Při testování referenčních objemových dat vektorizace spotřebovala kolem 600 MB volné videopaměti.

A.2 Spuštění aplikace

Aplikace je pouze konzolová, spustitelná včetně povinných parametrů z příkazové řádky. Seznam parametrů uvádíme v následujícím přehledu:

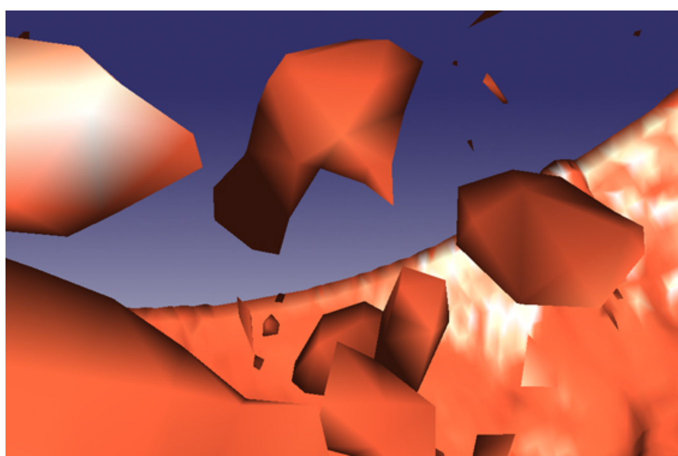
MarchingCubes.exe -i soubor -o soubor -s hodnota [-l soubor] [-h] [-g] [-n]

Přepínače:

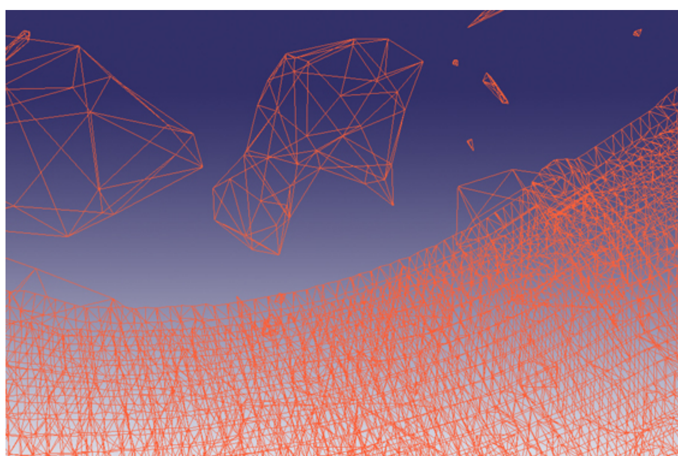
-i soubor	*Vstupní soubor – volumetrická data v RAW formátu*
-o soubor	*Výstupní soubor – polygonální model v OBJ formátu*
-s hodnota	*Hraniční hodnota vektorizace (desetinné číslo $\langle 0, 1 \rangle$, desetinná část musí být oddělená tečkou)*
-l soubor	*Soubor pro zprávy aplikace (pokud je zadán „std“, tak se zprávy vypíší na standardní výstup)*
-h	*Výpis nápovědy*
-g	*Vektorizace se provede na grafickém zařízení, jinak na CPU*
-n	*Invertování normálových vektorů polygonální sítě*

Dodatek B

Ukázky z vizualizace polygonálních modelů



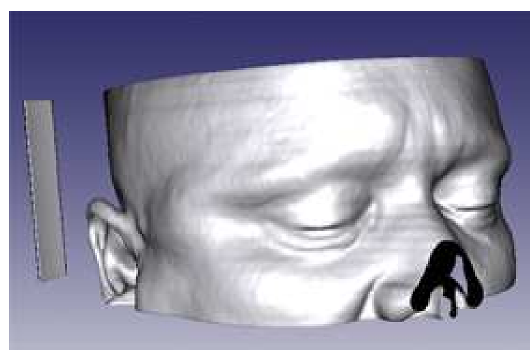
(a) Osvětlený model



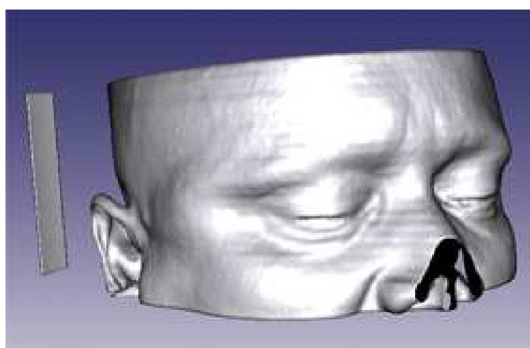
(b) Drátěný model

Obrázek B.1: Detailní pohled na strukturu polygonální sítě

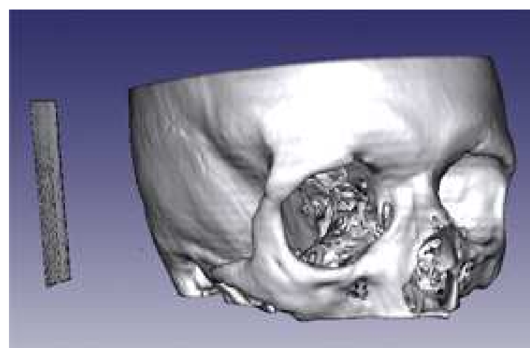
(a) Hranice 0,0



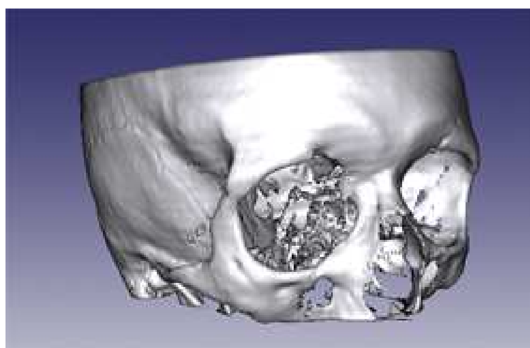
(b) Hranice 0,1



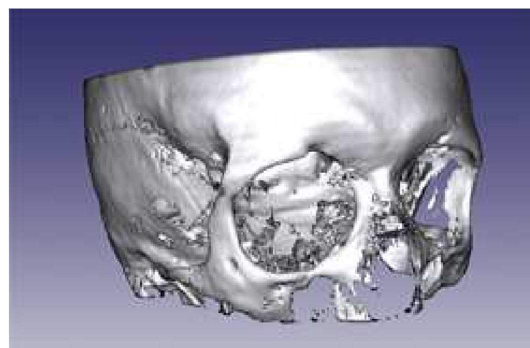
(c) Hranice 0,2



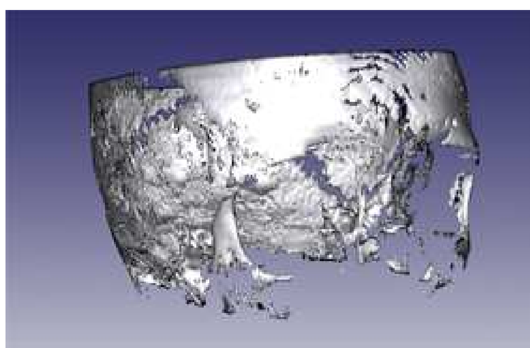
(d) Hranice 0,3



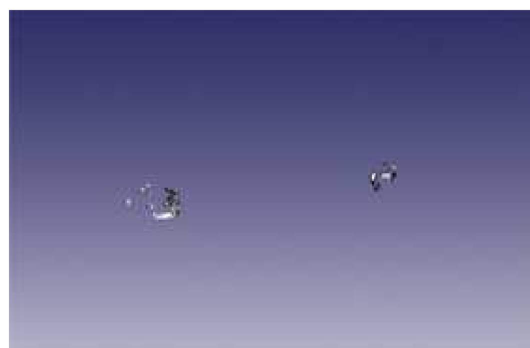
(e) Hranice 0,4



(f) Hranice 0,5

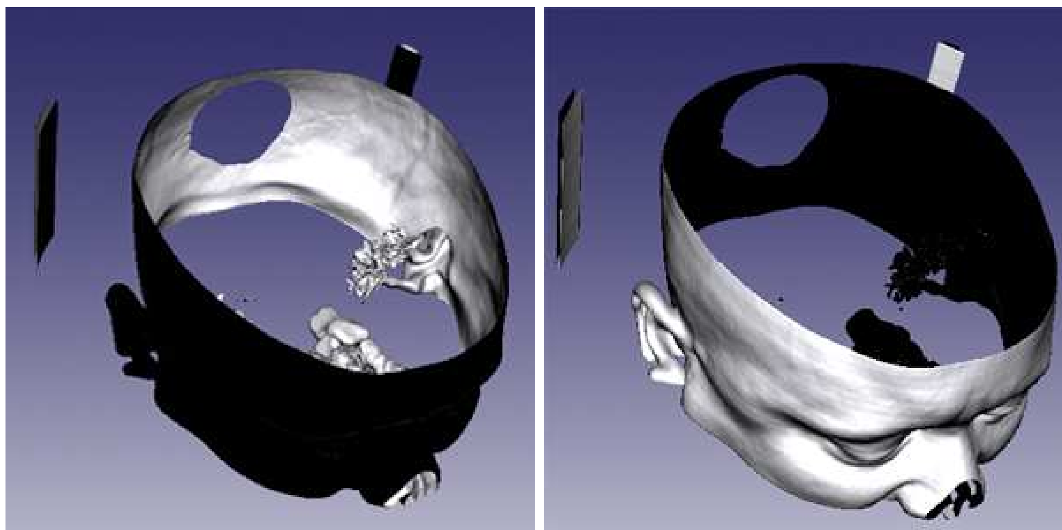


(g) Hranice 0,6



(h) Hranice 0,7

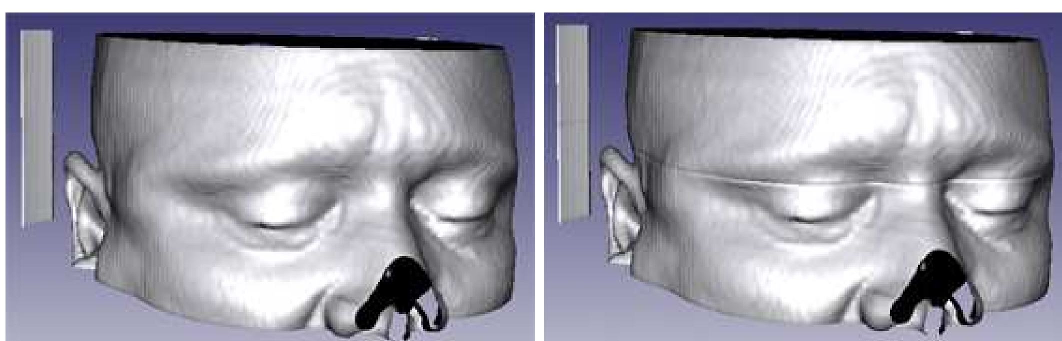
Obrázek B.2: Generované modely v závislosti na hodnotě hranice



(a) Normálový vektor podle metody Central difference gradient

(b) Invertovaný normálový vektor

Obrázek B.3: Orientace normálových vektorů



(a) Model generovaný na jedné grafické kartě

(b) Model generovaný na dvou grafických kartách

Obrázek B.4: Spojení částí modelu z více grafických zařízení

Dodatek C

Obsah DVD

- Demonstrační aplikace
  - Zdrojové soubory včetně knihovny (projekt Visual Studia 2005)
  - Spustitelná verze pro MS Windows XP
  - Projektová dokumentace (generovaná aplikací Doxygen)
- Zpráva diplomové práce
  - Zdrojové soubory (latex)
  - Binární verze (pdf)
- Plakát prezentující diplomovou práci
- Volumetrická data nasnímaných objektů
- Obrázky z vizualizace polygonálních modelů
- Generované polygonální modely v OBJ souborech
- Toolkit a SDK programového rozhraní CUDA verze 1.1
- Vizualizační nástroj pro soubory OBJ (GLC Player)