České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů

Diplomová práce

Rychlé zobrazovací algoritmy pro volumetrická data

Bc. Radomír Vávra

Vedoucí práce: Ing. Vlastimil Havran, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

12. května 2010

Poděkování

Děkuji vedoucímu mé diplomové práce Ing. Vlastimilu Havranovi, Ph.D. za všechny poskytnuté rady a doporučení vedoucí k úspěšnému a včasnému dokončení této práce. Také děkuji svým rodičům za veškerou podporu poskytnutou nejen při psaní diplomové práce, ale i při předchozím studiu. Dále bych rád poděkoval mým prarodičům za pomoc při drobných korekturách textu práce a v neposlední řadě děkuji za pomoc i mé přítelkyni.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 14. 5. 2010 .....

Abstract

Thesis deals with fast rendering of volumetric objects, especially with objects defined by distance functions (level sets). Implementations written in C++ and with SSE instructions utilize recent multicore processors efficiently. Implementations for graphics accelerators uses CUDA C language and OpenCL language. One chapter is focused on description of this languages and highlights factors which influence the final speed of developed applications. Methods for volumetric data storing and rendering are described in detail. Above all thesis focuses on rendering with ray tracing. Implemented algorithms are compared and their image generation times are presented. Most suitable algorithm is modified and rendering library for existing system for immersive virtual reality called Myslbek is created.

Abstrakt

Práce se zabývá rychlým zobrazováním volumetricky reprezentovaných objektů, konkrétně objektů popsaných distanční funkcí. Vytvořené implementace napsané v jazyce C++ a pomocí SSE instrukcí plně využívají moderní vícejádrové procesory. Implementace určené pro grafické akcelerátory používají jazyk C architektury CUDA i jazyk OpenCL. Jedna z kapitol se věnuje popisu těchto jazyků s upozorněním na různé faktory mající vliv na celkový výkon vyvíjených aplikací. Dále je podrobně rozebrána problematika spjatá s uchováváním a zobrazováním volumetrických dat. Práce se zejména zaměřuje na zobrazování dat metodou zpětného sledování paprsků. Implementované algoritmy jsou vzájemně porovnány a jsou prezentovány jejich časy generování obrazu. Úpravou nejvhodnějšího algoritmu je vytvořena zobrazovací knihovna, která je přidána do existujícího systému pro rozšířenou virtuální realitu, do programu Myslbek.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Metody zobrazování počítačových modelů různých objektů lze z hlediska principu generování obrazu rozdělit do dvou základních kategorií. První kategorii tvoří metody rasterizační, druhou metody beroucí v úvahu fyzikální podstatu šíření světla.

U rasterizačních metod musí být pro každý objekt určena povrchová reprezentace. Například pomocí množiny trojúhelníků. Tyto trojúhelníky jsou při vizualizaci transformovány v závislosti na nastavení pomyslné kamery sledující scénu tak, aby se po transformaci kamera nacházela v kladné poloose osy z a sledovala rovinu xy. Trojúhelníky jsou poté promítány do roviny xy, rasterizovány a pomocí paměti hloubky je rozhodnuto, které jejich části budou viditelné a které nikoli. Podrobněji je tento postup popsán v kapitole 3.2.1. Jeho algoritmická složitost je lineárně závislá na množství trojúhelníků, tedy $O(n)$.

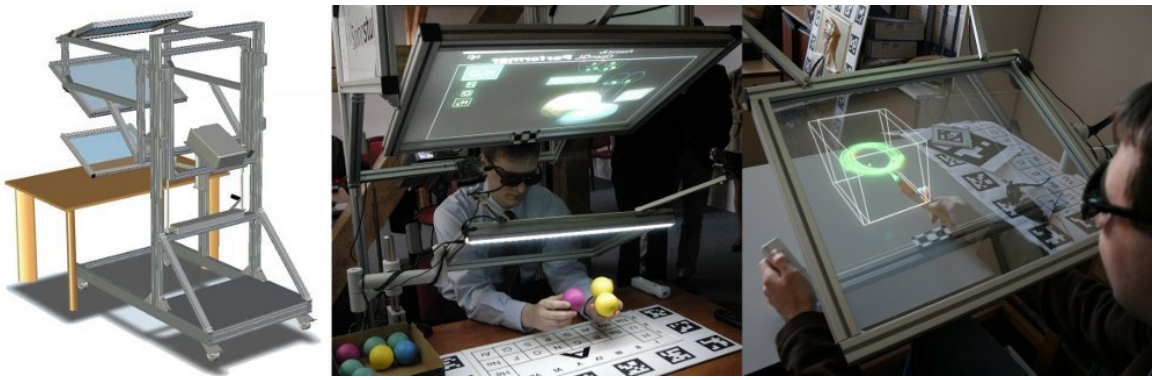
Do druhé kategorie patří metody simulující paprsky světla ve scéně. K jednodušším patří metoda sledování paprsků (ray tracing). Jejím základem je vyslání primárního paprsku od pozorovatele každým pixelem tvořeného obrazu směrem do scény. Zasáhne-li paprsek nějaký objekt, mohou být vyslány paprsky sekundární a stínové, je vyhodnoceno osvětlení zasaženého bodu objektu a tím je určena i barva pixelu. Podrobněji bude tato problematika rozebrána v kapitole 3.2.2. Složitost metody je závislá na složitosti nalezení průsečíku paprsku s nejbližším objektem, která je při použití některé z metod organizace scény $O(\log n)$. Zatímco u rasterizačních metod složitost tolik nezáleží na počtu pixelů obrazu, zde je tomuto počtu přímo úměrná. Celková složitost se proto často zapisuje jako $O(w \times h \times \log n)$, kde w je šířka a h výška obrazu v pixelech.

Z porovnání složitostí uvedených metod je zřejmé, že při zvyšování množství trojúhelníků ve scéně může být dosaženo bodu, od kterého je metoda sledování paprsků rychlejší než metoda rasterizační. Metody pracující s paprsky světla navíc umožňují efektivně simulovat efekty globálního osvětlení, které není možné rasterizačními metodami simulovat vůbec nebo jen s vynaložením vysokého výpočetního výkonu. Proto metody sledování paprsků nacházejí uplatnění i u scén méně rozsáhlých.

Zadáním této diplomové práce je implementovat rychlé zobrazovací algoritmy pro volumetrická data. Jak bylo uvedeno, rasterizační metody vyžadují, aby byla definována povrchová reprezentace vizualizovaného tělesa. Je-li těleso reprezentováno volumetricky, musí být nejprve převedeno. O tom, jak lze převod provést, je pojednáno v kapitole 3.2.1.1. Naproti tomu metoda sledování paprsků umožňuje zobrazit volumetricky reprezentované těleso

pomocí stejného principu, jako když je známa povrchová reprezentace. Není třeba provádět drahou převodní fázi a vizualizace je u rozsáhlejších dat prováděna rychleji. Proto se práce zaměřuje především na metodu sledování paprsků.

1.1 Specifikace cíle



Obrázek 1.1: Model zařízení Spinnstube a fotografie zařízení v chodu s různými aplikacemi.

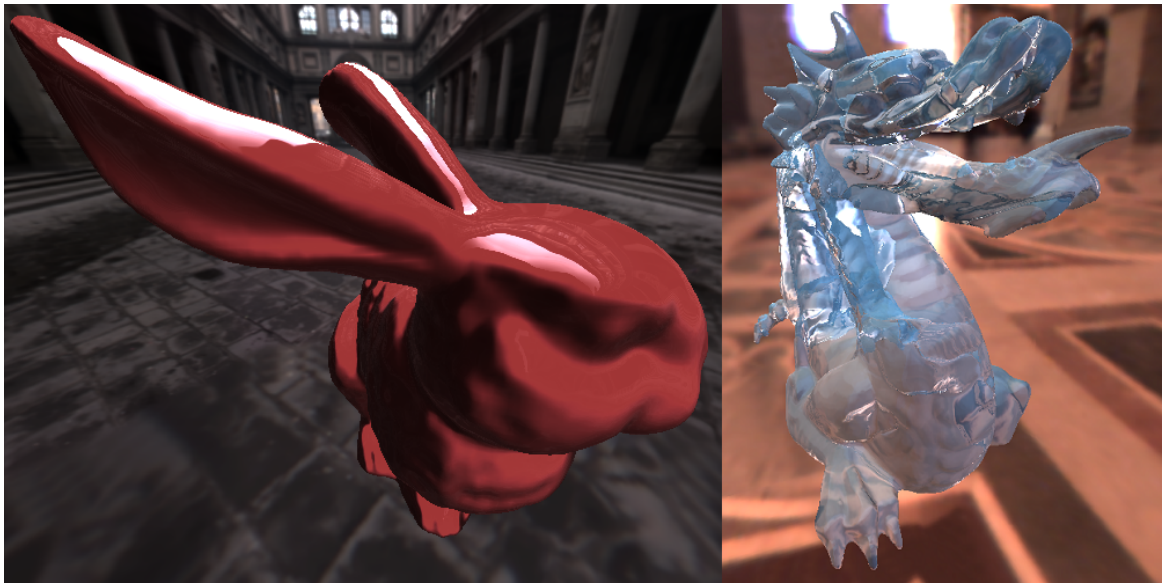
Diplomová práce vychází z bakalářské práce [14], ve které byla metoda sledování paprsků implementována pro vícejádrové procesory. Cílem je zrychlit program použitím algoritmu využívajícího koherence vržených paprsků a optimalizací kódu instrukcemi SSE. Dalším úkolem je upravit program pro grafický akcelerátor s využitím rozhraní CUDA i OpenCL. Nakonec je požadováno implementovat algoritmus do existujícího programu pro rozšířenou virtuální realitu Myslbek, který je provozován na zařízení Spinnstube, jehož model a fotografie s jinými aplikacemi je na obrázku 1.1.

Program Myslbek umožňuje virtuálně obrábět různé objekty pomocí světelného pera, jehož poloha je sledována kamerami. Takovéto obrábění - přidávání nebo odebírání materiálu se dobře realizuje na volumetrických datech. Jak bylo uvedeno výše, rasterizační algoritmy vyžadují nejprve časově náročný převod volumetrické reprezentace na povrchovou. Současná verze programu Myslbek proto zobrazuje pouze orientované body aproximující povrch objektu. Tato metoda je sice rychlá, ale kvalita výstupu je nedostatečná.

Motivací pro tuto diplomovou práci je urychlit metodu sledování paprsků natolik, aby bylo dosaženo dostatečného počtu snímků za sekundu s aktivovanou stereoskopickou projekcí v přijatelném rozlišení a zajistit tak programu Myslbek kvalitní obrazový výstup. Ukázka výstupu testovací aplikace je na obrázku 1.2.

1.2 Struktura práce

Teoretický úvod k obecným výpočtům na grafických kartách přináší kapitola 2. Konkrétněji, architektuře CUDA se věnuje kapitola 2.1 a jazyku OpenCL kapitola 2.2. Kapitola 3 poskytuje souhrn informací souvisejících se zobrazováním volumetrických dat. Co to jsou



Obrázek 1.2: Výsledky metody sledování paprsků pro dva volumetricky reprezentované objekty.

volumetrická data, jak je lze uchovávat a jakým způsobem je možné získat data ve spojitém prostoru i když jsou uložena pomocí diskrétních vzorků popisuje kapitola 3.1. Možnostmi zobrazování se zabývá kapitola 3.2 a kapitoly 3.3, 3.4 a 3.5 popisují problematiku spojenou se zobrazováním volumetrických dat metodou zpětného sledování paprsků. Konkrétně, kapitola 3.3 se věnuje efektivnímu traverzování datových struktur, kapitola 3.4 nalezení průsečíku paprsku s povrchem objektu a kapitola 3.5 výpočtu normálových vektorů v bodech průsečíku. Programy implementované v rámci diplomové práce představuje kapitola 4. Kapitola 5 poté prezentuje obrazy vygenerované implementovanými programy a poskytuje ucelený přehled časů generování těchto obrazů různými metodami s různým nastavením parametrů. Závěr práce je v kapitole 6.

Kapitola 2

Obecné výpočty na grafických kartách

Když se v roce 1981 objevila první grafická karta, nepředpokládal nikdo, že by grafické karty mohly jednou svým výkonem i v obecných výpočtech předstihnout procesor počítače. První karta dokonce ani nepodporovala grafický mód a jediným jejím úkolem bylo postarat se o obrazový výstup textu, aby mohl procesor nerušeně pracovat na jiných výpočtech. Tato myšlenka se osvědčila a grafické karty se začaly dále vyvíjet. Velmi brzo přibyl k textovému módu i mód grafický. Přesto se o výpočet prostorové grafiky zpočátku staral pouze procesor a grafická karta sloužila jen k zobrazení již připraveného obrazu.

3D režim začaly karty podporovat teprve od roku 1995, ale jejich výkon byl nízký. Zlom přinesla v roce 1996 karta Voodoo od 3dfx, která měla výkon vyšší a umožnila vývoj graficky přitažlivých her. Tím podnítila zájem o výkonné grafické karty mezi miliony uživatelů počítačů. Karty byly zpočátku vybaveny pouze fixní funkcionalitou. To neumožňovalo programátorům vytvářet efekty, které karty přímo nepodporovaly. Proto se jednotlivé bloky grafické karty začaly postupem času stávat programovatelnými. Podrobněji je vývoj popsán například v diplomové práci [36].

V současné době jsou grafické karty natolik univerzální, že je možné na GPU provádět téměř jakékoli výpočty. A to nejen uložením vstupních dat do textury a spuštěním programu pro shader napsaný v jazyce GLSL, HLSL nebo Cg jako tomu bylo dříve. Nyní je možné využít architektury CUDA společnosti nVidia, jazyka OpenCL vytvořeného společenstvím Khronos Group nebo například rozhraní DirectCompute společnosti Microsoft.

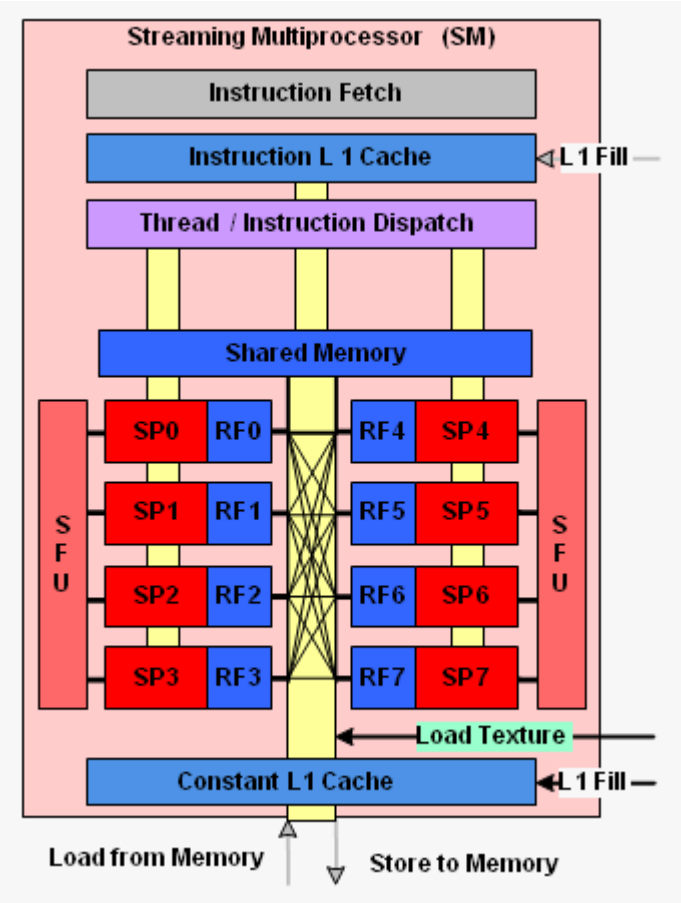
Díky své povaze se grafické karty hodí především pro provádění masově paralelních výpočtů. Hovoří se zde o tzv. architektuře SIMT (Single-Instruction, Multiple-Thread), ve které velký počet vláken programu vykonává v jeden okamžik stejnou instrukci s různými daty. Podobnost označení SIMT s označením SIMD (Single-Instruction, Multiple-Data) tedy není náhodná. Programy jsou na grafických kartách rychlejší než na procesorech, pokud je možno program paralelizovat alespoň do desítek tisíc nezávislých vláken, případně do vláken závislých pouze na omezeném počtu vláken okolních.

2.1 Compute Unified Device Architecture

První verze architektury CUDA vyvinuté společností nVidia byla představena v listopadu roku 2006. Jedná se o ucelenou platformu umožňující vytvářet masivně paralelní programy

spustitelné na grafických kartách nVidia. O této architektuře bylo napsáno již mnoho. Nejvíce informací obsahuje příručka programátora [26]. Přehledný popis architektury v českém jazyce poskytuje diplomová práce [36]. Z toho důvodu následuje stručnější přehled vlastností CUDA zařízení shrnující klíčové vlastnosti architektury. Také jsou uvedena doporučení pro dosažení co nejvyššího výkonu programu a jsou zmíněny aspekty programování, které nemusí být po prostudování příručky programátora zcela zřejmé.

2.1.1 Architektura výpočetních jednotek grafických karet

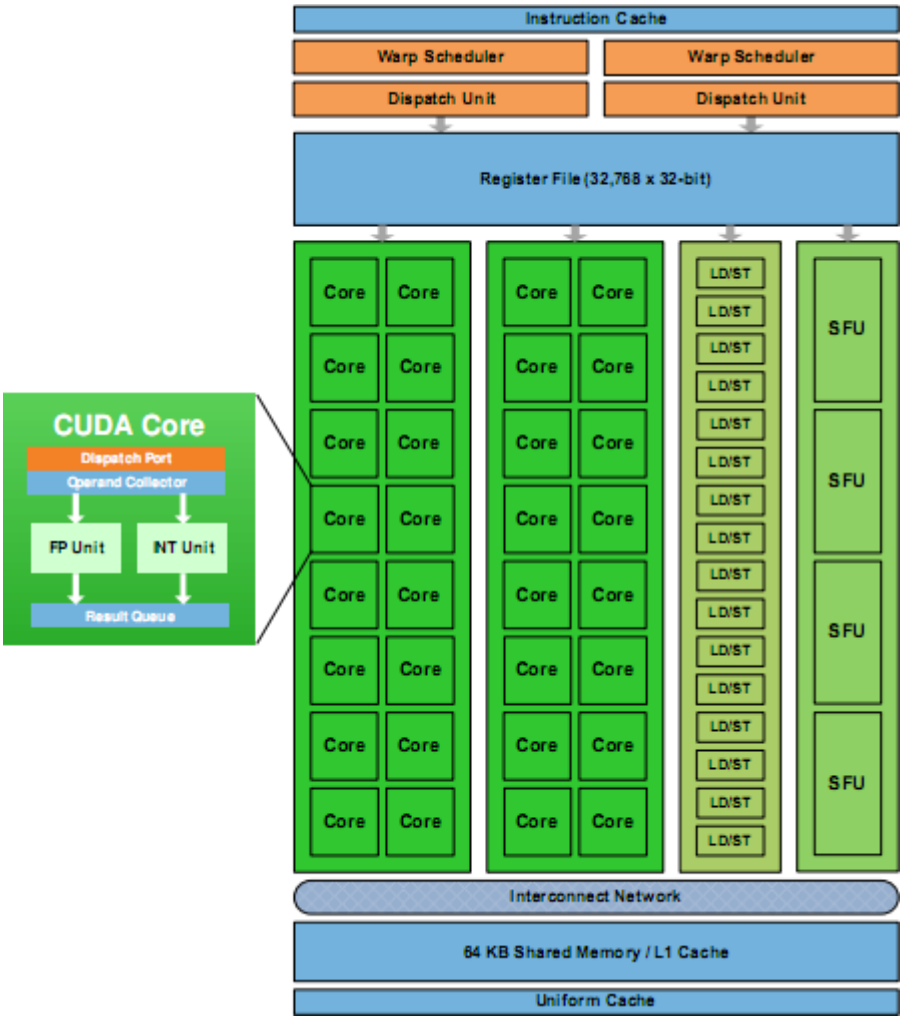


Obrázek 2.1: Schéma multiprocesoru grafických karet s Compute Capability 1.x. Zdroj [32].

Grafické karty nVidia jsou označovány majoritním a minoritním číslem Compute Capability na základě jejich možností pro obecné výpočty. Nejstarší karty podporující architekturu CUDA nesly označení 1.0. Novější karty byly postupně označovány čísly 1.1, 1.2 a 1.3. Nejmodernější karty s čipem Fermi jsou označovány číslem 2.0. Seznam karet s označením obsahuje příloha A a podrobný přehled schopností karet lze nalézt v příloze G příručky programátora [26].

Každá karta se skládá z určitého množství proudových multiprocesorů (Streaming Mul-

tiprocessor – SM). Schéma jednoho multiprocesoru karet 1.x je na obrázku 2.1. Skládá se z 8 procesorů pro celočíselné aritmetické operace a aritmetické operace v plovoucí desetinné čárce s jednoduchou přesností[1], jednoho procesoru pro aritmetické operace v plovoucí desetinné čárce s dvojitou přesností a 2 speciálních funkčních jednotek sloužících například pro výpočty trigonometrických funkcí nebo odmocniny. Na těchto procesorech jsou vykonávány jednotlivé instrukce.



Obrázek 2.2: Schéma multiprocesoru grafických karet s Compute Capability 2.0. Zdroj [23].

Multiprocesor karet 1.x obsahuje pouze jeden dekodér instrukcí. Proto musí být na všech příslušných procesorech daného multiprocesoru v jeden okamžik prováděna stejná instrukce. Pro zamaskování latencí při načítání operandů z paměti jsou vlákna sdružena do tzv. warpů složených z 32 vláken. Tato vlákna se poté na procesorech jednoho multiprocesoru rychle

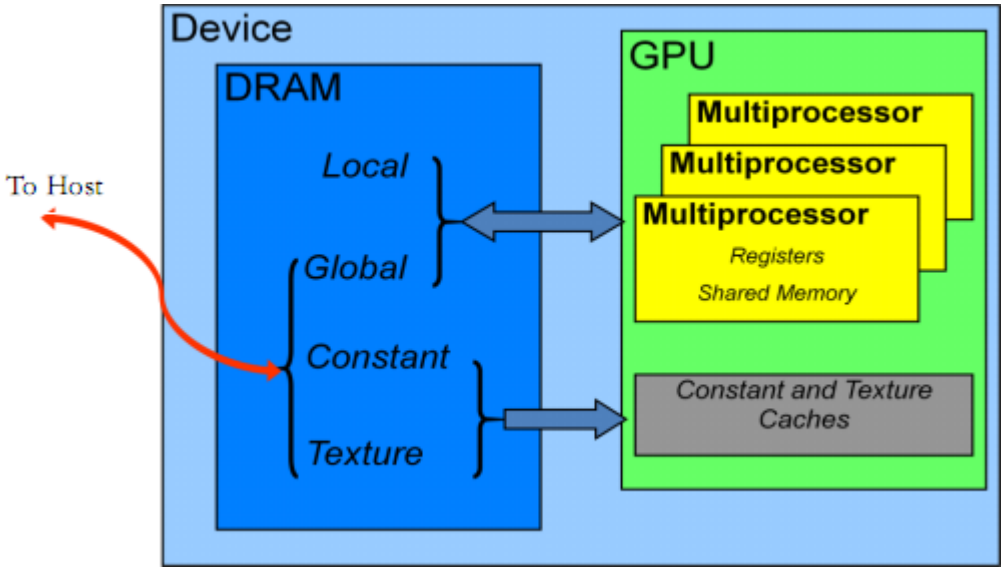
[1] Tyto procesory jsou označovány jako CUDA cores a jejich počet je udáván ve specifikacích prodávaných karet. Počet multiprocesorů lze zjistit vydělením udávaného čísla osmi, má-li karta Compute Capability 1.x nebo vydělením 32, má-li Compute Capability 2.0.

a pravidelně střídají. Díky tomuto principu jsou vlákna jednoho warpu vždy synchronizována.

Z množství jednotek daného typu lze odvodit i propustnost multiprocesoru karet 1.x. Než všechna vlákna warpu dokončí celočíselnou operaci nebo operaci v plovoucí desetinné čárce s jednoduchou přesností trvá 4 hodinové cykly. Výpočet v plovoucí desetinné čárce s dvojitou přesností trvá 32 cyklů, neboť se všechna vlákna musí vystřídat na jediné jednotce. Výpočty ve dvojité přesnosti podporují z karet s majoritním číslem 1 pouze karty 1.3. Provedení instrukce ve speciální funkční jednotce všemi vlákny trvá 16 cyklů.

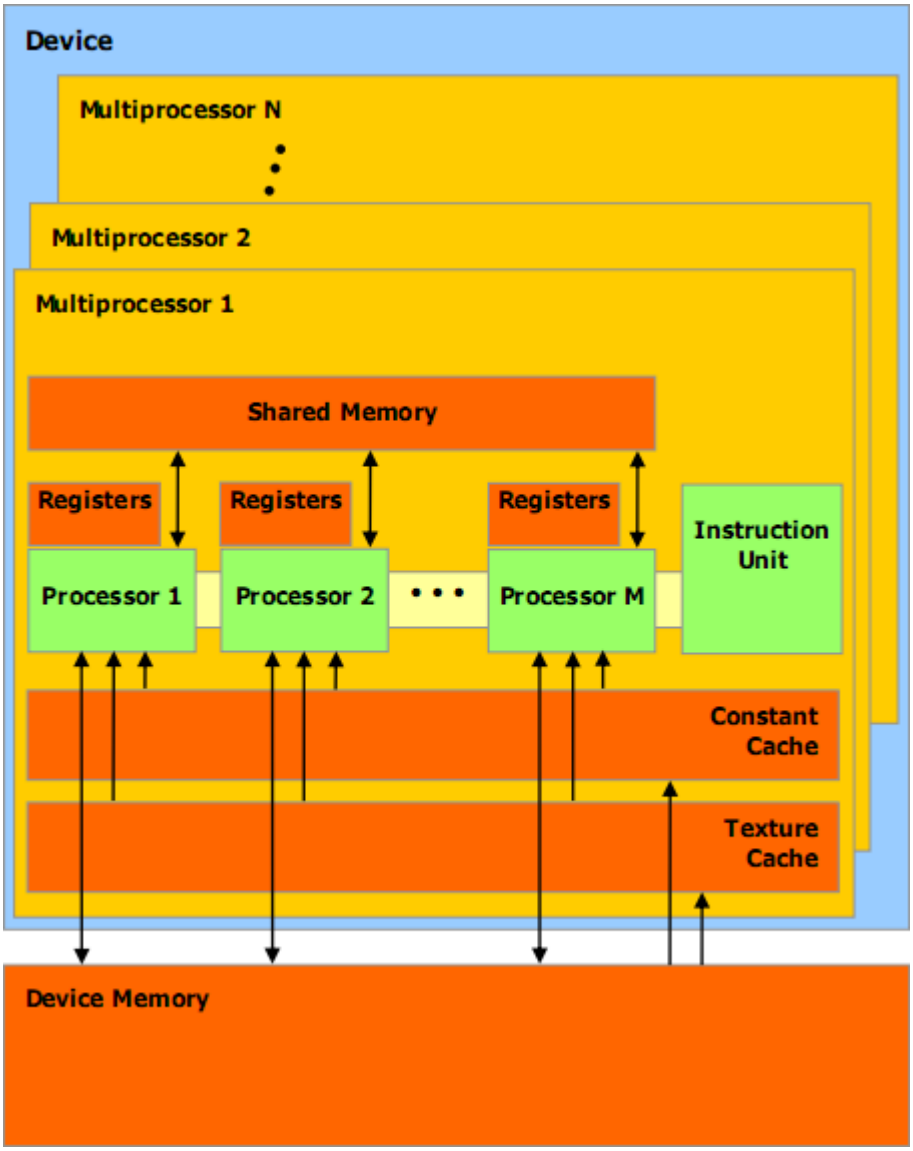
Multiprocesory karet s Compute Capability 2.0 sestávají z 32 procesorů pro celočíselné operace a operace v plovoucí desetinné čárce (s jednoduchou i dvojitou přesností) a 4 speciálních funkčních jednotek. Schéma je na obrázku 2.2. Multiprocesor obsahuje dva dekodéry instrukcí, které pracují současně. Každý s jiným warpem. Z toho důvodu může být warp vykonán pouze na polovině CUDA jader. Při použití operací v plovoucí desetinné čárce s dvojitou přesností může pracovat pouze jeden dekodér. Vlákna warpu vykonají celočíselnou operaci i operaci v plovoucí desetinné čárce ve dvou hodinových cyklech. Vyžaduje-li instrukce speciální funkční jednotku, trvá její provedení osm cyklů.

2.1.2 Organizace paměti



Obrázek 2.3: Paměťové prostory zařízení CUDA. Zdroj [25].

Grafická karta disponuje několika oddělenými paměťovými prostory. Každý se hodí k jinému použití a je nezbytné, aby této problematice programátor dobře rozuměl. Rozdělení jednotlivých druhů pamětí znázorňuje obrázky 2.3 a 2.4. Ty se týká především karet s majoritním číslem 1. Odlišnosti karet 2.0 jsou uvedeny dále v textu.



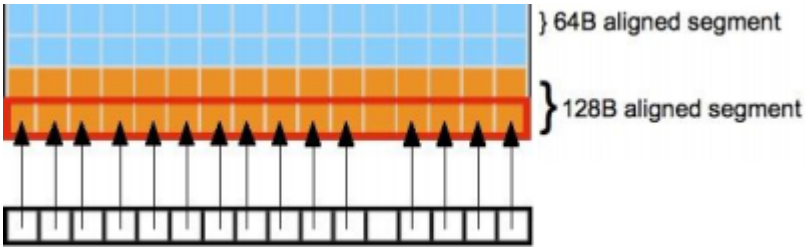
Obrázek 2.4: Organizace paměti grafických karet 1.x. Zdroj [24].

Nejblíže jednotlivým procesorům jsou registry. Jedná se o jednu z nejrychlejších pamětí. Přístup k nim nezabere žádný hodinový cyklus navíc. Obrázek 2.4 naznačuje, že každý procesor obsahuje registry vlastní. Je třeba si uvědomit, že na fyzických procesorech se střídají vlákna stejného, ale i různých warpů. Jelikož si každé vlákno musí uchovávat vlastní lokální proměnné a je třeba mezi jednotlivými vlákny rychle přepínat, jsou registry každého procesoru rozděleny mezi všechna vlákna, která se na tomto procesoru střídají. Tím je značně omezen počet registrů dostupných pro vlákno a počet vláken, která mohou být současně provozována na jednom multiprocesoru. Vždy je třeba nalézt vhodný kompromis. Je-li použit vysoký počet registrů pro vlákno, běží na multiprocesoru pouze nízký počet warpů a nelze efektivně zakrýt latence čtení z pomalejších pamětí. Je-li počet registrů pro vlákno nízký,

musí být lokální proměnné uchovávány v lokální paměti.

Lokální paměť se nachází v hlavní paměti zařízení, jak je patrné z obrázku 2.3. Každé vlákno má vyhrazený vlastní paměťový prostor, do kterého si ukládá lokální proměnné, které není možné uložit do registrů z kapacitních nebo principiálních důvodů. Principiální důvod vzniká, pokud programátor použije pole, jehož indexy mohou být určeny až v době běhu programu, neboť indexovat registry není možné. Programátor by se měl snažit vyhnout použití lokální paměti, neboť přístup k ní vyžaduje stovky hodinových cyklů a u zařízení 1.x není optimalizován použitím cache paměti. Vyšetřit, zda překladač umístil některé proměnné do této paměti lze prozkoumáním PTX kódu. Ten se získá překladem zdrojového kódu s parametrem `-ptx` nebo `-keep`. Proměnné umístěné do lokální paměti jsou deklarovány pomocí klíčového slova `.local` a čteny a zapisovány jsou pomocí `ld.local`, respektive `st.local`. Nicméně některé proměnné mohou být do lokální paměti umístěny až v pozdější fázi překladu. Poté lze již pouze zjistit, kolik lokální paměti pro celý kernel bylo použito pomocí parametru překladače `--ptxas-options=-v` (hodnota lmem).

Každý multiprocesor obsahuje sdílenou paměť. Tato paměť je rozdělena do 16 nezávislých bank (32 u zařízení 2.0). Ke stejné části sdílené paměti mohou přistupovat všechna vlákna bloku spuštěného na multiprocesoru. Přístup ke sdílené paměti je stejně rychlý jako přístup k registrům, pokud každé vlákno půlwarpu (prvních šestnáct vláken nebo druhých šestnáct vláken warpu) přistupuje k jiné paměťové bance nebo pokud všechna vlákna půlwarpu čtou stejnou adresu. Z toho důvodu je třeba snažit se data ve sdílené paměti vždy správně zarovnat. To zejména platí při použití datových typů jiné velikosti než 4 byty.

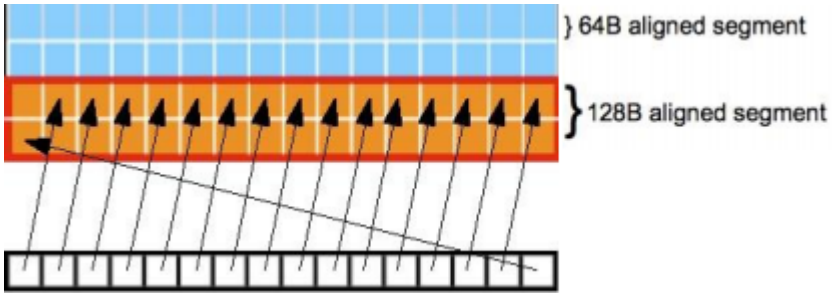


Obrázek 2.5: Zarovnaný přístup do paměti zařízení. Každé vlákno čte odpovídající adresu nebo nečte vůbec. Zdroj [25].

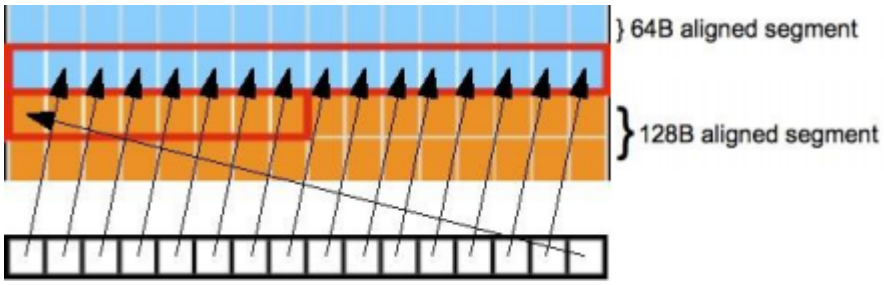
Jak bylo nepřímo uvedeno, přístup k hlavní paměti zařízení vyžaduje stovky hodinových cyklů. Pouze u zařízení 2.0 je přístup k datům optimalizován použitím dvou úrovní cache pamětí. Aby byl výkon co nejvyšší, je vhodné přístupy do paměti zarovnat tak, aby všech 16 vláken půlwarpu četlo data z jednoho zarovnaného segmentu o velikosti 16 slov (64 B), případně 32 slov (128 B). U zařízení 1.0 a 1.1 musí každé vlákno číst adresu, jejíž posun od začátku zarovnaného segmentu o velikosti 16 slov odpovídá indexu vlákna v půlwarpu. Ne všechna vlákna musí z paměti skutečně číst. Příklad je na obrázku 2.5.

U zařízení 1.2 a vyšších stačí, aby všechna vlákna četla data z jednoho segmentu. V rámci segmentu může každé vlákno číst libovolnou adresu. Je tedy i možno, aby více vláken četlo data ze stejné adresy. Obrázek 2.6 ukazuje situaci, ve které všechna vlákna čtou data ze segmentu o velikosti 32 slov. Je-li třeba číst data z více segmentů, adresují se segmenty postupně a vlákna čtou požadované adresy. Vždy se adresují nejmenší postačující segmenty,

jak ukazuje obrázek 2.7.



Obrázek 2.6: Nezarovnaný přístup do jednoho 128-bytového segmentu paměti. U zařízení 1.2 a vyšších bude stačit pouze jedno čtení paměti. U zařízení 1.0 a 1.1 bude provedeno 16 čtení paměti. Zdroj [25].



Obrázek 2.7: Nezarovnaný přístup do dvou 128-bytových segmentů. U zařízení 1.2 a vyšších se nejprve adresuje 64-bytový segment a vlákna 0 až 14 půlwarpu načtou požadovaná data. Poté se adresuje 32-bytový segment a i vlákno 15 uskuteční čtení. U zařízení 1.0 a 1.1 bude provedeno 16 čtení paměti. Zdroj [25].

Při čtení a zápisu do hlavní paměti je tedy nutné pro dosažení vysokého výkonu přístupy zarovnat. Zejména v situacích, kdy je používán datový typ o jiné velikosti než 4 byty nebo v situacích, kdy více vláken bude postupně číst stejné adresy, lze přístup k hlavní paměti optimalizovat využitím paměti sdílené. Vhodné schéma je, aby všechna vlákna bloku zarovnaně načetla data z hlavní paměti do paměti sdílené, provedla se synchronizace vláken a následně mohou všechna vlákna bloku načtená data používat. Jiným příkladem je situace, ve které potřebují vlákna zapsat do hlavní paměti například hodnoty typu `unsigned char`. Přístup všech vláken k paměti by byl pomalý. Lepším schématem je, aby vlákna zapsala hodnotu do sdílené paměti, provedla synchronizaci, přetypovala datový typ na `unsigned int` a první čtvrtina vláken zkopírovala hodnotu ze sdílené paměti do paměti hlavní. Tak se ušetří tři čtvrtiny přístupů k hlavní paměti a navíc může být zápis zarovnaný.

Dále grafické karty obsahují paměť konstant. Ta je mapována do hlavní paměti, ale na rozdíl od ní je cacheována i u zařízení 1.x. Velikost paměti konstant je omezena na 64 KB a každý multiprocesor má cache pro paměť konstant o velikosti 8 KB. Tato cache je stejně rychlá jako registry, pokud všechna vlákna půlwarpu přistupují v jedné instrukci ke

stejné hodnotě. Jinak musí být příkazy serializovány. Z paměti konstant mohou vlákna data pouze číst. Zapis do ní může iniciovat pouze CPU před spuštěním kernelu. Paměť konstant je vhodné použít pro předání většího množství dat kernelu. Parametry kernelu jsou jinak ukládány do sdílené paměti, které je pouze 16 KB na multiprocesor (48 KB u zařízení 2.0) a při spouštění většího množství bloků jí může být nedostatek. Použitím paměti konstant je také ušetřen čas opakovaného nahrávání neměnících se hodnot parametrů do sdílené paměti při opakovaném spouštění kernelu.

Poslední pamětí je texturovací paměť. I ta je mapována do hlavní paměti zařízení a je cacheována. Její velikost je omezena pouze velikostí hlavní paměti a jsou dána omezení pro maximální možnou velikost textur. Textury mohou být jedno, dvou i tří-rozměrné. Uložit data do texturovací paměti je vhodné, pokud nebudou vlákny půlwarpu čtena data ze stejných segmentů, ale stejné vlákno bude postupně číst data ze stejné oblasti. Výhodou textur je, že se zařízení samo postará o adresy směřující mimo oblast textury dle přednastaveného módu (opakování textury, ořez k hraně) a je možno získat lineárně interpolovanou hodnotu z dat v okolí adresovaného místa. Textury je možné adresovat i pomocí normalizovaných souřadnic.

2.1.3 Hierarchie vláken

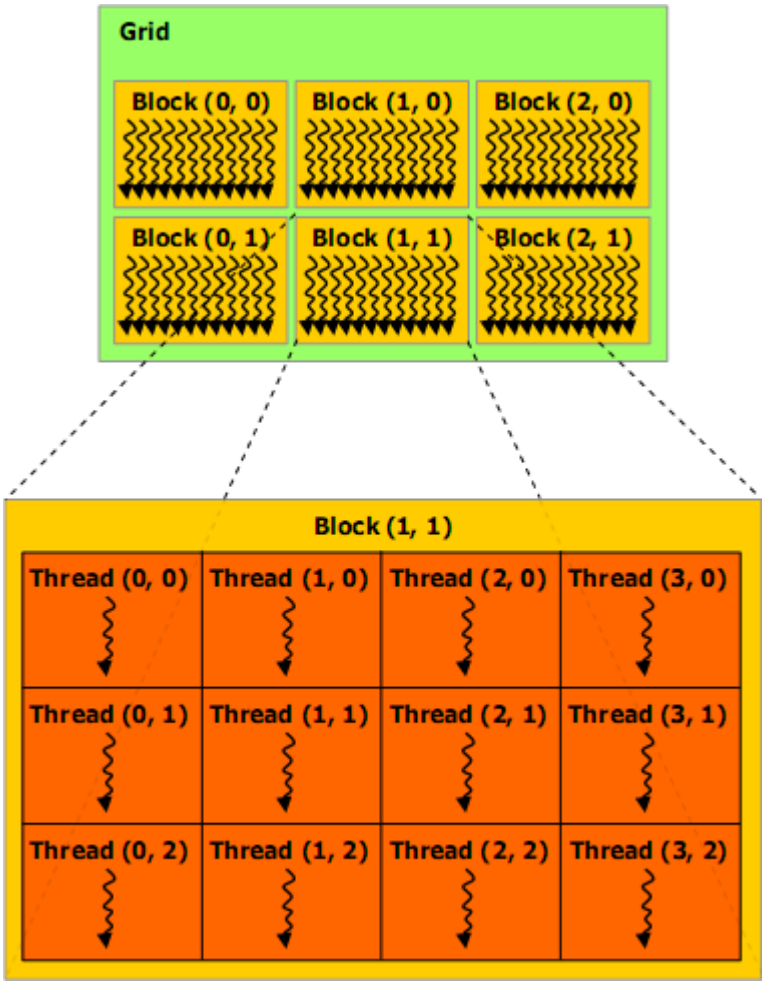
Jak bylo naznačeno výše, jednotlivá vlákna jsou organizována do bloků. Každý blok je vždy vykonáván na jednom multiprocesoru a všechna vlákna bloku lze synchronizovat bariérou. Blok může organizovat vlákna do 1D, 2D nebo 3D pole tak, aby mohla být intuitivně mapována na řešený problém. Počet vláken bloku nesmí překročit maximální povolený počet vláken na blok, který je 512 u karet 1.x a 1024 u karet 2.0. Také nesmí být překročen dostupný počet registrů a množství sdílené paměti na multiprocesoru. Zároveň není vhodné volit bloky příliš malé, jelikož může být současně spuštěno pouze osm bloků na multiprocesor a nebylo by možné maskovat latence pamětí vykonáváním instrukcí jiných warpů.

Vlákna uvnitř bloků jsou organizována do warpů skládajících se z 32 vláken. Z toho důvodu by počet vláken bloku měl být volen jako násobek tohoto čísla. V opačném případě je plýtváno výpočetními jednotkami. Bloky vláken jsou organizovány do mřížky (Grid). Ta může být jedno nebo dvou-rozměrná a její velikost by měla být nastavena tak, aby pokrývala celý problém. Organizace vláken do dvourozměrných bloků a těchto bloků do dvourozměrné mřížky je naznačena na obrázku 2.8.

Každé vlákno může při vykonávání programu zjistit svou pozici uvnitř bloku z tří-složkové vestavěné proměnné. Stejně tak může zjistit pozici bloku uvnitř mřížky a rozměry bloku. Z těchto hodnot lze vypočítat pozici vlákna v mřížce, která u většiny řešených problémů rozhoduje jaká vstupní data vlákno použije. Díky této organizaci vláken bývá velmi snadné mapovat řešené problémy na dostupný hardware.

2.1.4 Model programování

Programy pro grafické karty lze psát v jazyce C s použitím několika rozšíření. Takovýto kód je poté zpracován kompilátorem `nvcc`, který z funkcí označených kvalifikátory `__global__` a `__device__` vytvoří kód pro grafickou kartu a kód určený pro vykonání na hostiteli poskytne ke zpracování kompilátoru jazyka C.

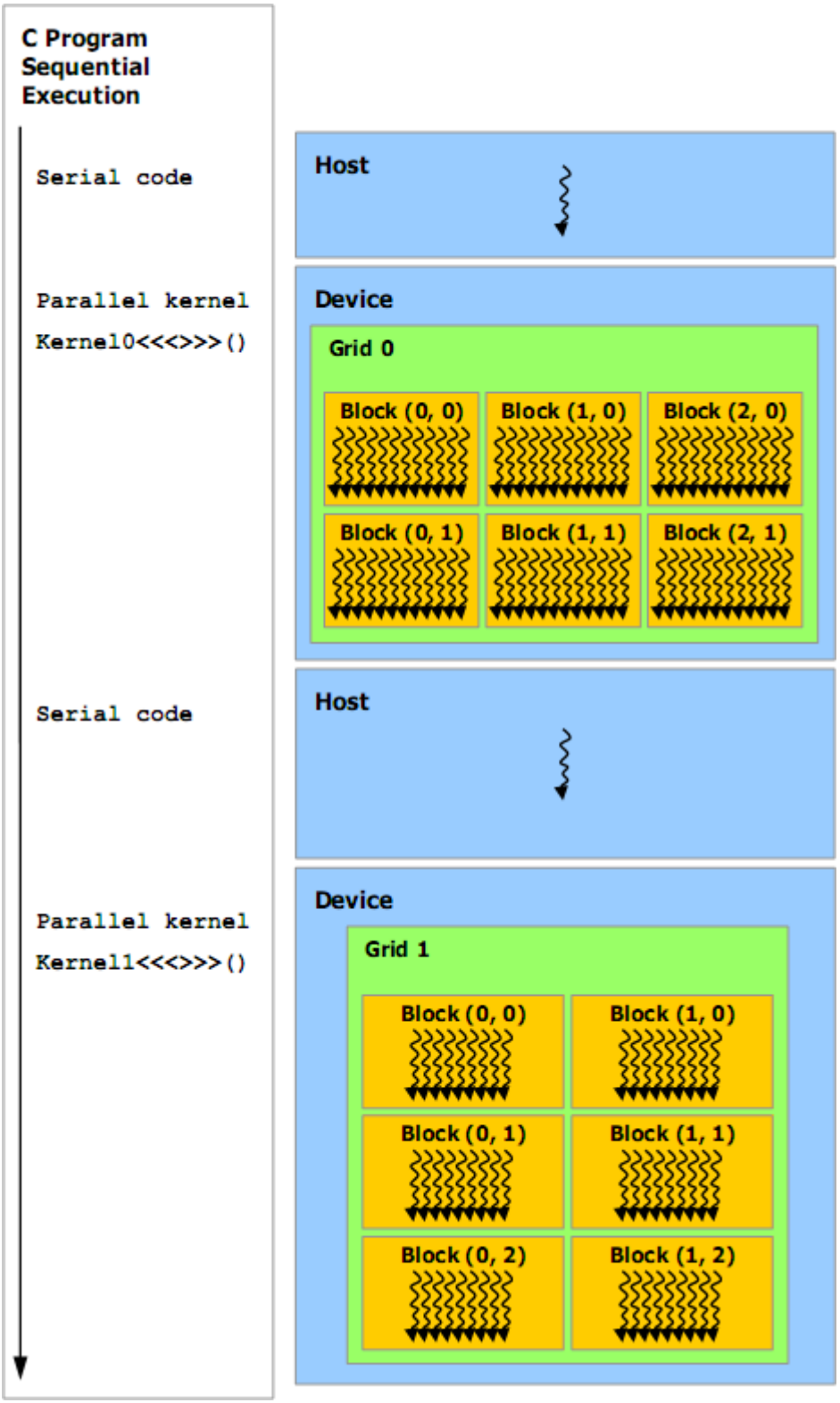


Obrázek 2.8: Organizace vláken do 2D bloků a organizace těchto bloků do 2D mřížky. Zdroj [26].

Obrázek 2.9 představuje model programu, ve kterém je kód vhodnější pro sériové zpracování vykonáván na hlavním procesoru počítače a pro paralelní kód jsou spouštěna výpočetní jádra (kernely) na grafické kartě. Obvykle jsou před spuštěním kernelu ještě kopírována data z hlavní paměti počítače do paměti grafické karty. U real-time aplikací je obvyklý model, ve kterém jsou nejprve nahrána statická data do paměti grafické karty. Následně se ve smyčce střídá provádění na procesoru a grafické kartě a kopíruje se pouze menší množství dat. Při ukončování programu je paměť grafické karty uvolněna.

Před samotným nahráním dat na grafickou kartu je třeba alokovat pro tato data příslušnou paměť. Lineární paměť se alokuje funkcí `cudaMalloc()`. Pro dvou a více-rozměrná pole bývá výhodnější použít funkci `cudaMallocPitch()`, která umožňuje zarovnat každý řádek pole a docílit zarovnaného čtení paměti vlákny bloku. Pokud do pole bude mapována textura, mělo by být použito speciálních funkcí pro alokaci pole (`cudaMallocArray()`, `cudaMalloc3D()`, `cudaMalloc3DArray()`).

Při úspěšné alokaci paměti vrací funkce ukazatel na počátek alokované paměti. Je třeba



Obrázek 2.9: Sériový kód vykonává hostitel – procesor počítače a paralelní zařízení – grafická karta. Zdroj [26].

si uvědomit, že tento ukazatel směřuje do paměti zařízení, ale hodnota byla vrácena hostiteli. Hostitel nemůže do této paměti přímo přistupovat. Pro nahrání dat na zařízení musí použít speciální funkci. Aby mohl program spuštěný na zařízení přistupovat k alokované paměti, je třeba ukazatel na tuto paměť předat kernelu jako vstupní parametr. Jinou možností je zkopírovat hodnotu ukazatele do paměti konstant, jak ukazuje následující příklad:

```
unsigned char               *d_array;
__constant__ unsigned char  *c_array;

void init(unsigned char *array, int count) {

    cudaMalloc((void**)&d_array, count*sizeof(unsigned char));
    cudaMemcpy(d_array, array, count*sizeof(unsigned char),
               cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol((const char*)&c_array, (void *)&d_array,
                       sizeof(unsigned char *));
}

void finalize() {
    cudaFree(d_array);
}

__global__ void inc(int count) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < count)
        c_array[tid]++;
}

void inc_parallel(int count) {
    dim3 threadsPerBlock(256);
    dim3 numBlocks((count + threadsPerBlock.x - 1) / threadsPerBlock.x);
    inc<<<numBlocks, threadsPerBlock>>>(count);
}
```

Kopírování dat z hostitele na zařízení, ze zařízení na hostitele, ale i ze zařízení na zařízení se provádí funkcemi v závislosti na způsobu alokace paměti. Je-li často přenášen větší objem dat z hostitele na zařízení nebo naopak, může být výhodné alokovat paměť hostitele funkcí `cudaMallocHost()`. Přenosy mezi paměťmi jsou poté tzv. zamčeny (page-locked) a jsou prováděny rychleji.

Jak je zřejmé z příkladu výše, kernel je každá funkce deklarovaná s použitím kvalifikátoru `__global__`. Takovouto funkci může volat pouze hostitel s použitím speciální syntaxe `<<<...>>>`, pomocí které se nastavuje konfigurace vláken, která funkci vykonají.

Funkce s kvalifikátorem `__device__` se chovají obdobně jako funkce klasické s tím rozdílem, že jsou vykonávány na zařízení. Mohou být volány pouze z funkcí s kvalifikátorem `__global__` nebo z jiných funkcí s kvalifikátorem `__device__`. Na rozdíl od běžných funkcí

jsou ve výchozím nastavení vždy inlineovány. Tím je sice ve většině případů dosaženo vyššího výkonu, ale vzniká velmi dlouhý program. Vyhnout se inlineování se lze pouze použitím klíčového slova `__noinline__` a nepoužíváním ukazatelů jako vstupních parametrů těchto funkcí. U `__device__` funkcí není podporována rekurze, ať už je inlineování zakázáno či nikoli.

Je-li nutné, aby vlákna bloku spolupracovala, používá se pro jejich synchronizaci bariéra `__syncthreads()`. Programátor musí zajistit, aby byla tato funkce volána skutečně všemi vlákny bloku. Z toho důvodu často nelze zavolat příkaz `return` vlákny, která již nebudou nic počítat. Naopak je nutné psát podmíněný kód, který bude vykonáván pouze pracujícími vlákny a například volání `__device__` funkce obsahující bariéru `__syncthreads()` provést všemi vlákny. V některých případech může program zdánlivě fungovat i pokud všechna vlákna funkci `__syncthreads()` nevykonají. Program může být ale nestabilní a není lehké příčinu nestability odhalit.

2.1.5 Spolupráce architektury CUDA s jazykem OpenGL

Spolupráce s OpenGL vyžaduje, aby bylo před voláním jakékoli jiné funkce pracující s CUDA zařízením zvoleno zařízení, které se má použít. K tomu slouží funkce `cudaGLSetGLDevice()`.

Do verze architektury CUDA 2.3 bylo možno pracovat pouze s OpenGL buffer objekty. Než bude objekt poprvé použit kernelem, je třeba registrovat ho funkcí `cudaGLRegisterBufferObject()`. Před každým použitím se získá adresa objektu v paměti zařízení pomocí `cudaGLMapBufferObject()` a po vykonání kernelu se objekt uvolní pro OpenGL funkcí `cudaGLUnmapBufferObject()`. Nebude-li již objekt kernelem používán, je vhodné volat `cudaGLUnregisterBufferObject()`.

CUDA verze 3.0 podporuje navíc i textury a renderbuffer objekty. Bohužel pouze takové, které obsahují barevné složky. Ne tedy buffer hloubky nebo šablony. Objekty se novými funkcemi registrují v závislosti na typu funkcí `cudaGraphicsGLRegisterBuffer()` nebo `cudaGraphicsGLRegisterImage()`. Před každým použitím se musí provést mapování objektu funkcí `cudaGraphicsMapResources()`. Jestliže je adresa objektu získána pomocí `cudaGraphicsResourceGetMappedPointer()`, zachází se s ním jako s lineární pamětí. Když je získána pomocí `cudaGraphicsSubResourceGetMappedArray()`, zachází se s ním jako s polem. To je možné mapovat i do textury, takže ve výsledku může kernel nativně pracovat i s OpenGL texturami. Odmapování objektu se provádí funkcí `cudaGraphicsUnmapResources()` a zrušení registrace funkcí `cudaGraphicsUnregisterResource()`.

2.2 Open Computing Language

První, na platformě nezávislé, řešení pro vývoj paralelních programů přinesl jazyk OpenCL. Byl vytvořen společenstvím Khronos Group a první verze byla uvolněna v prosinci roku 2008. Na rozdíl od architektury CUDA tedy OpenCL není omezeno na grafické karty nVidia. S příslušnými ovladači je možné překládat a spouštět OpenCL programy i na kartách ATI a také na běžných vícejádrových procesorech i procesorech Cell společnosti IBM.

Kompletní specifikace jazyka OpenCL [16] je zároveň vhodným studijním materiálem pro začínající programátory, neboť obsahuje informace v takovém pořadí, aby se čtenář postupně seznámil se všemi aspekty jazyka. Následuje souhrn nejdůležitějších vlastností.

2.2.1 Model vykonávání programu

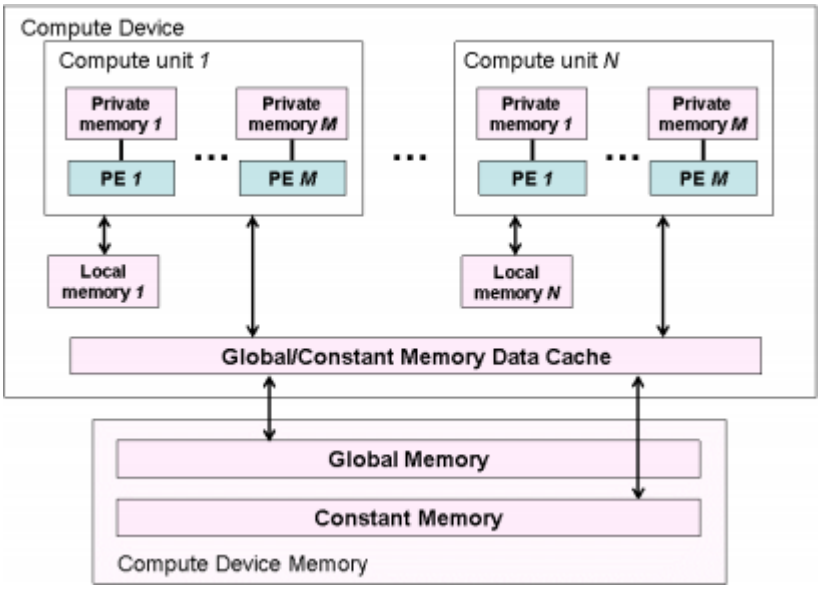
Navzdory své univerzalitě se OpenCL od architektury CUDA příliš neliší. Pro organizaci vláken je pouze použita odlišná terminologie. Vlákna jsou nazývána pracovními jednotkami (work-item) a bloky pracovními skupinami (work-group). Pracovní skupiny jsou uspořádány do n-dimenzionální oblasti (NDRange). Ta může být jedno, dvou i tří-rozměrná. Na rozdíl od definice velikosti mřížky u architektury CUDA, kde se udává počet bloků v každé dimenzi, určuje programátor velikost oblasti jako počet pracovních jednotek v každé dimenzi. Rozměry oblasti přesto musí být přesnými násobky velikosti pracovní skupiny.

Každá pracovní jednotka může zjistit svou polohu uvnitř oblasti dotazem na své globální ID v jednotlivých dimenzích a svou polohu uvnitř pracovní skupiny dotazem na lokální ID v jednotlivých dimenzích. Lze se také dotázat na počet pracovních skupin a velikost pracovní skupiny.

Jedna pracovní jednotka nemusí být nutně vykonávána na jednom procesoru proudového multiprocesoru grafické karty. Pokud algoritmus často pracuje například s datovým typem `float4`, může kompilátor jazyka OpenCL rozhodnout, že bude každá pracovní jednotka vykonávána na čtyřech procesorech. Programátor může toto rozhodnutí ovlivnit použitím kvalifikátoru `__attribute__((vec_type_hint(<typen>)))` u definice kernelu. Ten naznačuje kompilátoru na jaký datový typ by měl být přeložený program přizpůsoben. Může se jednat o libovolný skalární nebo vektorový datový typ. Například `int`, `char4` nebo `float16`.

2.2.2 Organizace paměti

Každá pracovní jednotka má přístup ke čtyřem druhům paměti, které se liší velikostí, rychlostí a možnostmi přístupu. Konceptuální schéma paměti je na obrázku 2.10.



Obrázek 2.10: Konceptuální schéma paměti OpenCL zařízení. Zdroj [16].

Privátní paměť slouží pro uložení lokálních proměnných. Každá pracovní jednotka má vlastní. Při vykonávání programu na grafické kartě nVidia bude privátní paměť mapována na CUDA registry a CUDA lokální paměť. Privátní proměnné se deklarují uvedením kvalifikátoru `__private` před datový typ proměnné.

Pro alokování proměnných sdílených všemi pracovními jednotkami pracovní skupiny je určena paměť lokální. Je to obdoba CUDA sdílené paměti. Nebude-li se sdílené paměti dostávat, může být část lokální paměti uložena i v CUDA paměti zařízení. Velikost pracovních skupin by tedy neměla být omezena nedostatkem sdílené paměti jako tomu může být u CUDA bloků. V OpenCL programech se pro deklaraci těchto proměnných užívá kvalifikátor `__local`.

Globální paměť je ekvivalentem CUDA paměti zařízení. K této paměti mohou přistupovat všechny pracovní jednotky. Není ovšem možné ji využít k předávání informací mezi pracovními jednotkami různých pracovních skupin, neboť OpenCL neposkytuje žádný mechanismus jak synchronizovat pracovní skupiny. V závislosti na možnostech použitého zařízení může být přístup k této paměti cacheován. Při deklaraci globálních proměnných je používán kvalifikátor `__global`.

Neměnná data lze před spuštěním kernelu uložit do paměti konstant. Tato paměť je mapována do globální paměti. V praxi se ukazuje, že její velikost je na grafických kartách nVidia omezena stejně jako CUDA paměť konstant. Konstanty se deklarují s pomocí kvalifikátoru `__constant`.

2.2.3 Konzistence paměti a synchronizace pracovních jednotek

Bez použití synchronizačních bariér není zaručeno, že se bude stav pamětí jevit shodně všem pracovním jednotkám. V rámci jedné pracovní jednotky konzistence zaručena je.

Lokální anebo globální paměť se stane konzistentní všem pracovním jednotkám příslušné pracovní skupiny po použití synchronizační bariéry pracovní skupiny. Neexistuje žádný způsob jak zajistit konzistenci globální paměti mezi pracovními jednotkami z různých pracovních skupin.

Synchronizační bariéra se volá funkcí `barrier()` s parametrem `CLK_LOCAL_MEM_FENCE` nebo s parametrem `CLK_GLOBAL_MEM_FENCE` v závislosti na tom, kterou paměť je třeba dostat do konzistentního stavu. Oba parametry lze i kombinovat. Synchronizační funkci musí volat buď všechny, nebo ji nesmí volat žádná pracovní jednotka synchronizované pracovní skupiny.

2.2.4 OpenCL program

OpenCL programy je nejlepší psát do samostatných souborů, neboť jsou překládány speciálním překladačem. K překladu dochází často až za běhu hostitelského programu, protože je nutné OpenCL program přizpůsobit platformě, na které bude provozován.

Funkce vykonávaná na OpenCL zařízení a volaná z hostitelského programu se, stejně jako u CUDA programů, nazývá kernel. Musí být označena klíčovým slovem `__kernel`. Mívá větší počet parametrů, neboť se jedná o jedinou cestu, jak předat kernelu samotná data nebo ukazatele na data. Kernel může volat další funkce, které se již žádným speciálním kvalifikátorem neoznačují.

2.2.5 Příprava k vykonání programu

Před samotným vykonáním kernelu musí být provedena série inicializačních kroků. Dostupné OpenCL platformy se získají funkcí `clGetPlatformIDs()`. Na těchto platformách je poté třeba získat seznam zařízení zavoláním `clGetDeviceIDs()`. Jedním z parametrů funkce je typ požadovaného zařízení, který může mít hodnotu například `CL_DEVICE_TYPE_CPU` nebo `CL_DEVICE_TYPE_GPU`. Podrobnější informace o jednotlivých zařízeních lze získat funkcí `clGetDeviceInfo()`.

Pro zvolenou platformu a na ní zvolená zařízení je následně potřeba vytvořit kontext. K tomu slouží funkce `clCreateContext()`. Kontext je určen ke správě objektů jako jsou fronty příkazů, paměťové objekty, programy a kernely a k vykonávání kernelů na specifikovaných zařízeních.

Fronty příkazů slouží pro sběr a následné vykonání operací s paměťovými objekty, programy a kernely. Operace jsou vykonávány v tom pořadí, ve kterém byly do fronty zadány. Po zadání příkazů je řízení vráceno hostujícímu programu a ten může dále pracovat nezávisle na stavu příkazů ve frontě. Je možné vytvořit více front příkazů. Ty jsou poté nezávislé. Pracuje-li více front se stejnými objekty, bývá nutné příkazy ve frontách synchronizovat vložením bariéry do front. Fronta příkazů se vytváří funkcí `clCreateCommandQueue()`. Vždy je nutné alespoň jednu frontu vytvořit.

Paměťové objekty se dělí na buffer objekty a image objekty. Buffer objekty jsou jednodimenzionální pole prvků. Tyto prvky mohou být skalárního datového typu (int, float), vektorového datového typu (int4, float8) nebo uživatelem definovanou strukturou. Buffer objekty se vytvářejí funkcí `clCreateBuffer()`. Pro čtení, zapsání a kopírování prvků bufferu se používají funkce `clEnqueueReadBuffer()`, `clEnqueueWriteBuffer()` a `clEnqueueCopyBuffer()`.

Image objekty se užívají k uložení dvou nebo tří-rozměrných textur, frame-bufferů nebo obrázků. Pro práci s image objekty v kernelu je nutné použít speciálních funkcí, data nelze číst přímo jako u buffer objektů. Image objekty se vytvářejí funkcí `clCreateImage2D()` nebo `clCreateImage3D()`. Číst, zapisovat nebo kopírovat data lze pomocí `clEnqueueReadImage()`, `clEnqueueWriteImage()` a `clEnqueueCopyImage()`. Funkcemi `clEnqueueCopyImageToBuffer()` a `clEnqueueCopyBufferToImage()` je možné kopírovat data z image objektu do buffer objektu a naopak.

Objekt s programem se vytváří funkcí `clCreateProgramWithSource()`, jejímž parametrem je zdrojový kód programu v podobě textového řetězce. K převedení zdrojového souboru s OpenCL programem na řetězec je možné využít například funkci `oclLoadProgSource()`, kterou lze nalézt v NVIDIA GPU Computing SDK. Je-li k dispozici binární verze programu pro zvolené zařízení, může být objekt s programem vytvořen funkcí `clCreateProgramWithBinary()`.

Je-li k dispozici pouze zdrojový kód programu, musí být přeložen pomocí funkce `clBuildProgram()`. Jejím argumentem jsou mimo jiné i parametry předané kompilátoru, které mohou mít významný vliv na výkon aplikace. Například parametr `-cl-mad-enable` dovolí nahradit instrukci násobení následovanou instrukcí sčítání jedinou instrukcí vykonanou za stejný počet hodinových cyklů jako jiné elementární instrukce. Bude-li výsledný program spouštěn na grafických kartách nVidia, je vhodné omezit maximální počet registrů pro vlákno a umožnit tak vykonávat na každém multiprocesoru větší počet warpů. Slouží k tomu parametr `-cl-nv-maxrregcount=N`.

Po vytvoření objektu programu je třeba vytvořit kernel objekty pro všechny kernely, které budou spouštěny. Kernel objekt zapouzdřuje zvolenou funkci programu definovanou s použitím kvalifikátoru `__kernel`. Vytváří se funkcí `clCreateKernel()`. Poté je možné přiřazovat hodnoty jednotlivým parametrům kernelu. Používá se k tomu funkce `clSetKernelArg()`.

2.2.6 Spuštění kernelu

Samotné spuštění kernelu, respektive naplánování spuštění kernelu do fronty příkazů, se provádí funkcí `clEnqueueNDRangeKernel()`. Jako parametry funkce je třeba mimo jiné zadat velikost oblasti práce a velikost pracovní skupiny, o kterých bylo pojednáno v kapitole 2.2.1.

Je-li třeba při opakovaném spouštění kernelu předávat mu aktualizovaná data, lze to provést několika způsoby. Pro menší množství dat je možno volat vždy funkce `clSetKernelArg()` a aktualizovat tak hodnoty některých parametrů předávaných kernelu. U většího množství dat je vhodnější vytvořit buffer objekt, ve kterém budou data uložena. Kernelu se poté předá ukazatel na tento buffer jako argument, který se nemění. Data v buffer objektu se před každým spuštěním kernelu aktualizují voláním funkce `clEnqueueWriteBuffer()`.

Kapitola 3

Zobrazování volumetrických dat

Náplní této kapitoly je ucelený popis problematiky zobrazování volumetrických dat. Nejprve je vysvětleno, co jsou volumetrická data a co představují. Jsou uvedeny možnosti interpolace diskrétních vzorků volumetrických funkcí a jsou popsány datové struktury vhodné pro uchování těchto vzorků. Také je uvedeno, z jakého důvodu je vhodnější uchovávat volumetrické funkce pomocí vzorků dat, než použitím množiny funkčních předpisů.

Následující část této kapitoly popisuje možnosti zobrazování povrchů objektů zaznamenaných pomocí vzorků volumetrické funkce. Práce se zaměřuje na metody sledování paprsků a proto je dále popsána problematika efektivního traverzování datových struktur uchovávajících vzorky dat, nalezení průsečíku paprsku s povrchem objektu a výpočet normálového vektoru v bodě průsečíku. Jsou vysvětleny i metody, které celý proces generování obrazu výrazně urychlují.

3.1 Volumetrická data

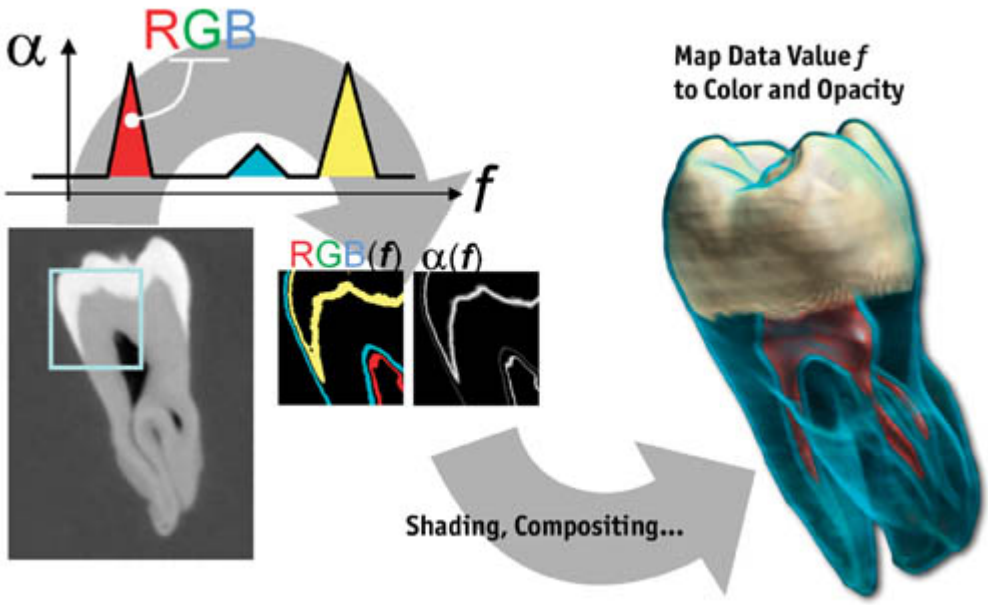
Ve volumetrické grafice jsou objekty nejčastěji uchovávány pomocí diskrétních polí vzorků dat. Tato data mohou vyjadřovat buď hustotu objektu v daném místě, vzdálenost tohoto místa od nejbližšího povrchu objektu nebo pouze to, zda se v daném místě objekt nachází. Díky této reprezentaci je na rozdíl od reprezentace povrchové uchována i vnitřní struktura objektu.

S využitím volumetrické reprezentace lze velmi snadno zjistit, zda se daný bod prostoru nachází uvnitř objektu nebo vně. Dále je možné vizualizovat například vnitřní anatomickou strukturu z lékařských dat nebo modelovat deformace 3D objektů. Tato reprezentace je také vhodná pro úpravy objektů virtuálním řezáním, krájením, trháním, odebíráním nebo přidáváním materiálu.

3.1.1 Pole hustot

Uchovává-li diskrétní pole vzorků hustotu materiálu v daném místě prostoru $\rho(x, y, z)$, nabízí se dvě různé možnosti vizualizace těchto dat. První způsob využívá algoritmus vrhání paprsků (ray casting). Vržený paprsek prochází prostorem, v určených intervalech testuje hustotu, akumuluje barvu a snižuje zbývající průhlednost. Procházení trvá, dokud zbývající

průhlednost neklesne na nulu. Je-li místům s nižší hustotou přiřazena větší průhlednost, vyniknou ve výsledném obrázku více místa s vyšší hustotou. Jak ukazuje obrázek 3.1, mapování hustoty na průhlednost a barvu je možné volit libovolně a docílit tak pěkných a užitečných vizualizací volumetrických dat.



Obrázek 3.1: Vizualizace volumetrických dat uchovávajících hustotu metodou vrhání paprsků. Zdroj [13].

Druhou možností je zobrazení pouze zvolené iso-plochy s hustotou ρ_{iso}. K tomu lze použít buď převedení této iso-plochy na trojúhelníky, které musí být následně rasterizovány nebo využít metody sledování paprsků. Narozdíl od prvního způsobu nebude paprsek akumulovat barvu podél celé dráhy šíření, ale pouze nalezne nejbližší bod, ve kterém platí $\rho(x, y, z) = \rho_{iso}$.

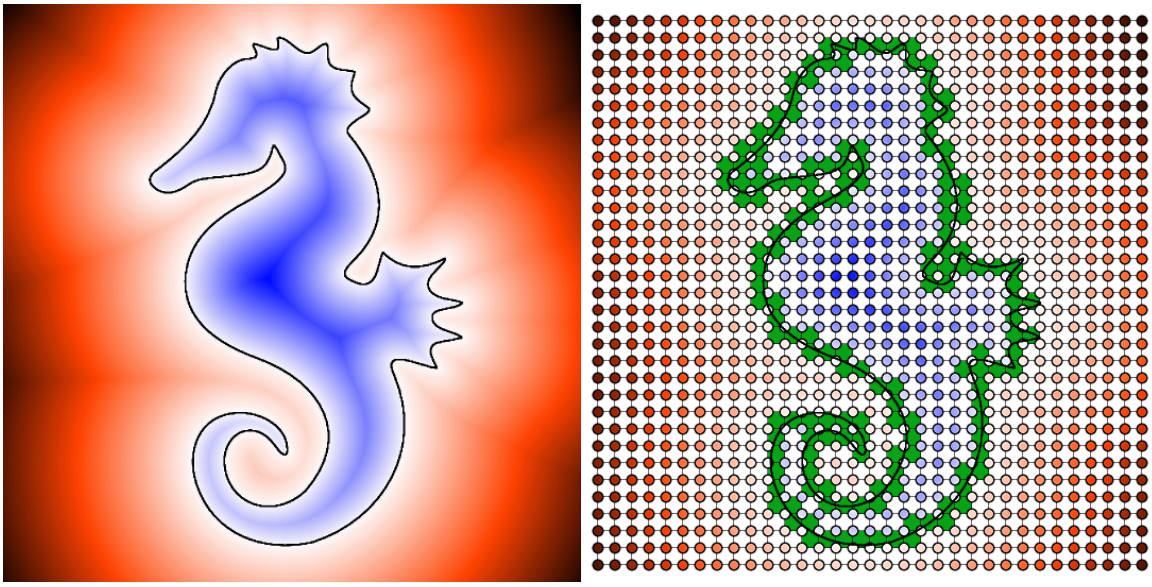
Uchovávání objektů v polích hustot je vhodné pouze u objektů složených z různých materiálů s různými hustotami nebo u objektů s hustotou měnící se plynule. U objektů majících v celém svém objemu stejnou hustotu to význam nemá, neboť se data uvnitř nebo vně vůbec nemění a naopak na povrchu dochází ke skokové změně hodnot.

Maximální uchovatelná změna je omezena frekvencí vzorkování a na povrchu tak vznikají terasovité nebo zubaté hrany objektů. Tyto artefakty lze sice filtrovat, ale hrany jsou poté málo ostré a zanikají drobné detaily. Kvalita objektu je také závislá na jeho orientaci v poli.

3.1.2 Distanční funkce

Možnost uchování detailů na povrchu objektu i s použitím relativně nízké vzorkovací frekvence a s možností libovolné orientace objektu v diskrétní mřížce přinášejí distanční funkce. Ty byly poprvé představeny v článku [9]. Vzorky distanční funkce uložené v diskrétním poli jsou zde nazývány distanční mapou. Tyto vzorky vyjadřují vzdálenost daného místa prostoru

od nejbližšího povrchu objektu. Pro snadné rozlišení vnitřku a vnějšku objektu jsou vzdálenosti uvnitř objektu podle konvence navíc opatřeny záporným znaménkem. Vizualizace dvou-rozměrné distanční funkce a distanční mapy jsou na obrázku 3.2.



Obrázek 3.2: Vizualizace dvou-rozměrné spojité distanční funkce (vlevo) a vzorky této funkce uložené v diskrétním poli nazývaném distanční mapa nebo distanční pole. Obrázek vpravo zobrazuje navíc binární pole indikující přítomnost povrchu ve voxelu.

Distanční mapy přináší několik užitečných vlastností. Směr gradientu distančního pole je shodný se směrem normály nejbližšího povrchu. Iso-plocha distančního pole s nulovou hodnotou je povrchem uchovávaného objektu. Změny hodnot distančního pole poblíž povrchu jsou na rozdíl od pole s uloženými hustotami malé. Objekty, které mají v různých svých částech různé hustoty, lze zaznamenat použitím pole hustot a distanční mapy současně a využít tak výhod obou reprezentací.

Pro vytvoření distanční mapy objektu je třeba znát jeho analytický popis, polygonální reprezentaci nebo volumetrickou reprezentaci, ze které lze získat povrch objektu. Jednotlivé objekty by měly být v distanční mapě uloženy daleko od sebe, aby nedocházelo k interferenci, nebo by pro každý objekt měla být použita vlastní distanční mapa.

Aby byly zachovány detaily objektu a zároveň nebylo používáno nadbytečné množství vzorků, je při přípravě distanční mapy potřeba vhodně volit vzorkovací frekvenci. Na rozdíl od polí uchovávajících hustoty objektu není nutno používat vyšší vzorkovací frekvence v blízkosti povrchu objektu, ale v místech s vysokým zakřivením povrchu. Problém nastává především v případech, kdy povrch obsahuje velké ohyby a vznikají tak místa stejně vzdálená od dvou bodů povrchu. Další problém způsobují objekty s výraznými hranami, neboť tak vznikají oblasti s nelineárním distančním polem. Tyto nepravidelnosti v distančním poli je naštěstí lehké detekovat. Pokud je distanční mapa přibližně lineární, je velikost gradientu vypočítaného centrální diferencí (viz. kapitola 3.5.2) konstantní a rovna dvojnásobku vzdálenosti mezi vzorky. V místech poblíž singularit hran nebo rohů se hodnota gradientu značně

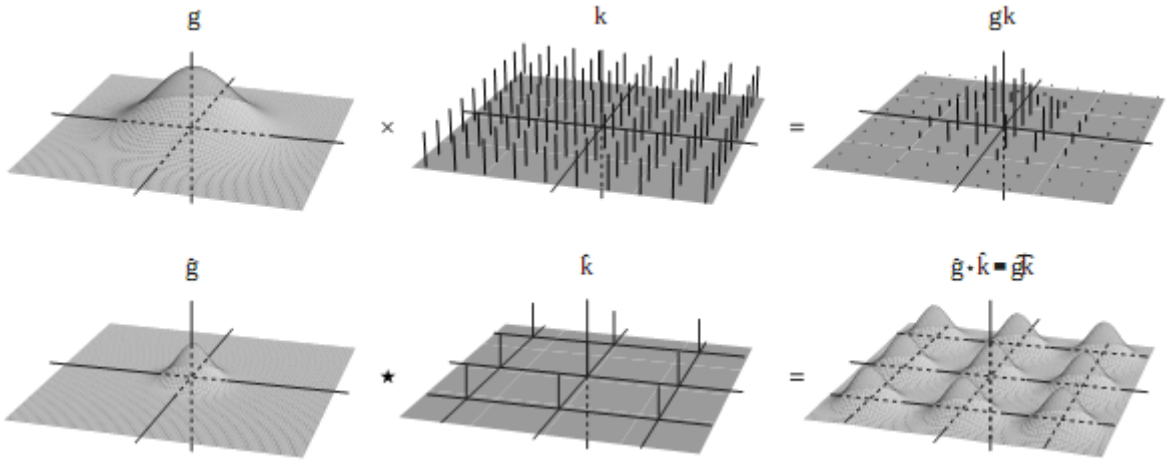
liší od této hodnoty a nepravidelnost je tak detekována. V takovém případě je vhodnější použít sofistikovanější metodu pro určení normálového vektoru povrchu.

3.1.3 Interpolace vzorků

Při práci se vzorky distanční funkce nebo hustoty je pro zjištění hodnoty v zadaném bodě prostoru nutno mezi těmito vzorky interpolovat. Většina běžně používaných interpolačních metod je ekvivalentní konvoluci vzorků s rekonstrukčním filtrem. Filtrů existuje velké množství a pro každou konkrétní aplikaci je nutné zvolit filtr správný. Porovnáním v počítačové grafice nejčastěji používaných filtrů pro interpolaci vzorků uložených v rovnoměrné pravoúhlé tří-dimenzionální mřížce se zabývá článek [20].

Problematika interpolace je často opomíjena a je používáno pouze interpolace trilineární. Ač je tato metoda rychlá, nepřináší tak kvalitní výsledky jako některé jiné. Cena použití sofistikovanější metody by mohla být vyvážena možností uchovávat nižší počet vzorků se zachováním stejné kvality výstupu rekonstrukční funkce.

Vzorek spojité funkce lze definovat jako Diracův impuls s konečnou hodnotou. Fourierovou transformací dvou-dimenzionální mřížky impulsů $k(x, y)$ s frekvencí f_x v ose x a f_y v y je mřížka impulsů $\widehat{k}$ s periodou f_x v x a f_y v y. Označme $g(x, y)$ signál, který bude vzorkován a gk vzorkovaný signál. Poté Fourierova transformace vzorkovaného signálu $\widehat{gk}$ je rovna konvoluci Fourierova obrazu původního signálu $\widehat{g}$ s $\widehat{k}$, jak ukazuje obrázek 3.3.



Obrázek 3.3: Dvou-dimenzionální vzorkování v prostorové oblasti (nahoře) a ve frekvenční oblasti. Zdroj [20].

Vzorkováním dojde k vytvoření kopie $\widehat{g}$ v každém bodě $\widehat{k}$. Kopie v počátku je označována jako základní spektrum a ostatní kopie jako alias spektra. Aby se spektra vzorkovaného signálu nepřekrývala, musí být nejvyšší frekvence signálu nižší než polovina frekvence vzorkovací. Tato nejvyšší frekvence je nazývána *Nyquistova frekvence* a značí se f_N.

Pokud základní spektrum není překryto alias spektry, lze rekonstruovat $\widehat{g}$ vynásobením $\widehat{gk}$ funkcí $\widehat{h}$. Funkce h, inverzní transformace $\widehat{h}$, je nazývána rekonstrukční filtr. Ideální rekonstrukční filtr h je definován tak, že $\widehat{h}$ má hodnotu jedna uvnitř kruhu o poloměru f_N a

hodnotu nula jinde. Takovýto filtr nelze v praxi implementovat, neboť má nekonečný rozvoj v prostorové oblasti.

Pokud je do rekonstrukce zahrnuta i část alias spektra kvůli nedokonalosti rekonstrukčního filtru, vzniká postaliasing. Naopak prealiasing je způsoben vzorkováním signálu s nižší než dvojnásobnou frekvencí nejvyšší frekvence signálu. V obou případech se složky frekvencí původního signálu objevují v rekonstruovaném signálu na jiných frekvencích. Vyhlazení (smoothing) signálu je zapříčiněno odebráním prudkých změn v signálu prostorovým průměrováním. Doznívání (ringing, overshoot) vzniká v místech nespojitosti signálu. Výše definovaný ideální filtr způsobuje výrazné doznívání v každé nespojitosti. Naopak filtr s pozvolnou hranicí tento nepříznivý efekt potlačí.

Filtry analyzované v článku [20] se dělí do dvou kategorií na *separabilní* a *sféricky symetrické*. Separabilní filtr lze zapsat jako součin filtrů pro jednotlivé dimenze. Do této kategorie patří trilineární filtr, kubické filtry, do kterých lze zahrnout i B-spline filtr a Catmull-Rom filtr, Gaussův filtr, cosine bell filtr a okénkový sinc filtr. Dále sem patří filtry optimální vůči propouštěnému pásmu, které se dají definovat množinou diskrétních filtrů vytvořených tak, aby bylo minimalizováno vyhlazení signálu. Jejich nevýhodou je ale ignorování problému postaliasingu. Hodnota sféricky symetrických filtrů závisí pouze na vzdálenosti od počátku. Patří mezi ně rotační verze cosine bell filtru a sféricky symetrický ekvivalent okénkového sinc filtru.

Za účelem vyhodnocení kvality filtrů definuje článek [20] metriky pro nejdůležitější parametry: vyhlazení signálu, postaliasing a doznívání. Na jejich základě lze filtry vzájemně porovnat. B-spline nejvíce vyhlazuje hrany avšak postaliasing je nejnižší. Naopak Catmull-Rom produkuje mnohem méně vyhlazení, ale velmi slabě potlačuje postaliasing. U kubických filtrů je třeba mezi těmito dvěma parametry dělat kompromis v závislosti na požadavcích konkrétní situace. Okénkové sinc filtry poskytují lepší postaliasing a vyhlazení než kubické filtry. Avšak jsou mnohem dražší kvůli své velikosti. Trilinearní filtr poskytuje horší výsledek než ostatní filtry, ale operační složitost je naopak velmi nízká. Jak bylo očekáváno, filtry optimální vůči propouštěnému pásmu vykazují nízké vyhlazení hran za cenu slabého potlačení postaliasingu. Jejich operační složitost je vysoká a nedoporučuje se proto používat je k univerzální rekonstrukci. Z dalších filtrů je významný pouze cosine bell s poloměrem 1,5, který s nižší složitostí produkuje podobné výsledky jako B-spline.

Při volbě filtru je vždy nutno dělat kompromis mezi vyhlazením hran, postaliasingem, doznívánim a operační složitostí filtru. Ukazuje se, že v časově kritické aplikaci jako je nalezení průsečíku paprsku s volumetricky reprezentovaným povrchem a výpočtu normálového vektoru takovéhoto povrchu kvalita trilineární interpolace dostačuje a je bohatě vyvážena rychlostí tohoto filtru.

3.1.4 Datové struktury

Distanční funkci nebo funkci hustoty geometrických objektů je možno definovat pomocí jedné nebo více rovnic. Skládáním základních geometrických objektů lze vytvořit i objekty složitější. Takovýmto skládáním se zabývá disciplína zvaná konstruktivní geometrie těles (CSG), ve které jsou tělesa popsána stromovou strukturou. Listy tohoto stromu představují jednotlivé geometrické objekty a vnitřní uzly určují operace, které se provedou s potomky.

Těmito operacemi mohou být různé množinové operace a prostorové transformace. Více o konstruktivní geometrii těles poskytuje například kniha [21].

Pro vytvoření komplexnějších objektů pomocí konstruktivní geometrie těles by mohlo být zapotřebí obrovského množství základních objektů. Paměťové nároky pro uchování takového objektu by byly velmi vysoké a i zobrazování by bylo velmi zdlouhavé, neboť je vždy třeba procházet celou stromovou strukturou. U složitějších objektů je výhodnější uchovat distanční funkci nebo funkci hustoty pomocí vzorků těchto funkcí.

3.1.4.1 Uniformní mřížky

Nejjednodušší použitelnou datovou strukturou pro uchování vzorků funkcí je uniformní pravoúhlá ekvidistantní mřížka. Takováto mřížka je složena z $P \times Q \times R$ stejných krychlových buněk – voxelů. Vzorky funkce jsou uloženy v rozích těchto buněk, což znamená, že každé dvě sousední buňky sdílí čtyři vzorky. Velikost pole pro uchování vzorků musí proto být $(P+1) \times (Q+1) \times (R+1)$ hodnot. Uložení vzorků dvou-dimenzionální distanční funkce do uniformní pravoúhlé ekvidistantní mřížky o velikosti 32×32 voxelů je znázorněno na obrázku 3.2.

Z pole vzorků lze snadno zjistit, zda nějakým konkrétním voxelem prochází povrch. Stačí ověřit, že alespoň jeden ze vzorků v rozích tohoto voxelu leží uvnitř objektu (jeho hodnota je záporná) a alespoň jeden vně objektu (jeho hodnota je kladná). Aby tato informace nemusela být pokaždé zjišťována, je vhodné uchovávat ji v binárním poli o velikosti $P \times Q \times R$ voxelů. I toto zachycuje obrázek 3.2, kde jsou voxely obsahující povrch zvýrazněny zeleně.

Pokud by buňky měly tvar obecného kvádru, nepředstavovalo by to pro běžné zobrazovací algoritmy překážku. Stejně tak pokud by vzorky v ose nebyly rozmístěny rovnoměrně. Bylo by pouze třeba poupravit traverzační algoritmus a interpolační funkci. Buňky tvaru kvádru poskytují například některé přístroje pro magnetickou rezonanci, neboť mají v ose posunu skenovacího zařízení jiné rozlišení než v ostatních osách. Kvalita reprezentace objektu je poté v jednotlivých směrech odlišná.

V některých případech může být výhodné zvolit jiný tvar buněk než je krychle, respektive kvádr. Měl by být však takový, aby se takovýmito útvary dal vyplnit celý prostor. Ve 2D se tedy může jednat například o trojúhelník nebo hexagon a ve 3D například o tetrahedron nebo rhombic dodecahedron. Traverzovací a interpolační funkce je ale pro takovéto útvary potřeba značně upravit. Nepravoúhlými ani neekvidistantními mřížkami se tato práce nebude dále zabývat. Výhodou uniformních mřížek je konstantní časová složitost náhodného dotazu, nevýhodou naopak kubická složitost paměťová.

3.1.4.2 Víceúrovňové mřížky

Má-li být vždy zobrazována pouze iso-plocha s nulovou hodnotou distanční funkce, je možné omezit uchovávání vzorků ve větších vzdálenostech od povrchu objektu. Za tím účelem je třeba organizovat prostor využitím hierarchických datových struktur. Takovouto strukturou může být i víceúrovňová mřížka.

Její využití pro uchování volumetrických dat navrhl Parker a kol. [27] a pro uložení distanční mapy ji později použil Bridson [6]. Nejspodnějším patrem víceúrovňové mřížky je výše zmíněné binární pole indikující přítomnost povrchu. V patře nad ním je binární pole

o velikosti $\frac{P}{k} \times \frac{Q}{k} \times \frac{R}{k}$, jehož každá buňka byla vytvořena seskupením $k \times k \times k$ buněk z nižší úrovně. Pokud alespoň jedna z buněk nižší úrovně obsahuje povrch objektu, musí i buňka vyšší úrovně indikovat jeho přítomnost. Pater je možno vytvořit různé množství. Je třeba si však uvědomit, že při náhodném přístupu musí být všechna tato patra projita, pokud se poblíž daného místa nalézá povrch. Proto se v praxi často užívá mřížek dvouúrovňových, jejichž spodní úroveň je nazývána jemnou mřížkou a vrchní hrubou mřížkou. Kromě počtu úrovní ovlivňuje výkon i počet buněk seskupených do jedné buňky vyšší úrovně.

Hlavním přínosem dvouúrovňových mřížek pro algoritmy sledování paprsků je značné omezení počtu traverzačních kroků na paprsek. Neindikuje-li hrubá mřížka přítomnost povrchu, prochází paprsek tímto prázdným prostorem velmi rychle. Pokud přítomnost povrchu indikuje, musí paprsek procházet buňky jemné mřížky a v těch, které obsahují povrch, hledat průsečík paprsku s povrchem. Podrobněji tuto problematiku popisuje kapitola 3.3.

Při zavedení takto popsané dvouúrovňové mřížky není nezbytné upravovat pole vzorků distanční funkce. Pro snížení paměťových nároků je však možné uchovávat vzorky pouze v těch místech, kde buňky hrubé mřížky indikují přítomnost povrchu. Za tím účelem musí být vytvořeno příslušné množství polí o velikosti $(k+1) \times (k+1) \times (k+1)$ vzorků. Obdobně lze ušetřit paměť definováním jemné mřížky pouze v místech, kde hrubá mřížka indikuje přítomnost povrchu.

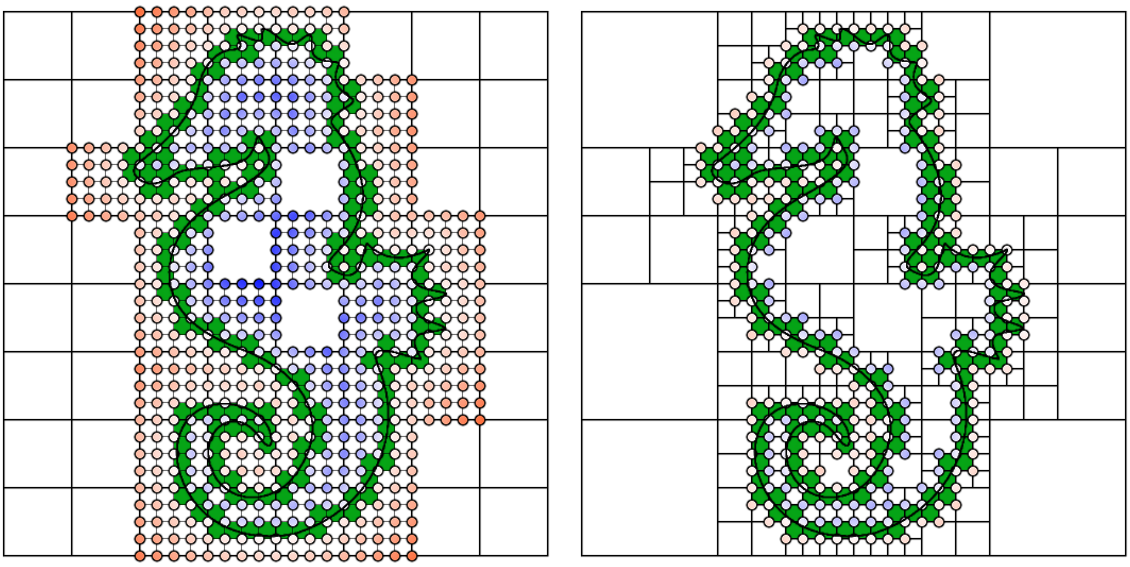
Vyžaduje-li řešený problém možnost rychle zjistit, zda se dotazovaný bod prostoru nalézá uvnitř nebo vně objektu a nejsou-li v zájmu ušetření paměti vzorky distančního pole zaznamenány pro celý prostor, je výhodné namísto binárních mřížek indikujících přítomnost povrchu použít ternární mřížky indikující, zda je buňka celá uvnitř objektu, vně objektu nebo jí prochází povrch.

Víceúrovňové mřížky dovolují ušetřit paměť oproti mřížkám uniformním. Přitom operační složitost náhodných dotazů zůstává $O(1)$ a rychlost přístupu ovlivňuje pouze počet úrovní mřížky. Především ale tyto mřížky dovolují omezit počet traverzačních kroků algoritmů sledování paprsků. Dvourozměrné distanční pole z obrázku 3.2 zaznamenané pomocí vzorků do dvouúrovňové mřížky je na obrázku 3.4 vlevo.

3.1.4.3 Oktalové stromy

Jinou hierarchickou datovou strukturou pro uložení distančního pole může být oktalový strom. Každý jeho vnitřní uzel dělí prostor na osm podprostorů. Obsahuje-li takovýto podprostor povrch objektu, je dále dělen. Jinak může strom pouze uchovávat informaci, zda je podprostor celý uvnitř objektu nebo vně objektu. U klasických oktalových stromů je dělení zastaveno až v momentě, kdy je velikost podprostorů shodná s velikostí buněk mřížky uchovávající distanční pole a vzorky distanční funkce jsou uloženy pouze v těchto listech stromu. Tímto postupem vytvořený strom pro dvou-dimenzionální distanční funkci z obrázku 3.2 je na obrázku 3.4 vpravo.

Článek [7] navrhuje použít oktalové stromy pro uložení distančních polí vzorkovaných adaptivně v závislosti na lokálních detailech (Adaptively sampled Distance Fields – ADFs). Každý uzel stromu představující buňku prostoru obsahuje distanční hodnoty v rozích, ukazatel na rodiče a ukazatele na potomky. K rozdělení buňky obsahující povrch dojde pouze tehdy, pokud není distanční pole dobře aproximováno trilineární interpolací hodnot v jejích



Obrázek 3.4: Vzorky distanční funkce z obrázku 3.2 uložené pomocí dvouúrovňové mřížky (vlevo) a kvadrantového stromu.

rozích. Díky tomu mohou být v místech, kde je tvar objektu relativně hladký, použity větší buňky a oproti klasickým oktalovým stromům je dosaženo mnohem vyšší komprese.

Je-li dána distanční funkce, existuje více možností jak vytvořit ADF. Jednou z nich je postup zdola nahoru. Na začátku je vytvořeno pravidelně vzorkované distanční pole s konečným rozlišením a je postaven úplný oktalový strom. Poté probíhá seskupování. Začíná se s nejmenšími buňkami stromu a postupně se jde k větším. 8 buněk se seskupí právě tehdy, když žádná z nich nemá potomka a navzorkované vzdálenosti všech osmi buněk mohou být rekonstruovány z hodnot jejich rodiče s určitou tolerancí chyby.

Při postupu shora dolů jsou nejprve spočteny distanční hodnoty kořene. Buňky jsou poté děleny podle daných pravidel. Například dělení může být zastaveno, pokud daná buňka neobsahuje povrch objektu, pokud obsahuje povrch, ale splňuje nějaký predikát, nebo pokud je dosaženo maximální úrovně dělení. Řídících predikátů může být mnoho. Například obdobně jako u postupu zdola nahoru, pokud distanční vzdálenosti uvnitř buňky lze spočítat z hodnot uložených v buňce s maximálně určenou chybou.

Článek [7] uvádí, že je ve 2D vůči klasickým kvadrantovým stromům dosahováno redukce více než 20:1 se zachováním stejné kvality rekonstruovaných dat a hrany a rohy lze rekonstruovat dokonce přesněji. Oktalové stromy uchovávající klasické i adaptivně vzorkované distanční funkce zachovávají všechny výhody distančních polí, jako je jednoduché provádění booleovských operací, snadné určení, zda se zadaný bod nachází uvnitř či vně objektu a možnost výpočtu gradientu funkce. Uložením vzorků distanční funkce ve všech úrovních oktalového stromu lze efektivně renderovat objekty s různým stupněm LOD. Jejich paměťové nároky jsou pouze $O(n^2)$ oproti nárokům mřížek $O(n^3)$, kde n je počet vzorků distanční funkce v jedné dimenzi. Nevýhodou je vyšší operační složitost náhodného dotazu $O(\log n)$ a složitost stavby stromu $O(n^3 \log n)$.

3.2 Možnosti zobrazení

Jak bylo uvedeno v úvodu této práce, lze metody zobrazování objektů rozdělit z hlediska principu generování obrazu do dvou kategorií. Jedná se o kategorii rasterizačních metod a kategorii metod počítajících s paprsky světla.

3.2.1 Rasterizační metody

Rasterizačními metodami je možné zpracovat pouze objekty, které mají určenu ploškovou reprezentaci povrchu. Pro volumetricky reprezentované objekty je třeba nejprve tuto reprezentaci získat. K tomu slouží metoda Marching Cubes popsaná dále v této kapitole.

Zásadním problémem rasterizace plošek je určení jejich viditelných částí. Existuje několik algoritmů, které tuto problematiku řeší. Patří mezi ně například malířův algoritmus nebo algoritmus dělení obrazovky. Více o nich lze nalézt v knize [29]. K nejefektivnějším metodám však patří algoritmus pracující s pamětí hloubky, který implementují i všechny moderní grafické karty podporující práci s 3D grafikou.

Před samotnou rasterizací plošek je třeba provést jejich transformaci tak, aby se pomyslná kamera nacházela v kladné části osy z a dívala se na rovinu xy, do které se plošky promítají. Transformaci lze provést jedinou maticí o velikosti 4x4 prvků a je tedy velmi rychlá.

Následně jsou plošky rozděleny na trojúhelníky a ty se rasterizují některou z metod uvedenou v knize [29]. Při rasterizaci jsou trojúhelníky promítnuty do roviny xy a jsou nalezeny pixely, které pokrývají. Zároveň je třeba v těchto pixelech určit vzdálenost plošky od kamery. Do obrazové paměti a paměti hloubky se barva plošky a vzdálenost v daném pixelu zapíše pouze pokud je tato vzdálenost menší než vzdálenost zaznamenaná.

Pokud je výpočet barvy plošky v daném pixelu časově náročnější, například kvůli nanášení textury, je vhodné vykonat nejprve algoritmus bez zápisu do paměti obrazu. Tím je získána vzdálenost nejbližších plošek ve všech pixelech. Nyní je třeba vykonat rasterizaci znovu a pokud bude vzdálenost plošky rovna zaznamenané vzdálenosti v paměti hloubky, provede se i výpočet barvy pixelu. Díky tomu se pro každý pixel počítá barva pouze jednou a zároveň není třeba provádět časově náročné řazení objektů podle vzdálenosti od kamery.

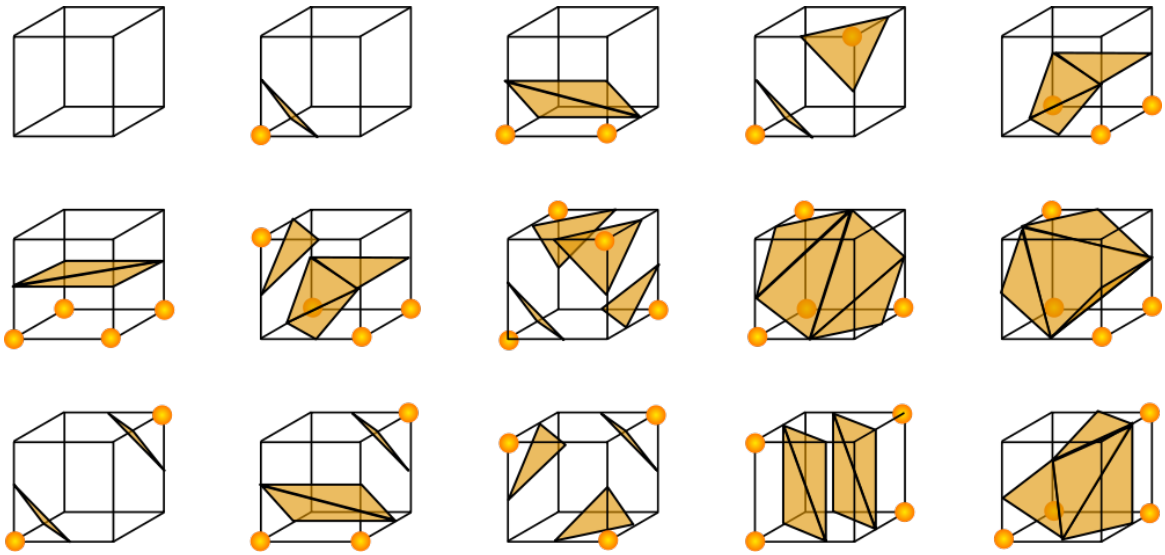
Výhodou tohoto algoritmu je, že s každou ploškou pracuje pouze jednou nebo dvakrát a časová složitost tedy je $O(n)$, kde n značí počet plošek. Další výhodou je, že se jednotlivé plošky zpracovávají postupně a není nutné využívat jiné datové struktury než paměť obrazu a paměť hloubky, jejichž velikost závisí pouze na počtu pixelů obrazu. Díky tomu je tento algoritmus velmi rychlý. Zvláště když je hardwarově podporován grafickou kartou.

3.2.1.1 Marching Cubes

Algoritmus Marching Cubes sloužící k vytvoření polygonální reprezentace zvolené iso-plochy pole hustot nebo distanční funkce byl navržen v roce 1987 a publikován v článku [18]. Celý algoritmus se skládá ze dvou základních kroků. Prvním je lokalizace povrchu odpovídajícího uživatelem nastavené hodnotě hustoty a vytvoření trojúhelníků. Druhým krokem je výpočet normál povrchu ve vrcholech trojúhelníků.

V počáteční fázi prvního kroku je všem vrcholům všech voxelů nacházejících se vně objektu přiřazena hodnota nula. Vrcholům uvnitř objektu nebo na povrchu je přiřazena

hodnota jedna. Povrch se nalézá v těch voxelech, ve kterých je alespoň jeden z jeho vrcholů vně objektu (hodnota nula) a alespoň jeden uvnitř (hodnota jedna). Voxel má osm vrcholů a každý může nabývat dvou stavů. Existuje tedy právě $2^8 = 256$ možných způsobů, jak povrch prochází voxelem. Pomocí dvou různých symetrií lze problém redukovat na 15 vzorů, pro které je třeba určit, jak je triangulovat. Tyto vzory jsou na obrázku 3.5. Algoritmus lze efektivně implementovat použitím tabulky všech možností tvaru povrchu ve voxelu.



Obrázek 3.5: 15 vzorů ze kterých lze pomocí dvou symetrií získat všech 256 možností vzhledu povrchu v buňce. Zdroj [2].

Při výpočtu normály povrchu lze využít skutečnosti, že složky gradientu ve směru tečny k iso-ploše jsou nulové a kolmá složka je nenulová. Normalizací velikosti gradientu lze tedy získat požadovanou normálu. Gradient v požadovaném bodě se určí interpolací gradientů ve vrcholech voxelu. Realistického vzhledu je dosaženo díky tomu, že se tyto gradienty počítají z původních dat.

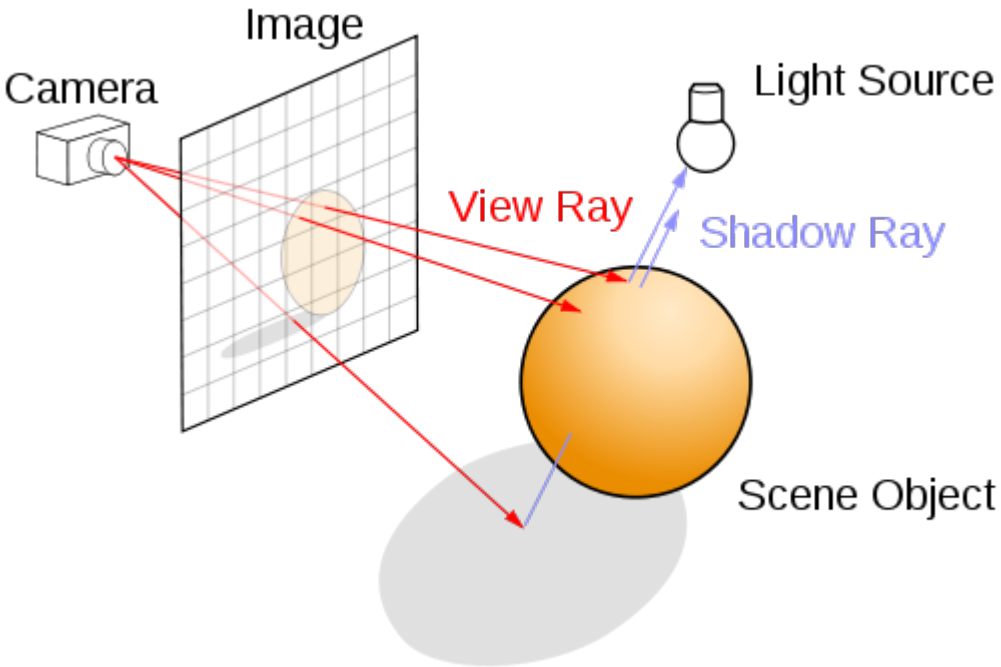
Jsou-li procházeny všechny voxely mřížky voxelů, je operační složitost tohoto algoritmu $O(n^3)$, kde n je počet voxelů vedle sebe v jedné ose. Pokud je pro uložení dat použit oktalový strom, klesá složitost na $O(n^2)$. V obou případech je časová náročnost tohoto algoritmu příliš vysoká a pro větší modely není únosné provádět převod v reálném čase.

3.2.2 Algoritmy vrhání a sledování paprsků

Kromě nutnosti převádět volumetrická data na polygonální je další nevýhodou rasterizačních metod pouze lokální výpočet osvětlení jednotlivých plošek. Není uvažováno jejich zastínění jinými ploškami ani osvětlení odraženým světlem. Naproti tomu existují metody, které vycházejí z principu šíření světla prostorem a vypočítané osvětlení závisí nejen na umístění světel, ale i na rozmístění objektů ve scéně. Těchto metod je více a liší se svou schopností simulovat různé optické jevy a rychlostí výpočtu. Patří mezi ně kromě metod vrhání a sledování paprsků, které budou popsány podrobněji, metody sledování cest (path tracing) nebo

fotonové mapy. Podrobnosti o nich lze nalézt například v knize [29].

Společným rysem všech metod je vyslání světelných paprsků do scény a sledování trajektorií jejich šíření a interakcí s objekty ve scéně. Tento princip navrhl Appel [5] již v roce 1968, který paprsky používal pro určení osvětlených a pozorovatelem viditelných plošek. Nicméně až Whitted [34] v roce 1980 použil paprsky vyslané směrem od pozorovatele k určení viditelných částí objektů a výpočet jejich osvětlení a vytvořil tak metodu rekurzivního zpětného sledování paprsků (ray tracing). Její princip ukazuje obrázek 3.6.



Obrázek 3.6: Princip metody zpětného sledování paprsků. Zdroj [3].

Základem této metody, zkráceně nazývané pouze metoda sledování paprsků, je vyslání alespoň jednoho paprsku s počátkem v ohnisku pomyslné kamery každým pixelem generovaného obrazu a nalezení jeho průsečíku s povrchem nejbližšího objektu. Těmto paprskům se říká primární paprsky. Intenzita světla v nalezeném bodě průsečíku je dána upraveným vztahem Phongova osvětlovacího modelu [28]:

$$I = I_a + k_d \sum_{j=1}^{j=ls} (\overrightarrow{N} \cdot \overrightarrow{L_j}) + k_s S + k_t T,$$

kde I_a je intenzita ambientního světla, k_d koeficient difúzního odrazu, ls počet světel ve scéně, $\overrightarrow{N}$ normála povrchu, $\overrightarrow{L_j}$ jednotkový vektor ke světlu j, k_s koeficient zrcadlového odrazu, S intenzita světla přicházejícího ze směru dokonalého odrazu, k_t koeficient přenosu světla vnitřkem objektu a T intenzita světla ze směru lomeného paprsku.

Difúzní odraz daného světla je započítáván pouze pokud toto světlo vyšetřovaný bod přímo osvětluje, což je vyšetřováno vysláním takzvaného stínového paprsku z bodu průsečíku ke světlu. Pokud stínový paprsek cestou zasáhne nějaký objekt, difúzní odraz se nezapočítá.

Pro určení intenzity světla přicházejícího ze směru dokonalého odrazu a lomu je třeba vyslat další paprsky. Ty jsou nazývány paprsky sekundárními. Postup nalezení průsečíku a vyhodnocení intenzity světla se rekurzivně opakuje a vzniká tím strom vyslaných paprsků. V každém jeho uzlu je třeba určit intenzitu světla závisející na intenzitách v potomcích uzlu. Hloubku stromu lze omezit na rozumnou hodnotu, aniž by degradace konečného obrazu byla znatelná. Jsou-li intenzity světla počítány zvlášť pro jednotlivé barevné složky, je po určení intenzity v kořeni stromu určena barva pixelu, kterým byl primární paprsek vyslán.

Název metoda vrhání paprsků (ray casting) je užívána pro zjednodušenou metodu sledování paprsků, ve které se pracuje pouze s primárními, případně stínovými paprsky. Tato metoda tedy neumožňuje simulovat dokonalý odraz a lom paprsku na povrchu objektu, ale je rychlejší. Také bývá užívána pro vizualizaci vnitřní struktury volumetricky reprezentovaných objektů, jak bylo uvedeno v kapitole 3.1.1, kde by byly tyto efekty nežádoucí.

Ke snížení aliasu je možno vyslat paprsky procházející rohy pixelu a výslednou barvu určit průměrem hodnot. Zároveň se počet paprsků zvýší pouze nepatrně, neboť čtyři sousední pixely sdílí jeden vzorek. Nejsou-li si intenzity v rozích dostatečně podobné, je možno pixel rozdělit na menší části a postup opakovat. Pro dosažení ještě vyšší kvality je lepší volit roztřesené (jittered) vzorkování pixelu.

Jak vyplývá z textu úvodní kapitoly, mohou být metody sledování paprsků využity k vizualizaci polygonálních i volumetrických dat. U polygonální reprezentace objektů však nelze použít tradiční metody zjednodušení scény, jelikož objekty mohou být viditelné díky odrazům. Naproti tomu lze použít obálky objektů a nejprve testovat, zda paprsek protne tyto obálky. Také je vhodné použít některou z metod organizace scény.

U volumetrických dat je situace odlišná. Netestuje se průsečík s ploškami, ale paprsek musí postupně traverzovat datovou strukturou uchovávající vzorky dat a nalézt první bod, ve kterém se nalézá povrch objektu.

Jak bylo uvedeno v kapitole 3.1.4, používají se pro uchování volumetrických dat různé datové struktury. Tato diplomová práce implementuje metodu sledování paprsků pro vzorky distanční funkce uložené v rozích voxelů pravidelné pravoúhlé uniformní mřížky. Je-li vzorků $(P+1)\times(Q+1)\times(R+1)$, může být pro urychlení nalezení průsečíku předpočítáno binární pole indikující přítomnost povrchu ve voxelu, jehož velikost bude $P \times Q \times R$ voxelů. Případně může být použito dvouúrovňové mřížky popsané v kapitole 3.1.4.2. Potom se musí navíc předpočítat hrubé binární pole o velikosti $\frac{P}{k} \times \frac{Q}{k} \times \frac{R}{k}$ buněk indikující přítomnost povrchu alespoň v jedné z $k \times k \times k$ příslušných buněk jemné mřížky.

Pokud by předpočítání binárních mřížek provedeno nebylo, muselo by být v každém voxelu protnutém vyslaným paprskem vyhodnoceno, zda zde paprsek protíná povrch objektu. Takto je úkolem traverzačních algoritmů popsaných v kapitole 3.3 pouze procházení binárních mřížek a nalezení nejbližší buňky jemné mřížky obsahující povrch objektu.

Jakmile je nalezen voxel obsahující povrch objektu, vyhodnotí se některou z metod popsaných v kapitole 3.4, zda v něm paprsek povrch protíná. Pokud ano, určí se souřadnice tohoto průsečíku. Jestliže ne, je pokračováno v traverzování binární mřížky dokud průsečík není nalezen nebo dokud paprsek neopustí mřížku.

Při nalezení bodu průsečíku se v něm pomocí metod uvedených v kapitole 3.5 vypočte normálový vektor. Na jeho základě lze spočítat směr stínových, odražených i lomených paprsků a rekurzivně pokračovat v upřesňování barvy vyhodnocovaného pixelu.

3.3 Efektivní traverzování datových struktur

Tato kapitola představuje techniky efektivního traverzování jedno i dvou-úrovňové pravidelné pravoúhlé mřížky specifikované v kapitole 3.1.4. Každý vyslaný paprsek je dán množinou bodů splňujících parametrickou rovnici přímky $\vec{r}(t) = \vec{o} + t\vec{d}$, kde $\vec{o}$ je počáteční bod paprsku a $\vec{d}$ jeho směrový vektor. Před započetím traverzování mřížky je pro každý paprsek vypočítána hodnota parametru t v bodě vstupu paprsku do mřížky t_0 a hodnota v bodě výstupu paprsku z mřížky t_1. V intervalu daném těmito hodnotami mohou paprsky traverzovat mřížku nezávisle algoritmem 3DDDA (3D Digital Differential Analyzer) nebo mohou využít prostorové koherence a více paprsků může traverzovat společně algoritmem CGT (Coherent Grid Traversal). Nakonec jsou uvedeny techniky pro omezení intervalu traverzování primárních paprsků a tím pádem i urychlení nalezení průsečíku paprsku s povrchem.

3.3.1 Průsečík paprsku s kvádrem obklopujícím mřížku

Obklopující kvádr je zarovnaný se souřadnicovými osami a lze ho proto definovat pomocí minimálního a maximálního bodu v jednotlivých osách $\overrightarrow{gridMin}$ a $\overrightarrow{gridMax}$. Průsečíky paprsku s tímto kvádrem se získají postupným výpočtem průsečíku paprsku s všemi šesti rovinami, ve kterých leží stěny kvádru. Jelikož je každá stěna rovnoběžná s rovinou tvořenou některými dvěma souřadnicovými osami, stačí vypočítat hodnotu parametru t v místě rovnosti hodnoty na třetí ose s příslušnou složkou bodu definujícího kvádr. Například pro průsečík paprsku s rovinou, ve které leží první stěna kolmá na osu x, se hodnota parametru t vypočítá rovnicí:

$$t = \frac{gridMin_x - o_x}{d_x}.$$

Původní metodu představil v roce 1986 Kay a Kajiya [15]. Smits [31] poukázal na to, že standard IEEE pro operace v plovoucí desetinné čárce [12] definuje výsledek dělení kladného čísla nulou jako $+\infty$ a výsledek dělení záporného čísla nulou jako $-\infty$ a není proto třeba speciálně ošetřovat případy, ve kterých je směrový vektor $\vec{d}$ v některé z os nulový.

Williams a kol. [35] odhalil, že kód představený Smitsem funguje korektně téměř ve všech případech kromě takových, kdy je směr paprsku v některé dimenzi roven -0.0 a výsledný interval paprsku v dané dimenzi (t_{near}, t_{far}) není $(-\infty, +\infty)$, ale $(+\infty, -\infty)$. Záporná nula vzniká například vynásobením záporné konstanty kladnou nulou. Problém lze odstranit výpočtem převrácené hodnoty směru paprsku a v případě zápornosti výsledku prohozením hodnot t_{near} a t_{far}. Navíc je tím ušetřena jedna operace dělení za cenu přidání dvou operací násobení. Efektivitu výpočtu snižuje pouze test znaménka převrácené hodnoty směru paprsku. Ten je však nezbytný pro zajištění správnosti algoritmu.

Celý algoritmus poté pracuje tak, že jsou postupně pro jednotlivé osy počítány hodnoty t_{near} a t_{far} udávající interval paprsku v dané ose. Používají se k tomu vzorce $t_{near} = (gridMin_i - o_i) \cdot invd_i$ a $t_{far} = (gridMax_i - o_i) \cdot invd_i$, kde $\overrightarrow{invd} = 1/\vec{d}$ a $i \in \{x, y, z\}$ je index osy. Je-li poté t_{near} větší než t_{far}, hodnoty se vymění.

Interval, ve kterém paprsek prochází kvádrem (t_0, t_1), je na počátku nekonečný $(-\infty, +\infty)$, nebo zleva omezený počátkem paprsku $(0, +\infty)$. Tento interval se postupně omezuje intervaly v jednotlivých osách (t_{near}, t_{far}). Konkrétně, je-li t_{near} větší než t_0, přiřadí se do t_0. Obdobně, je-li t_{far} menší než t_1, nová hodnota t_1 bude hodnota t_{far}. Pokud t_0 překročí

t_1, znamená to, že paprsek obklopující kvádr neprotíná a algoritmus může být i předčasně ukončen.

3.3.2 3D Digital Differential Analyzer

3DDDA je rychlý a jednoduchý inkrementální algoritmus pro traverzování jednoho paprsku prostorem rozděleným mřížkou. V roce 1986 ho popsal Fujimoto a kol. [8]. O rok později ho Amanatides a Woo [4] navrhl efektivněji. Algoritmus je možné upravit i pro další způsoby dělení prostoru.

Princip je vysvětlen pro dvou-dimenzionální případ. Rozšíření do tří dimenzí lze provést přímo. Nejprve je vhodné transformovat paprsky do souřadného systému, ve kterém mřížka o velikosti $P \times Q$ voxelů vyplňuje prostor o velikosti $[0, P) \times [0, Q)$. Za tím účelem postačuje přepočítat souřadnice počátku paprsku do tohoto systému a poté nalézt hodnotu parametru t pro vstup a výstup paprsku z mřížky o nové velikosti. Bod lze do nového systému přepočítat vzorcem $\vec{O} = (\vec{o} - \overrightarrow{gridMin}) \cdot \overrightarrow{invVoxelSize}$, kde $\overrightarrow{invVoxelSize}$ je převrácená hodnota velikosti voxelu. Pokud je velikost mřížky v ose x například 1 a počet voxelů mřížky v této ose například 64, velikost voxelu $voxelSize_x = 1/64$ a převrácená hodnota velikosti voxelu $invVoxelSize_x = 64$. V takovémto souřadném systému lze souřadnice voxelu, ve kterém se nachází bod p, získat prostým zanedbáním desetinného rozvoje bodu p.

V inicializační fázi algoritmu je nalezen interval traverzování mřížky. V bodě vstupu paprsku do mřížky, respektive v bodě počátku paprsku, nalézá-li se uvnitř mřížky, je určen odpovídající voxel. Celočíselné proměnné X a Y jsou nastaveny na souřadnice tohoto voxelu. Proměnné $stepX$ a $stepY$ jsou nastaveny na -1 nebo 1 v závislosti na směru paprsku. Při průchodu mřížkou budou přičítány k proměnným X a Y a ty jimi budou inkrementovány nebo dekrementovány bez nutnosti vždy znovu směr paprsku vyhodnocovat.

Dále je určena hodnota parametru t při prvním průchodu paprsku vertikální hranicí mezi voxely a výsledek je uložen do $tMaxX$. Obdobně je určeno $tMaxY$. Minimum z těchto hodnot specifikuje, jak daleko lze jít podél paprsku a stále zůstat ve stejném voxelu. Také je spočítáno $tDeltaX$ a $tDeltaY$. $tDeltaX$ udává, jak daleko podél paprsku je třeba se posunout, aby paprsek dosáhl vertikální hranice oddělující voxely. Obdobně v $tDeltaY$ je uloženo, o kolik je třeba zvýšit hodnotu t, aby byl posun dostatečný pro dosažení horizontální hranice mezi voxely.

Nakonec jsou nastaveny souřadnice OX a OY na hodnotu odpovídající počtu voxelů mřížky nebo hodnotu -1 v závislosti na jednotlivých složkách směrového vektoru paprsku. Tyto hodnoty slouží k efektivnímu vyhodnocení, že paprsek opustil mřížku.

V traverzační fázi algoritmu se porovnáním hodnot $tMaxX$ s $tMaxY$ rozhoduje, zda paprsek opustí současný voxel horizontální nebo vertikální hranicí. Pokud vertikální ($tMaxX < tMaxY$), k $tMaxX$ je přičteno $tDeltaX$ a k X je přičteno $stepX$. Pokud hodnota X dosáhne hodnoty OX, paprsek opustil mřížku a traverzování končí. Podobně v případě průchodu paprsku horizontální hranicí voxelu, $tDeltaY$ je přičteno k $tMaxY$ a $stepY$ k Y. Pokud je Y rovno OY, traverzování končí.

Při rozšíření cyklu do tří dimenzí jsou pro každý traverzační krok vyžadována dvě porovnání v plovoucí desetinné čárce, jedno sečtení v plovoucí desetinné čárce, jedno celočíselné porovnání a jedno celočíselné sečtení. Inicializační fáze vyžaduje kolem 40 operací.

Algoritmus může být velmi snadno upraven pro traverzování dvouúrovňové mřížky. Traverzování hrubé mřížky je prováděno popsaným algoritmem, pouze je třeba na místo velikosti voxelu uvažovat velikost bloku voxelů. Jakmile buňka hrubé mřížky indikuje přítomnost povrchu objektu, přejde se k traverzování jemné mřížky.

Nejprve je opět nutné předpočítat proměnné a konstanty jako u traverzování hrubé mřížky. Lze využít toho, že směr paprsku a velikost bloku voxelů zůstává stejná a tak je vhodné předpočítat pomocné konstanty už před zahájením traverzování hrubé mřížky. Při vhodném zvolení souřadných systémů hrubá a jemné mřížky je možno v obou traverzovacích algoritmech využít stejných konstant, čímž jsou ušetřeny registry procesoru.

Pokud jsou souřadnice vstupu paprsku do voxelu jemné mřížky počítány na základě parametru t získaného postupným přičítáním hodnot $tDelta$ při traverzování hrubé mřížky, nemusí být vlivem numerických nepřesností dostatečně přesné. Kvůli tomu se může stát, že paprsek traverzuje navíc jedním voxelem jemné mřížky, který se nenachází v oblasti příslušející buňce hrubé mřížky. To v běžné situaci nepředstavuje problém. Jestliže je však jemná mřížka definována pouze v oblastech, kde hrubá mřížka indikuje přítomnost povrchu, dojde k chybě. V tom případě je vhodnější počítat souřadnice vstupu paprsku do voxelu algoritmem pro nalezení průsečíku paprsku s kvádrem obklopujícím buňku hrubé mřížky.

Traverzování jemné mřížky vyžaduje stejný počet operací jako bylo uvedeno výše. Pokud paprsek dosáhne voxelu jemné mřížky, který již nenáleží oblasti buňky hrubé mřížky, traverzuje se dále opět mřížka hrubá. Výhodou představeného algoritmu je, že každý krok traverzování mřížky vyžaduje velmi nízký počet operací a je k němu třeba pouze malého počtu proměnných.

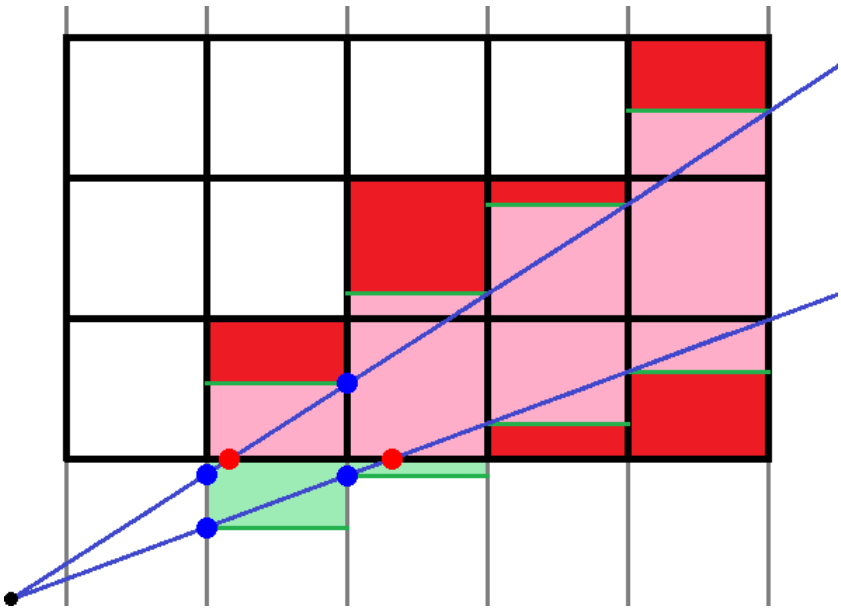
3.3.3 Coherent Grid Traversal

Jiný algoritmus traverzování paprsků mřížkou představil v roce 2006 Wald a kol. [33]. Ten využívá podobného směru primárních paprsků a umožňuje vyslat celou skupinu paprsků najednou. Tyto paprsky tvoří takzvaný paket paprsků a traverzují mřížku společně. Výsledný čas generování obrazu je kratší díky tomu, že je snížen počet traverzování mřížky, i když čas jednoho traverzování je výrazně vyšší než čas traverzování samostatného paprsku algoritmem 3DDDA. Představený algoritmus je možno implementovat s použitím SSE instrukcí a tím lépe využít výkon současných procesorů.

Gribble a kol. [10] algoritmus upravil pro vizualizaci částic a navrhl nahradit každý stínový paprsek paketem paprsků, který umožňuje vzorkovat zastínění jednoho plošného zdroje světla. Díky tomu získal měkké stíny. Knoll a kol. [17] použil algoritmu koherentního traverzování mřížky k traverzování makrobuněk oktalového stromu. Obsahuje-li makrobuňka povrch objektu, mohou být pro dosažení kvalitnějšího výsledku traverzovány i menší buňky stromu. Pro zefektivnění navrhl metody omezující traverzování menších buněk, které nejsou protnuty paprsky paketu, neobsahují povrch nebo jsou protnuty pouze paprsky, které již povrch zasáhly. Tyto metody lze využít i u traverzování dvouúrovňové mřížky.

Díky své koherenci procházejí paprsky jednoho paketu většinou stejnými voxely mřížky. Proto je vhodné testovat na přítomnost povrchu postupně všechny voxely, které paket protíná. Nalézá-li se ve voxelu povrch objektu, zjistí se, které paprsky tento voxel skutečně protínají algoritmem nalezení průsečíku paprsku s kvádrem. Poté se přistoupí k hledání prů-

sečíku vyhovujících paprsků s povrchem. Postupné procházení voxelů protnutých paketem paprsků znázorňuje obrázek 3.7.



Obrázek 3.7: Traverzování 2D mřížky o velikosti 5×3 voxelů metodou CGT. Modře jsou vyznačeny krajní paprsky svazku paprsků. Vnitřní paprsky vyznačeny nejsou, může jich být libovolné množství. Podle libovolného z paprsků je zvolena hlavní osa procházení $\vec{K}$ – v tomto případě osa x. Kolmá osa na hlavní osu šíření $\vec{U}$ je poté osa y. Prostor je šedými vertikálními čarami rozdělen na plátky podél hlavní osy. Je nalezen první průsečík všech paprsků s mřížkou (pro krajní paprsky vyznačeno červeně) a podle minimální hodnoty je určen plátek, ve kterém paket vstupuje do mřížky. Pro tento plátek jsou určeny body vstupu a výstupu všech paprsků a určeny extrémní hodnoty – modré body. Z nich je určen osově zarovnaný obklopující obdélník vyznačující oblast, kterou paket prochází – vyznačeno růžově a zeleně. K této oblasti je v jednotlivých traverzačních krocích přičítána konstantní diference taktéž určená z extrémních bodů. Které voxely je nutno testovat je určeno oříznutím obdélníku o oblasti mimo mřížku (odstraní se zelené obdélníky, zbydou růžové) a rozšířením obdélníku do celé plochy voxelů pokrytých pouze částečně (k růžovým se přidají červedé obdélníky).

Před započetím traverzování je nejprve nutné transformovat paprsky do souřadného systému, ve kterém mřížka o velikosti $P \times Q \times R$ voxelů vyplňuje prostor o velikosti $[0, P) \times [0, Q) \times [0, R)$ stejným postupem jako u algoritmu 3DDDA. Dále je třeba vybrat hlavní osu šíření paketu paprsků $\vec{K}$. Ta se zvolí podle dominantní složky směrového vektoru $\vec{d}$ libovolného paprsku. Všechny paprsky budou procházet podle stejné osy šíření nezávisle na jejich skutečné dominantní složce směrového vektoru. Pokud by byl odklon některého z paprsků od paprsku, podle kterého byla osa šíření určena, větší než 90°, nebyl by pro tento paprsek nalezen průsečík s povrchem. Takovéto problémy však u primárních paprsků typicky nehrozí. Osy kolmé na hlavní osu šíření jsou označeny $\vec{U}$ a $\vec{V}$.

Prostor je pomyslně rozdělen na plátky o šířce jednoho voxelu podél hlavní osy šíření. Jsou nalezeny body vstupu všech paprsků do mřížky. Z nich je určen plátek, ve kterém paket

do mřížky vstupuje. Jestliže je složka $\vec{K}$ směrového vektoru paprsků kladná (v ose $\vec{K}$ směřuje paket zleva doprava), určí se první plátek podle minimální hodnoty složky $\vec{K}$ bodů vstupu paprsků do mřížky. V opačném případě se určí z maximální hodnoty.

Je-li určen plátek, ve kterém paket vstupuje do mřížky, vypočítají se pro všechny paprsky body vstupu a výstupu z tohoto plátku. Extrémní hodnoty v osách $\vec{U}$ a $\vec{V}$ tvoří osově zarovnaný obklopující kvádr (AABB) oblasti, kterou paket v tomto plátku prochází. Tento kvádr se označuje $B^{(0)} = (u_{min}, u_{max}, v_{min}, v_{max})$. Z rozdílu extrémních bodů v místě vstupu paprsků do plátku a v místě výstupu paprsků z plátku se určí diference dvou po sobě následujících plátků $\triangle B = (du_{min}, du_{max}, dv_{min}, dv_{max})$. Osově zarovnaný obklopující kvádr paketu paprsků v plátku $i+1$ lze poté vždy určit přičtením diference k obklopujícímu kvádru plátku předchozího: $B^{(i+1)} = B^{(i)} + \triangle B$.

Při traverzování jsou voxely protnuté paketem v každém plátku určeny oříznutím obklopujícího kvádru na velikost mřížky a zanedbáním desetinného rozvoje bodů určujících vzniklý kvádr. V každém paketem protnutém voxelu obsahujícím povrch objektu jsou nalezeny paprsky, které tímto voxelem procházejí a je hledán jejich průsečík s povrchem. Jestliže je průsečík nalezen, nesmí být paprsek vyřazen z dalšího hledání, dokud se nepřejde k následujícímu plátku. Pro tento paprsek by totiž mohl existovat průsečík s povrchem i v jiném voxelu plátku a tento průsečík by mohl být blíž počátku paprsku než průsečík dosud nalezený. Traverzování mřížky může být ukončeno při přechodu do nového plátku, pokud již všechny paprsky paketu nalezly průsečík s povrchem nebo pokud paket opustil mřížku.

Algoritmus lze upravit pro traverzování dvouúrovňové mřížky. Stačí vhodně spočítat obklopující kvádr paketu pro plátek buněk hrubé mřížky a traverzovat tyto plátky. Pokud se v oblasti plátku pokryté paketem nalézá buňka obsahující povrch objektu, rozdělí se tento plátek na plátky jemné mřížky a ty jsou traverzovány zvlášť s použitím obklopujícího boxu paketu přizpůsobeného pro buňky jemné mřížky. Je možné testovat všechny buňky jemné mřížky patřící do obklopujícího kvádru paketu bez ohledu na stav buněk hrubé mřížky, nebo obklopující kvádr oříznout o oblasti, ve kterých buňky hrubé mřížky přítomnost povrchu neindikují. Problém nastane, pokud se buňka hrubé mřížky neindikující přítomnost povrchu ocitne uvnitř oblasti pokryté paketem, pro tuto buňku nejsou definovány buňky jemné mřížky a okolní buňky hrubé mřížky indikují přítomnost povrchu. Proto je v případě paměťově úsporné definice dvouúrovňové mřížky potřeba implementovat sofistikovanější metodu uříznutí traverzovaných buněk jemné mřížky.

Algoritmus CGT je efektivní zejména pro paprsky s velmi podobným směrovým vektorem. Jestliže se směrové vektory liší více, zahrnuje obklopující kvádr paketu po několika traverzačních krocích velké množství voxelů a výkon klesá. Obdobná situace nastává, je-li objekt pozorován z větší dálky nebo je-li zaznamenán ve vysokém rozlišení.

3.3.4 Omezení intervalu traverzování primárních paprsků

Neubauer a kol. [22] představil metodu, která umožňuje urychlit nalezení prvního průsečíku paprsku s povrchem objektu. Předpokladem je sdružení voxelů mřížky do větších buněk, které jsou zde nazývány makro-buňkami. Tento předpoklad splňuje organizace voxelů dvouúrovňovou mřížkou.

Makro-buňky obsahující povrch objektu jsou postupně projektovány do roviny obrazu a jsou nalezeny pixely, které tato projekce pokrývá. Každým takovýmto pixelem, kterému ne-

byla zatím přiřazena barva, je vržen paprsek. Hledání průsečíku paprsku s povrchem začíná v bodě, ve kterém paprsek vstupuje do makro-buňky a končí nalezením průsečíku nebo opuštěním makro-buňky. Paprsek tedy traverzuje pouze voxely jemné mřížky patřící do makro-buňky a je hledán první voxel indikující přítomnost povrchu.

Jelikož první průsečík paprsku s povrchem určí výslednou barvu pixelu, je důležité, aby byly makro-buňky projektovány do roviny obrazu ve správném pořadí. Protože je použita perspektivní projekce, každý paprsek má unikátní směr. Nicméně všechny primární paprsky (paprsky, které mají stejný bod počátku) protínající nějaké dvě makro-buňky je protnou ve stejném pořadí.

Pořadí se snadno vyhodnocuje u oktalového stromu, kde může být použit reprezentativní paprsek protínající všechny podprostory zpracovávaného uzlu k určení správného pořadí procházení potomků uzlu. Paprsek protínající všechny podprostory uzlu reprezentované jeho potomky je ten, který prochází středem prostoru reprezentovaného uzlem. Znaménka komponent jeho směrového vektoru určují pořadí procházení. Při použití dvouúrovňové mřížky je nalezení správného pořadí obtížnější. Makro-buňky lze například projektovat v pořadí jejich rostoucí vzdálenosti od počátku paprsků.

Vržení jednoho paprsku traverzujícího k místu nalezení průsečíku, nebo dokonce celou mřížkou, nemusí být v obecném případě pomalejší, než vrhání mnoha lokálních paprsků. Pro lokální paprsky je totiž nutno opakovaně provádět inicializační fázi vržení paprsku. Čas je ušetřen tehdy, je-li pro každý pixel vrženo pouze malé množství lokálních paprsků nebo musí-li většina samostatných paprsků traverzovat velkou část prostoru. Pro omezení počtu lokálních paprsků mohou být použity techniky ořezávání makro-buněk a brzkého ukončení linie obrazu, které jsou však složité a jsou nad rámec této práce. Jejich vysvětlení lze nalézt v článku [22].

Metoda omezení intervalu traverzování primárních paprsků vylepšená o metody omezení počtu lokálních paprsků byla v článku [22] porovnána s optimalizovaným algoritmem nalezení prvního průsečíku vrženého paprsku. Ukázalo se, že velmi záleží na tvaru povrchu objektu. Nový algoritmus je rychlejší, pouze pokud před nalezením povrchu původním algoritmem musel paprsek traverzovat větší část prostoru.

Nevýhodou této metody je složitá paralelizace, neboť není předem jasné, do které části obrazu bude která makro-buňka projektována. Pro paralelní implemetaci metody sledování paprsků, zejména na grafické kartě, může být metoda upravena a do jisté míry i zjednodušena, čímž vznikne dále popsaný algoritmus.

V inicializační fázi jsou vytvořeny polygonální krychle pro všechny buňky hrubé, případně dokonce jemné mřížky obsahující povrch objektu. Při generování obrazu jsou všechny tyto krychle projektovány do roviny obrazu a jsou rasterizovány s použitím paměti hloubky. Projekční matice musí odpovídat způsobu generování primárních paprsků. Po rasterizaci obsahuje paměť hloubky pro všechny pixely informaci o minimální vzdálenosti od kamery, ve které se může povrch objektu nacházet. Nakonec jsou vyslány primární paprsky, jejichž počátek intervalu traverzování je nastaven na zjištěnou minimální vzdálenost povrchu od kamery a je nalezen jejich průsečík s povrchem.

Interval traverzování je možné omezit i shora, pokud se při rasterizaci krychlí vyhodnotí v každém pixelu i maximální vzdálenost nejvzdálenějších částí krychlí od kamery. Vyžaduje-li zjištění této informace více času, nebo je-li dokonce třeba provádět opakovanou projekci a

rasterizaci krychlí, je třeba zvážit její přínos. Pouze paprsky, které vůbec nezasáhly povrch, ale cestou procházely buňkami obsahujícími povrch, budou díky této informaci traverzovat menší část prostoru a ušetří čas. Takovýchto paprsků ale bývá velmi málo.

3.4 Nalezení průsečíku paprsku s povrchem

Byl-li nalezen voxel obsahující povrch objektu, je třeba co nejpřesněji a nejrychleji určit bod, ve kterém paprsek v tomto voxelu povrch protíná. Metody sloužící k tomuto účelu lze rozdělit do dvou skupin. První tvoří skupiny aproximativní, které jsou rychlé, ale výsledek je pouze přibližný. Patří sem metoda středního bodu a metody založené na lineární interpolaci. Druhou skupinu tvoří metoda analytická, která je z lokálního pohledu přesná, ale velmi pomalá. Rychlejší a téměř stejně přesnou metodou je metoda izolace a iterativního nalezení kořenů, která část výpočtu provádí analyticky a část aproximativně.

3.4.1 Metoda středního bodu

Tato metoda je nejrychlejším možným řešením daného problému. Její přesnost je však nedostatečná a metoda tak slouží spíše jako referenční pro zhodnocení rychlosti ostatních metod. Principem je prohlášení bodu ve středu spojnice mezi bodem vstupu a bodem výstupu paprsku z voxelu za bod průsečíku. Není ani ověřováno, že body vstupu a výstupu neleží na stejné straně povrchu. Metoda tedy vůbec nečte distanční pole a neinterpoluje hodnoty, což je důvodem nižšího výkonu ostatních metod.

3.4.2 Metody lineární interpolace

Základem metody lineární interpolace je určení hodnot distančního pole v bodě vstupu i výstupu paprsku z voxelu. Pro jejich určení je třeba provést bilineární interpolace hodnot v rozích čtvercových stěn voxelu, ve kterých bod vstupu a výstupu leží. Porovnáním znamének interpolací získaných hodnot distanční funkce je vyhodnoceno, zda oba body leží na opačných stranách povrchu a tím pádem paprsek v tomto voxelu povrch protíná. Pokud protíná, určí se bod průsečíku lineární interpolací souřadnic vstupu a výstupu s vahou danou hodnotami distanční funkce v těchto bodech.

Neubauer a kol. [22] zpřesnil metodu opakováním postupu lineární interpolace. V bodě určeném lineární interpolací jako bod průsečíku s povrchem se určí hodnota distanční funkce trilineární interpolací hodnot v rozích voxelu. Porovnáním znaménka získané hodnoty se znaménkem hodnoty v bodě vstupu se rozhodne, zda bod průsečíku leží v intervalu mezi bodem vstupu a získaným bodem nebo v intervalu mezi získaným bodem a bodem výstupu. Pro interval, kde průsečík leží, se opět lineární interpolací určí předpokládaný bod průsečíku a postup se opakuje.

Opakování je možno provádět, dokud není průsečík nalezen s požadovanou přesností, nebo je možno provést přednastavený počet opakování, aby nemusela být přesnost složitě vyhodnocována. Tato metoda je rychlá a zároveň poskytuje dobrý odhad bodu průsečíku. V situacích, kdy je tvar povrchu ve voxelu složitější, však nemusí pracovat korektně. Například pokud se bod vstupu i výstupu nacházejí na stejné straně povrchu, ale přesto paprsek povrch protíná.

3.4.3 Analytická metoda

Analytickou metodu pro nalezení průsečíku paprsku s povrchem objektu ve voxelu popsal Parker a kol. [27]. Hodnota distanční funkce v každém bodě $\vec{p} = (x, y, z)$ uvnitř voxelu o jednotkové velikosti může být spočítána trilineární interpolací hodnot ve vrcholech voxelu

$$\begin{aligned}\rho(x,y,z) = \; & (1-u)(1-v)(1-w)\rho_{000} \\ & +(1-u)(1-v)(\quad w)\rho_{001} \\ & +(1-u)(\quad v)(1-w)\rho_{010} \\ & +(1-u)(\quad v)(\quad w)\rho_{011} \\ & +(\quad u)(1-v)(1-w)\rho_{100} \\ & +(\quad u)(1-v)(\quad w)\rho_{101} \\ & +(\quad u)(\quad v)(1-w)\rho_{110} \\ & +(\quad u)(\quad v)(\quad w)\rho_{111},\end{aligned}$$

kde $u = x - \lfloor x \rfloor$, $v = y - \lfloor y \rfloor$, $w = z - \lfloor z \rfloor$ a ρ_{000} až ρ_{111} jsou hodnoty distanční funkce v rozích voxelu. Definicí $u_0 = 1 - u$ a $u_1 = u$ a podobnou definicí pro v_0, v_1, w_0 a w_1 lze funkci zapsat

$$\rho(x,y,z) = \sum_{i,j,k \in \{0,1\}} u_i v_j w_k \rho_{ijk}.$$

Hodnoty distanční funkce v bodech voxelu, kterými prochází paprsek $\vec{r}(t) = \vec{o} + t\vec{d}$ lze získat dosazením rovnice paprsku do interpolační funkce

$$\rho(t) = \sum_{i,j,k \in \{0,1\}} (u_i^o + t u_i^d)(v_j^o + t v_j^d)(w_k^o + t w_k^d),$$

kde $u_0^o = \lceil x \rceil - p_{in,x}$, p_{in} je bod vstupu paprsku do voxelu, $u_1^o = 1 - u_0^o$, $u_0^d = -d_x$ a $u_1^d = d_x$. v_j^o, v_j^d, w_k^o a w_k^d jsou definovány obdobně.

Po úpravách je získána kubická polynomiální rovnice tvaru

$$\rho(t) = A\, t^3 + Bt^2 + Ct + D, \tag{3.1}$$

jejíž koeficienty jsou

$$A = \sum_{i,j,k \in \{0,1\}} u_i^d v_j^d w_k^d \rho_{ijk},$$

$$B = \sum_{i,j,k \in \{0,1\}} (u_i^o v_j^d w_k^d + u_i^d v_j^o w_k^d + u_i^d v_j^d w_k^o)\rho_{ijk},$$

$$C = \sum_{i,j,k \in \{0,1\}} (u_i^d v_j^o w_k^o + u_i^o v_j^d w_k^o + u_i^o v_j^o w_k^d)\rho_{ijk},$$

$$D = \sum_{i,j,k \in \{0,1\}} u_i^o v_j^o w_k^o \rho_{ijk}.$$

Paprsek protíná povrch objektu ve všech bodech, v nichž je hodnota polynomu rovna nule. Pro získání řešení je třeba nalézt všechny kořeny rovnice a vybrat ten nejmenší ležící

v intervalu, ve kterém paprsek protíná voxel (t_{in}, t_{out}). Snadno implementovatelné řešení kubických rovnic popsal Schwarze [30].

Kubická rovnice tvaru

$$c_3x^3 + c_2x^2 + c_1x + c_0 = 0$$

je vydělena konstantou c_3, čímž je získána rovnice v normální formě

$$x^3 + Ax^2 + Bx + C = 0.$$

Substituce $x = y - \frac{A}{3}$ eliminuje kvadratický člen, vznikne

$$y^3 + 3py + 2q = 0.$$

Použitím Cardanova vzorce je dán determinant $D = q^2 + p^3$ a $u, v = \sqrt[3]{-q \pm \sqrt{D}}$. Kořeny rovnice jsou

$$y_1 = u + v$$

$$y_{2,3} = -\frac{u+v}{2} \pm \frac{\sqrt{3}}{2}(u - v)i.$$

Je-li $D > 0$, y_1 je reálný kořen a y_2 spolu s y_3 jsou dva komplexně sdružené kořeny. Pokud $D = 0$, řešením rovnice jsou dvě reálná čísla ($y_2 = y_3$). Pro $D < 0$ jsou všechny tři kořeny různá reálná čísla. V takovém případě je vhodné použít trigonometrickou substituci pro nalezení všech řešení bez nutnosti použití aritmetiky komplexních čísel

$$\cos\varphi = -\frac{q}{\sqrt{-p^3}}$$

$$y_1 = 2\sqrt{-p}\cos\frac{\varphi}{3}$$

$$y_{2,3} = -2\sqrt{-p}\cos\frac{\varphi \pm \pi}{3}.$$

Výsledky jsou poté získány resubstitucí vypočítaných hodnot za x.

Analytická metoda je přesná, ale kvůli velkému množství výpočtů pomalá. Pro určité koeficienty polynomiální rovnice může být navíc výpočet numericky nestabilní.

3.4.4 Metoda izolace a iterativního nalezení kořenů

Cílem metody izolace a iterativního nalezení kořenů, kterou představil Marmitt a kol. [19], je dosáhnout přesnosti analytické metody a rychlosti metody opakované lineární interpolace. Jejími základními kameny jsou hledání pouze prvního průsečíku a fakt, že opakovaná lineární interpolace nalezne správný kořen, pokud je v počátečním intervalu hledání obsažen právě jeden. Proto jsou nejprve izolovány kořeny výpočtem extrémních hodnot polynomiální rovnice. To lze provést derivací polynomiální rovnice 3.1 a vyřešením vzniklé kvadratické rovnice.

Jsou nalezeny maximálně dva extrémy dělící spolu s bodem vstupu paprsku do voxelu a bodem výstupu paprsek na maximálně tři segmenty. Ty jsou následně procházeny zepředu dozadu a v bodech začátku a konce segmentu je trilineární interpolací hodnot v rozích voxelu

počítána hodnota distanční funkce. Jestliže mají vypočítané hodnoty odlišné znaménko, protíná v daném segmentu paprsek právě jednou povrch objektu. Použitím opakované lineární interpolace lze poté libovolně přesně získat hledaný průsečík.

Představená metoda je sice velmi přesná a dokonce ani tolik netrpí numerickou nestabilitou jako analytická metoda, ale rychlosti opakované lineární interpolace nedosahuje. Proto se hodí pouze v situacích, kdy je přesnost nalezení povrchu upřednostňována před rychlostí.

3.5 Výpočet normálových vektorů

Pro správný výpočet osvětlení povrchu objektu v bodě průsečíku s paprskem je důležité co nejpřesněji určit normálový vektor. Tedy vektor o jednotkové velikosti, který je na povrch v daném bodě kolmý a směřuje ven z objektu. Vlastností distanční funkce je, že změny funkční hodnoty jsou při posunu rovnoběžně s povrchem nulové a při posunu kolmo k povrchu maximální. Směr normálového vektoru v bodě průsečíku paprsku s povrchem je proto shodný se směrem gradientu distanční funkce v tomto bodě. Normálový vektor je tedy možné vypočítat normalizací gradientu. Pro výpočet gradientu ze vzorků distanční mapy slouží metody uvedené v této kapitole.

3.5.1 Analytická derivace interpolační funkce

Gradient distančního pole v daném bodě je třísložkový vektor udávající směr a velikost největší změny tohoto pole. Jednotlivé složky gradientu lze určit výpočtem parciálních derivací distanční funkce. Ta je uvnitř voxelu definována předpisem interpolační funkce. Například pro trilineární interpolaci je distanční funkce uvnitř voxelu o jednotkové velikosti definována

$$\begin{aligned}\rho(u,v,w) = \; & (1-u)(1-v)(1-w)\rho_{000} \\ & +(1-u)(1-v)(\quad w)\rho_{001} \\ & +(1-u)(\quad v)(1-w)\rho_{010} \\ & +(1-u)(\quad v)(\quad w)\rho_{011} \\ & +(\quad u)(1-v)(1-w)\rho_{100} \\ & +(\quad u)(1-v)(\quad w)\rho_{101} \\ & +(\quad u)(\quad v)(1-w)\rho_{110} \\ & +(\quad u)(\quad v)(\quad w)\rho_{111},\end{aligned}$$

kde $u, v, w \in \langle 0, 1\rangle$ a ρ_{000} až ρ_{111} jsou hodnoty distanční funkce v rozích voxelu. Parciální derivace této funkce podle u je

$$\begin{aligned}\frac{\partial\rho(u,v,w)}{\partial u} = & -(1-v)(1-w)\rho_{000} \\ & -(1-v)(w)\rho_{001} \\ & -(v)(1-w)\rho_{010} \\ & -(v)(w)\rho_{011} \\ & +(1-v)(1-w)\rho_{100} \\ & +(1-v)(w)\rho_{101} \\ & +(v)(1-w)\rho_{110} \\ & +(v)(w)\rho_{111}.\end{aligned}$$

Úpravou výrazu a definováním $v_0 = 1 - v$ a $v_1 = v$ a obdobným definováním w_0 a w_1 může být konečná podoba vzorce pro výpočet první složky gradientu

$$\frac{\partial \rho(u,v,w)}{\partial u} = v_0[w_0(\rho_{100} - \rho_{000}) + w_1(\rho_{101} - \rho_{001})] + v_1[w_0(\rho_{110} - \rho_{010}) + w_1(\rho_{111} - \rho_{011})].$$

Vzorce pro výpočet ostatních složek gradientu lze odvodit obdobným postupem.

Tato metoda je z lokálního hlediska přesná, ale je-li použita pouze trilineární interpolace, může se směr gradientu v sousedních voxelech výrazněji lišit a rekonstruovaný povrch obsahuje ostré přechody. Použitím lepší interpolační funkce je možné tyto přechody odstranit, nicméně existují metody, které určí gradient téměř stejně kvalitně, ale rychleji.

3.5.2 Výpočet gradientu centrální, dopřednou nebo zpětnou diferencí

Gradient udává směr a velikost změny distančního pole. Při výpočtu normálového vektoru není třeba znát velikost gradientu, postačuje směr. Odhad směru lze získat výpočtem diferencí hodnot v okolí bodu, pro který je gradient počítán. Pokud jsou hodnoty, ze kterých je diference počítána, symetricky posunuté od daného bodu ve směru souřadnicových os, hovoří se o centrální diferenci. Posun hodnot d od vyhodnocovaného bodu lze volit libovolně, avšak nejvhodnější volbou je nastavit posun na velikost voxelu. Hodnota distanční funkce v posunutých bodech musí být vypočítána interpolačním filtrem. Předpis pro výpočet gradientu centrální diferencí má poté tvar

$$\nabla\rho\,(x,y,z) = \begin{pmatrix} \rho\,(x+d,y,z) - \rho\,(x-d,y,z) \\ \rho\,(x,y+d,z) - \rho\,(x,y-d,z) \\ \rho\,(x,y,z+d) - \rho\,(x,y,z-d) \end{pmatrix}^T .$$

Pro výpočet centrální diference je třeba provést šest interpolací. Rychlejším, i když méně přesným řešením, je použití dopředné nebo zpětné diference dané výrazem

$$\nabla\rho\,(x,y,z) = \begin{pmatrix} \rho\,(x+d,y,z) - \rho\,(x,y,z) \\ \rho\,(x,y+d,z) - \rho\,(x,y,z) \\ \rho\,(x,y,z+d) - \rho\,(x,y,z) \end{pmatrix}^T .$$

3.5.3 Interpolace gradientů v rozích voxelu

Základem metody je výpočet gradientu v rozích voxelů a jejich následná interpolace do požadovaného bodu uvnitř voxelu. Gradienty v rozích voxelu lze spočítat centrální, dopřednou nebo zpětnou diferencí. Při nastavení posunu hodnot d na velikost voxelu není třeba pro výpočet gradientu v rozích interpolovat žádné hodnoty. Při výpočtu se tedy provádí pouze jedna interpolace, díky čemuž má tato metoda potenciál být rychlejší než metoda počítající gradient diferencí přímo v daném bodě.

Jak lze odvodit, výsledek obou metod je v takovém případě stejný. Nechť je velikost voxelu rovna jedné a ρ_{ijk}, $i,j,k \in \{-1,0,1,2\}$ jsou hodnoty v rozích vyhodnocovaného a okolních voxelů. První složka gradientu počítaná centrální diferencí je

$$\frac{\partial \rho(x,y,z)}{\partial x} = \rho\,(x+1,y,z) - \rho\,(x-1,y,z)\,.$$

Hodnoty distanční funkce lze spočítat trilineární interpolací hodnot v rozích příslušných voxelů

$$\begin{aligned}\frac{\partial\rho(x,y,z)}{\partial x} = &+u_0v_0w_0\rho_{100} &&+u_0v_0w_1\rho_{101} &&+u_0v_1w_0\rho_{110} &&+u_0v_1w_1\rho_{111}\\ &+u_1v_0w_0\rho_{200} &&+u_1v_0w_1\rho_{201} &&+u_1v_1w_0\rho_{210} &&+u_1v_1w_1\rho_{211}\\ &-u_0v_0w_0\rho_{-100} &&-u_0v_0w_1\rho_{-101} &&-u_0v_1w_0\rho_{-110} &&-u_0v_1w_1\rho_{-111}\\ &-u_1v_0w_0\rho_{000} &&-u_1v_0w_1\rho_{001} &&-u_1v_1w_0\rho_{010} &&-u_1v_1w_1\rho_{011},\end{aligned}$$

kde $u_0 = 1-u$, $u_1 = u$ a $u = x - \lfloor x \rfloor = x - \lfloor x \rfloor + k \ -k \ = x + k - \lfloor x + k \rfloor$, $k \in \mathbf{Z}$. Obdobně je definováno v_0, v_1, w_0 a w_1. Vytknutím lze získat tvar, ve kterém jsou nejprve centrální diferencí vypočítány gradienty v rozích a ty jsou následně interpolovány do požadovaného bodu

$$\begin{aligned}\frac{\partial\rho(x,y,z)}{\partial x} = \quad & u_0\{v0[w0(\rho_{100}-\rho_{-100})+w1(\rho_{101}-\rho_{-101})]\\ & \quad +v1[w0(\rho_{110}-\rho_{-110})+w1(\rho_{111}-\rho_{-111})]\}\\ & +u_1\{v0[w0(\rho_{200}-\rho_{000}) \ \ +w1(\rho_{201}-\rho_{001})]\\ & \quad +v1[w0(\rho_{210}-\rho_{010}) \ \ +w1(\rho_{211}-\rho_{011})]\}.\end{aligned}$$

Pro ostatní složky gradientu lze odvození provést podobným způsobem.

Při použití trilineárního filtru je u metody výpočtu gradientu centrální diferencí i u metody interpolace gradientů v rozích voxelu vypočítaných centrální diferencí potřeba načíst 32 vzorků distanční funkce. První metoda vyžaduje výpočet šesti interpolací a tří diferencí, druhá metoda výpočet jedné interpolace a dvaceti čtyř diferencí. Jelikož je výpočet interpolace mnohem pomalejší než výpočet diference, je v klasické implementaci druhá metoda rychlejší. Uloží-li se vzorky distanční funkce v implementaci na grafické kartě jako 3D textura, poskytne interpolované hodnoty požadované první metodou texturovací hardware a rychlejší je v tomto případě metoda první.

Kapitola 4

Implementace

Za účelem otestování různých možností generování obrazu objektů reprezentovaných distančním polem s použitím různých technologií bylo implementováno devět samostatně fungujících metod. U každé metody je možno volit mezi traverzováním uniformní nebo dvouúrovňové mřížky. Dále lze u některých metod vybrat metodu výpočtu průsečíku vyslaných paprsků s povrchem a metodu výpočtu normálových vektorů.

Prvních osm implementovaných metod využívá pro zobrazení vygenerovaného obrazu a pro interakci s uživatelem grafickou knihovnu CImg [1]. Jsou implementovány v rámci testovací aplikace DFRT (Distance Field Ray Tracing) a lze mezi nimi přepínat za běhu programu. Devátá metoda implementuje metodu omezení intervalu traverzování primárních paprsků rasterizací buněk mřížky obsahujících povrch popsanou v kapitole 3.3.4. K tomu využívá grafickou knihovnu OpenGL a toolkit GLUT a je implementována jako samostatný program DFRRT (Distance Field Rasterization and Ray Tracing). Úpravou metody pracující na GPU, traverzující mřížku algoritmem 3DDDA a užívající architektury CUDA byla vytvořena knihovna GPU DFRT. Tato knihovna může sloužit jako nový zobrazovací modul aplikace Myslbek.

4.1 Program DFRT

Vstupními parametry programu jsou požadované rozlišení obrazu, soubor s modelem, jméno adresáře s texturami okolí a jméno konfiguračního souboru. Pokud je zadané rozlišení shodné s aktuálním rozlišením monitoru, spustí se program v režimu na celou obrazovku. Konfigurační soubor obsahuje další nastavení související s paralelizací výpočtu a s použitým modelem. Konkrétně se zde nastavuje velikost obrazových dlaždic v pixelech, počet programových vláken, intenzita jednotlivých složek barvy modelu, koeficient odrazu a průhlednosti modelu, zorné pole kamery, maximální hloubka rekurze sledování paprsků, index lomu materiálu modelu a povolení použití Fresnelova faktoru pro výpočet osvětlení.

Po načtení všech parametrů nahraje program do hlavní paměti počítače i do paměti grafické karty textury mapy prostředí a zobrazovaný model a inicializuje proměnné a konstanty všech metod. Poté je spuštěna hlavní programová smyčka, ve které jsou nastavovány aktuální hodnoty parametrů a pozice kamery, zvolenou metodou je generován obraz a knihovnou CImg je zobrazován. Pozici kamery je možno měnit pomocí myši a některé parametry, jako

například koeficient odrazu a průhlednosti, lze měnit pomocí klávesnice. Také lze pomocí klávesnice vybrat požadovanou metodu generování obrazu.

Při generování obrazu je v závislosti na pozici kamery představované jedním bodem, vektorem směru pohledu a na něj kolmým vektorem udávajícím směr vzhůru každým pixelem vyslán jeden paprsek. Jestliže tento paprsek nezasáhne kvádr obklopující mřížku s modelem, je vrácen pouze příslušný vzorek, respektive bilineární interpolace příslušných vzorků textury okolí. Pokud mřížku protíná, traverzuje jí metodou závislou na zvoleném způsobu generování obrazu a hledá první průsečík s povrchem objektu. Když povrch nezasáhne, opět je barva pixelu, kterým byl vyslán, nastavena na barvu příslušného vzorku mapy prostředí. V případě nalezení průsečíku je vypočítán normálový vektor a pokud je koeficient odrazu nebo průhlednosti nenulový, jsou vyslány příslušné sekundární paprsky. Směr lomených paprsků je určen na základě nastaveného indexu lomu materiálu objektu. Jestliže bylo dosaženo limitu hloubky rekurze, netraverzují sekundární paprsky mřížku, ale pouze získají vzorek mapy okolí. K barvě vrácené sekundárními paprsky vynásobené příslušnými koeficienty a případně i koeficientem závisejícím na hodnotě Fresnelova faktoru je přidána barva materiálu a postupně je vyhodnocena barva primárního bodu průsečíku a tím i barva pixelu, kterým byl primární paprsek vyslán.

Program neumožňuje volit mezi traverzováním uniformní a dvouúrovňové mřížky za běhu. Stejně tak za běhu nelze změnit metodu nalezení průsečíku paprsku s povrchem nebo metodu výpočtu gradientu. Požadované nastavení je třeba provést před překladem programu definováním příslušných maker preprocesoru jazyka C v souboru **setup.h**. Pro metodu traverzující mřížku algoritmem 3DDDA na CPU bez použití SSE instrukcí se metoda nalezení průsečíku a výpočtu gradientu volí definováním šablony v tomtéž souboru. U metod využívajících jazyk OpenCL je třeba nastavení provést i v souboru **OpenCL_3DDDA.cl** a **OpenCL_CGT.cl**. Díky použití preprocesoru jazyka C a šablon funkcí může překladač program více optimalizovat a je tak dosahováno vyššího výkonu.

4.1.1 Traverzace algoritmem 3DDDA na CPU bez použití SSE instrukcí

Jak bylo uvedeno, metoda traverzující mřížku algoritmem 3DDDA na CPU bez použití SSE instrukcí byla implementována v bakalářské práci [14], ze které tato diplomová práce vychází. Implementace využívá šablon funkcí, pomocí kterých je možné vybrat, zda bude traverzována uniformní nebo dvouúrovňová mřížka a také lze zvolit metodu nalezení průsečíku a výpočtu gradientu. Navíc byla metoda nalezení průsečíku lineární interpolací a metoda výpočtu gradientu trilineární interpolací gradientů v rozích vypočítaných centrální diferencí implementována do jedné funkce, jelikož obě metody částečně používají stejná data a takto nemusí být opakovaně načítána.

Smyslem metody generování obrazu traverzací mřížky algoritmem 3DDDA na CPU bez použití SSE je vyzkoušet vliv různých metod nalezení průsečíku paprsku s povrchem objektu a metod výpočtu gradientu na rychlost vytvoření výsledného obrazu a na jeho kvalitu. Dalším jejím úkolem je sloužit jako referenční metoda pro měření výkonu ostatních metod. Jelikož bakalářská práce implementovala pro nalezení průsečíku paprsku s povrchem pouze metodu středního bodu a metodu lineární interpolace, byla dále implementována metoda opakované lineární interpolace, analytická metoda i metoda izolace a iterativního nalezení kořenů. K metodám výpočtu gradientu centrální diferencí, dopřednou diferencí, trilineární

interpolací gradientů v rozích voxelu určených centrální diferencí a k metodě analytické derivace interpolační funkce byla přidána metoda výpočtu gradientu trilineární interpolací gradientů v rozích voxelu určených dopřednou diferencí.

Jelikož jsou vyslané paprsky nezávislé, je snadné výpočet paralelizovat. Aby byla zachována datová souvislost a lépe se využilo cache pamětí, není generování obrazu paralelizováno na úrovni jednotlivých paprsků, ale obraz je rozdělen na uživatelem definované dlaždice čtvercového, případně obdélníkového tvaru a jejich zpracování je poté přidělováno jednotlivým programovým vláknům. Vzniká tím však problém s vyvažováním zátěže, neboť generování jednotlivých dlaždic má různou časovou náročnost. Je-li dlaždic výrazně více než výpočetních jednotek, projevuje se tento problém pouze minimálně. K paralelizaci bylo použito rozhraní OpenMP.

4.1.2 Traverzace algoritmem 3DDDA na CPU s použitím SSE instrukcí

Současné procesory jsou koncipovány tak, že umožňují zpracovat čtyři 32-bitová čísla s plovoucí desetinnou čárkou nebo čtyři 32-bitová celá čísla uložená do jednoho 128-bitového vektoru pomocí jediné instrukce. Slouží k tomu instrukční sada SSE. Zpracováním čtyř čísel jednou SSE instrukcí lze tedy teoreticky dosáhnout až čtyřnásobného výkonu celé aplikace. Ve výsledku je výkon nižší, neboť instrukční sada SSE neumožňuje nahradit úplně všechny běžné instrukce a svou roli hraje i omezená propustnost pamětí, neboť je potřeba zpracovat stejný objem dat.

Aby bylo možností SSE instrukcí maximálně využito, vytváří metoda traverzování mřížky algoritmem 3DDDA na CPU s použitím SSE čtvercové pakety o velikosti 2x2 paprsky. Data paprsků těchto paketů jsou uložena do čtyřsložkových 128-bitových SSE vektorů, přičemž data v jedné složce těchto vektorů patří jednomu paprsku. Například směrový vektor paprsku $\vec{d}$ byl původně zaznamenán pomocí tří 32-bitových čísel s plovoucí desetinnou čárkou. Nyní je zaznamenán pomocí tří 128-bitových SSE vektorů, respektive je uložen například v první složce těchto čtyřsložkových vektorů. Ostatní tři složky jsou využity dalšími třemi paprsky.

Po vytvoření paketů paprsků se vyhodnotí průsečík paprsků s kvádrem obklopujícím mřížku a je přistoupeno k traverzování. Paprsky v paketu využívají po celou dobu své existence stejný kód, ale jsou datově naprosto nezávislé. Pokud některý paprsek nemá pokračovat v traverzování z důvodu, že nezasáhl kvádr obklopující mřížku, dokončil traverzování mřížky, nebo nalezl průsečík s povrchem, je jeho bitová maska nastavena na nulu a data tohoto paprsku se dále nemění. Traverzování končí, pokud je maska všech čtyř paprsků paketu nulová.

Algoritmus 3DDDA vyžaduje v každém traverzačním kroku vyhodnocení, kterou hranicí paprsek opustí buňku a v závislosti na tom přičtení několika konstant k traverzačním proměnným. Jelikož mohou paprsky paketu opustit buňku různými hranicemi, je prováděno postupné přičtení konstant všech možností. Vždy ale pouze k těm paprskům, jejichž dočasně nastavená maska odpovídá dané situaci. Tento mechanismus velmi dobře funguje při traverzování uniformní mřížky, neboť je prováděno jen minimum operací navíc oproti traverzování bez SSE. Traverzují-li paprsky dvouúrovňovou mřížku, mohou v jednom traverzačním kroku traverzovat některé paprsky paketu hrubou mřížku a některé jemnou. Proto je třeba k paprskům příslušně přičítat konstanty obou částí traverzování a efektivita klesá.

Za účelem opětovného zvýšení efektivity byl implementován mechanismus rozdělení paketů na jednotlivé paprsky a dokončení traverzování mřížky těmito paprsky metodou nepo-

užívající SSE instrukce. Mechanismus lze aktivovat a konfigurovat pomocí maker v souboru **setup.h**. K rozdělení paketu může poté dojít, zbývá-li v paketu jeden, dva nebo tři paprsky a může být provedeno přímo v traverzovací funkci, po testu protnutí obklopujícího kvádru mřížky anebo po vyslání sekundárních paprsků. Přesto je metoda nejefektivnější při použití pouze uniformní mřížky. Při použití dvouúrovňové mřížky její výkon vzroste méně než u metody, která SSE nepoužívá.

4.1.3 Traverzace algoritmem CGT na CPU s použitím SSE instrukcí

Jak bylo uvedeno v kapitole 3.3.3, paprsky jsou při traverzování mřížky algoritmem CGT seskupeny do paketů a traverzují mřížku společně. Pro implementaci lze použít SSE instrukce obdobným způsobem jako v předchozí popsané metodě. Čtyři paprsky jsou sloučeny do čtvercových sub-paketů a uloženy pomocí čtyřsložkových 128-bitových SSE vektorů. Vyslaný paket je poté složen z $W \times H$ sub-paketů a tedy z $2\ W \times 2\ H$ paprsků. Jelikož je za účelem paralelizace výpočtu stejně jako u předchozích metod použito rozdělení obrazu na dlaždice, je pro optimální výkon vhodné volit rozměry paketu tak, aby beze zbytku dělily rozměry dlaždic. Požadované rozměry paketu lze nastavit v souboru **setup.h**.

Dále byla implementována možnost oříznout při přechodu z traverzování hrubé mřížky na jemnou mřížku ty buňky jemné mřížky, pro které buňka hrubé mřížky neindikuje přítomnost povrchu. Často však může být rychlejší ořezávání neprovádět a raději otestovat více buněk jemné mřížky. Proto lze tuto funkci povolit nebo zakázat nastavením příslušného makra v souboru **setup.h**.

Pro testovací účely byla metoda traverzování mřížky algoritmem CGT na CPU implementována tak, aby byla používána i pro sekundární paprsky. Tyto paprsky nemusí být koherentní a proto je třeba při hledání jejich průsečíku s povrchem traverzovat velkou část mřížky. Průsečíky jsou tak sice nalezeny, ale čas generování obrazu se výrazně zvýší. V praktickém nasazení je vhodnější použít pro sekundární paprsky algoritmus 3DDDA.

4.1.4 Traverzace algoritmem 3DDDA na GPU s použitím arch. CUDA

Mechanismus rekurzivního sledování paprsků, traverzování mřížky algoritmem 3DDDA, výpočet průsečíku paprsku s povrchem metodou lineární interpolace a několik způsobů výpočtu gradientu bylo implementováno i pomocí architektury CUDA. Pro dosažení vysokého výkonu jsou před spuštěním hlavní smyčky programu nahrána data distančního pole do hlavní paměti grafické karty a ukazatele na tato data jsou nahrány do paměti konstant. Do této paměti jsou také nahrány ostatní konstanty jako například barva modelu nebo rozměry mřížky distančního pole. Data, která se za běhu programu mohou měnit, například pozice kamery nebo index lomu materiálu, jsou ukládána do jedné datové struktury a ta je do paměti konstant kopírována před každým spuštěním kernelu.

Přizpůsobení výpočtu pro grafickou kartu je implementováno tak, že se každé vlákno programu stará o výpočet barvy jednoho pixelu. Vlákna jsou organizována do čtvercových bloků, jejichž velikost je ve výchozím nastavení 16×16 vláken. Hodnotu lze změnit v souboru **setup.h**. Jelikož architektura CUDA nepodporuje rekurzivní volání funkcí, je pomocí preprocesoru jazyka do zdrojového kódu vkládána funkce starající se o výpočet osvětlení vícekrát. Pokaždé však s jiným názvem a jinými názvy funkcí, které jsou dále volány. Také

je zajištěno, aby tyto funkce nebyly inlineovány, neboť každá z nich volá další funkci dvakrát (jednou pro odražený a jednou pro lomený paprsek) a vznikl by velmi dlouhý kód. Po spuštění kernelu každé vlákno vypočítá příslušný směr primárního paprsku a je volána funkce pro výpočet osvětlení podle nastavené hloubky rekurze.

Traverzační funkce jsou přizpůsobené, aby rozdílné větve podmínek byly co nejkratší a aby tak všechna vlákna bloku mohla po co nejdelší čas vykonávat stejný kód. Proto se v každém traverzačním kroku uniformní mřížky vždy nejprve vyhodnotí, kterou hranicí paprsek opustí voxel, uloží se který případ nastal a uloží se hodnota parametru t paprsku v místě opuštění voxelu, aby bylo možno vypočítat souřadnice tohoto bodu. Poté se pro všechny paprsky současně vyhodnotí, zda se v jejich aktuálním voxelu nalézá povrch a pokud ano, přistoupí se k výpočtu jejich průsečíku s povrchem a případně i k výpočtu gradientu. Nakonec se k traverzačním proměnným paprsků přičtou traverzační konstanty podle uloženého způsobu opuštění voxelu a přejde se k dalšímu traverzačnímu kroku. Při traverzování dvouúrovňové mřížky se v každém traverzačním kroku vyhodnocuje, zda paprsek traverzuje hrubou nebo jemnou mřížku a teprve podle toho se vykonává další kód.

Metoda výpočtu gradientu trilineární interpolací gradientů v rozích vypočítaných centrální diferencí vyžaduje načtení 24 hodnot distančního pole, neboť 8 hodnot bylo načteno již pro nalezení průsečíku a provedení jedné interpolace. Takovéto čtení dat z hlavní paměti je velmi pomalé. Proto je implementována i metoda výpočtu gradientu centrální diferencí, pro kterou je distanční pole uloženo do 3D textury. Získat hodnotu distanční funkce v jakémkoli místě prostoru lze poté pouhým zavoláním jedné funkce, která poskytne příslušnou hodnotu vypočítanou trilineární interpolací vzorků v okolí. Takto lze získat i hodnotu distanční funkce v bodě vstupu a výstupu paprsku z voxelu vyžadovanou při hledání průsečíku paprsku s povrchem metodou lineární interpolace. Použitím 3D textury pro uložení distančního pole se proces nalezení průsečíku a výpočtu gradientu značně urychluje.

Po vyhodnocení barvy pixelu je třeba uložit barvy do hlavní paměti grafické karty, odkud jsou po dokončení kernelu kopírovány a předávány knihovně CImg, která se stará o zobrazení vygenerovaného obrazu. Aby každé vlákno nemuselo zapisovat do hlavní paměti tři hodnoty datového typu `unsigned char` zvlášť, zapisuje vypočítanou barvu pouze do sdílené paměti. Některé vlákna se poté postarají o zkopírování dat ze sdílené paměti do hlavní paměti po větších částech přetypováním dat na `unsigned int`. Podrobněji je takováto strategie popsána v kapitole 2.1.2.

4.1.5 Traverzace algoritmem CGT na GPU s použitím arch. CUDA

Použitím algoritmu CGT by mělo dojít k zamezení vykonávání odlišných větví kódu různými vlákny jednoho bloku a díky tomu by tato metoda měla být rychlejší než traverzace algoritmem 3DDDA. Velikost jednotlivých bloků však musí být volena menší, aby pakety nepokrývaly příliš velké množství buněk mřížky. Proto je výchozí velikost nastavitelná v souboru **setup.h** nastavena na 8×8 vláken. Nevýhodou algoritmu CGT je potřeba většího množství traverzačních proměnných než u algoritmu 3DDDA. Také je třeba indexovat proměnné dynamicky pomocí indexu hlavní osy a vedlejších os šíření paprsku $\vec{K}$, $\vec{U}$ a $\vec{V}$. Kvůli tomu musí překladač takto indexované proměnné, mezi které patří například transformovaný bod počátku paprsku $\vec{O}$ nebo směrový vektor paprsku $\vec{d}$, umístit do lokální paměti, ke které má program velmi pomalý přístup.

Počet proměnných byl snížen a dynamické indexování bylo odstraněno implementací šesti specializovaných traverzačních funkcí. Jedné pro kladný a jedné pro záporný směr v každé možné hlavní ose šíření. Generování obrazu je díky tomuto přístupu rychlejší, avšak při překladu vzniká kvůli inlineování mnoha funkcí velmi dlouhý kód. I když se traverzování mřížky algoritmem CGT používá pouze pro primární paprsky a pro sekundární paprsky se používá algoritmus 3DDDA, neporadí si překladač s tvorbou kódu umožňujícího rekurzivní sledování paprsků do větší hloubky než 2.

Čtení binárního pole indikujícího přítomnost povrchu ve voxelu bylo optimalizováno tvorbou tří speciálních polí, jednoho pro každou možnou hlavní osu šíření paketu. V těchto polích je uloženo 32 bitů indikujících přítomnost povrchu ve 32 voxelech do jednoho čtyř-bytu. Tyto voxely leží v jedné řádce jednoho plátku binárního pole vytvořeného rozdělením tohoto pole podél hlavní osy šíření. Paket pokrývá obdélníkovou oblast plátku a voxely jsou testovány po řádcích. Před testováním řádku načte požadovaná data pouze několik vláken do sdílené paměti a ostatní vlákna poté k těmto datům mohou přistupovat. Čtení hlavní paměti grafické karty navíc může být zarovnáno. Hledání voxelů obsahujících povrch takto probíhá mnohem rychleji, než kdyby vlákna musela pro každý voxel číst z hlavní paměti karty nezarovnaně jeden byte indikující přítomnost povrchu. Pro hledání buněk hrubé mřížky obsahujících povrch při traverzování dvouúrovňové mřížky byl implementován obdobný mechanismus.

V inicializační fázi algoritmu CGT je třeba nalézt plátek, ve kterém paket vstupuje do mřížky a také plátek, ve kterém z mřížky celý vystupuje. Za účelem nalezení těchto plátků je třeba nalézt minimální nebo maximální hodnotu průsečíku paprsků paketu s kvádrem obklopujícím mřížku v ose $\vec{K}$. Tyto extrémní hodnoty lze získat paralelní redukcí hodnot jednotlivých vláken bloku. Proto byla implementována paralelní redukce vysvětlená v prezentaci [11], která nejprve redukuje nejvzdálenější prvky a až v posledním kroku prvky sousední, díky čemuž je větší množství dat čteno z různých bank a nedochází k tolika konfliktům. Dále je třeba nalézt obklopující obdélník paprsků vstupujících do prvního plátku. K tomu je potřeba získat minimální i maximální hodnotu průsečíků v každé z vedlejších os. Proto byla implementována i upravená paralelní redukce, která efektivně redukuje hodnoty dvěma směry a vrátí minimální i maximální nalezenou hodnotu.

Pro tuto i předchozí metodu generování obrazu byla implementována efektivní paralelizace na více grafických karet. Na každou kartu jsou nahrána všechna potřebná data a každá karta poté generuje samostatný obraz. Celkový čas generování více obrazů je tak stejný jako čas generování jednoho obrazu na nejpomalejší z karet. Toho lze výhodně využít při stereoskopické projekci.

4.1.6 Traverzace algoritmem 3DDDA na GPU s použitím jazyka OpenCL

Traverzování uniformní i dvouúrovňové mřížky na GPU algoritmem 3DDDA bylo implementováno i pomocí jazyka OpenCL. Zdrojový kód je velmi podobný kódu pro architekturu CUDA. Výraznější rozdíly jsou pouze v předávání většího množství parametrů funkcím, neboť jazyk OpenCL neumožňuje použít globální proměnné pro textury. Také není možné, aby vstupním parametrem funkce byla reference na proměnnou. Proto je pro proměnné, jejichž hodnotu má funkce měnit, předáván ukazatel na tyto proměnné uložené v jednoprvkových polích. Rekurzivní sledování paprsků v jazyce OpenCL implementováno nebylo.

Stejně jako v implementaci algoritmu 3DDDA pro architekturu CUDA byla implementována funkce pro výpočet gradientu trilineární interpolací gradientů v rozích voxelu získaných centrální diferencí, která využívá distanční pole uložené v globální paměti jako běžné lineární pole a funkce pro výpočet gradientu centrální diferencí, která používá distanční pole uložené do 3D textury. Současná verze překladače jazyka OpenCL si překvapivě lépe poradí s překladem první uvedené funkce a takovýto kód je výrazně rychlejší než kód využívající 3D texturu. Přesto je tato implementace oproti implementaci pro architekturu CUDA mnohem pomalejší.

4.1.7 Traverzace algoritmem CGT na GPU s použitím jazyka OpenCL

Oproti algoritmu CGT implementovanému na GPU s použitím architektury CUDA nepoužívá verze v jazyce OpenCL specializované funkce pro různé hlavní osy šíření paketu, neboť si překladač dovede lépe poradit s dynamicky indexovanými proměnnými. Jednotlivé pracovní jednotky pracovní skupiny zpracovávající jeden paket paprsků spolupracují pouze v inicializační fázi algoritmu CGT při vykonávání paralelních redukcí implementovaných obdobně jako u arch. CUDA. Traverzační fázi vykonávají pouze pracovní jednotky starající se o paprsek, který ještě hledá průsečík s povrchem. Ostatní pracovní jednotky jsou nečinné. V traverzační fázi tedy není prováděna žádná synchronizace. Jakmile paprsek pracovní jednotky zasáhne povrch, opustí pracovní jednotka traverzační funkci, vyhodnotí barvu pixelu, zapíše ji do lokální paměti a čeká, až i ostatní pracovní jednotky dokončí traverzování. Tato strategie pracuje při traverzování dvouúrovňové mřížky velmi dobře a specializované traverzační funkce implementované s použitím arch. CUDA nedosahují ani dvojnásobného výkonu. Navíc není nutné používat specializovaná binární pole, jejichž aktualizace v případě změny distančního pole je časově náročnější. Překladač kódu pro architekturu CUDA si však nedokáže s podobným schématem dobře poradit a proto nelze algoritmus CGT takto implementovat i pro arch. CUDA a očekávat vysoký výkon aplikace.

4.1.8 Generování obrazu na CPU a GPU současně

Za účelem maximálního využití dostupného hardware byla implementována metoda, která ke generování obrazu využívá současně procesor i grafickou kartu. Jejím základem je horizontální rozdělení obrazu na dvě části a tvorba horní části algoritmem 3DDDA na GPU s použitím architektury CUDA a dolní části algoritmem CGT na CPU s použitím SSE instrukcí. Byly tedy zvoleny ty nejrychlejší metody, aby celá metoda byla co nejrychlejší.

Jelikož může být poměr výkonnosti procesoru a grafické karty na různých počítačích rozdílný a jelikož rychlost generování různých částí obrazu výrazně závisí na obsahu, je rozdělení obrazu dynamicky vyvažováno. Obraz je pomyslně rozdělen na řádky, jejichž výška je rovna výšce dlaždic vytvořených pro paralelizaci výpočtu na CPU. Na počátku je polovina těchto řádků přidělena grafické kartě a druhá polovina procesoru. Při generování obrazu je měřen čas obou metod a rozdělení řádků je poté upraveno, aby předpokládaný čas generování dalšího obrazu byl pro obě metody co nejbližší. Díky tomu je hardware vždy maximálně vytížen a výsledný čas je nejnižší možný. Pro testovací účely je uživateli dovoleno měnit rozdělení obrazu i manuálně.

4.2 Program DFRRT

Vstupní parametry i konfigurační soubory jsou u programu DFRRT shodné s programem DFRT. Také se zde nalézá soubor **setup.h**, který dovoluje nastavit různé možnosti překladu programu. Úkolem tohoto programu je otestovat vliv metody omezení intervalu traverzování primárních paprsků rasterizací buněk jemné nebo hrubé mřížky obsahujících povrch na rychlost generování obrazu. Proto jsou po spuštění programu, načtení konfiguračního souboru a nahrání modelu vytvořeny polygonální krychle o velikosti zvolených buněk obsahujících povrch. Aby rasterizace těchto krychlí byla co nejrychlejší, jsou po vytvoření nahrány do vertex buffer objektu.

Program využívá grafickou knihovnu OpenGL. V hlavní smyčce programu jsou s její pomocí krychle rasterizovány do připojeného frame buffer objektu obsahujícího pouze buffer hloubky. Je nezbytné, aby k této rasterizaci byla použita projekční matice odpovídající generování primárních paprsků u metody sledování paprsků. Takováto matice je vytvořena nastavením stejného zorného pole kamery a poměru stran obrazu jako při tvorbě primárních paprsků. Hloubkový buffer je po rasterizaci předán kernelu architektury CUDA, který získanou minimální vzdáleností povrchu od kamery v každém pixelu zmenší interval traverzování paprsků a poté traverzuje mřížku algoritmem 3DDDA. V souboru **setup.h** je možné nastavit, jestli mají paprsky traverzovat uniformní nebo dvouúrovňovou mřížku. Jelikož bylo odstraněno traverzování prázdného prostoru před modelem a paprsky, které model míjejí ve větší vzdálenosti nemusí traverzovat vůbec, bývá rychlejší traverzovat primárními paprsky pouze uniformní mřížku. Pro sekundární paprsky lze nastavení provést samostatně.

Kombinací rasterizace buněk mřížky obsahujících povrch a traverzací mřížky algoritmem 3DDDA na GPU s použitím architektury CUDA je dosahováno vysoké rychlosti generování obrazu. Nevýhodou je omezení prostoru, ve kterém může být model zobrazen, přední a zadní ořeznou rovinou, které kvůli omezené přesnosti paměti hloubky nemohou být nastaveny libovolně. Další nevýhodou je potřeba tvorby polygonálních krychlí pro všechny buňky obsahující povrch. Při časté aktualizaci distančního pole se tak může výkon aplikace značně snížit. Jiným problémem je efektivní paralelizace výpočtu na více grafických karet. Při rozdělení obrazu na více částí by bylo stále třeba provádět rasterizaci a výkon by tedy nevzrostl úměrně počtu karet. Ideálním řešením by bylo generovat na každé kartě jiný obraz. K tomu by však bylo třeba provádět rasterizaci na každé kartě zvlášť, k čemuž možnosti toolkitu GLUT nedostačují.

4.3 Zobrazovací knihovna GPU DFRT

Úkolem této knihovny je sloužit jako rychlý zobrazovací modul existujícího systému pro rozšířenou virtuální realitu nazvaného Myslbek. Jelikož jsou zobrazovaná volumetrická data průběžně upravována, bylo by u nejrychlejší metody pracující na GPU, u programu DFRRT potřeba stále aktualizovat seznam polygonálních krychlí. To by mohlo mít nepříznivý dopad na celkový výkon systému. Proto byla použita druhá nejrychlejší metoda pracující pouze na GPU, která tuto nevýhodu nemá. Jedná se o metodu traverzující mřížku algoritmem 3DDDA a využívající architekturu CUDA. Jelikož je u této metody ve většině případů rychlejší traverzovat pouze uniformní mřížku, nebylo traverzování dvouúrovňové mřížky implementováno, aby v paměti grafické karty nemusela být uchovávána hrubá mřížka.

Voxely jemné mřížky a distanční pole jsou uloženy do 3D textur, aby mohly být v kernelu rychle čteny a zároveň bylo snadné provést při úpravě distančního pole snadnou a rychlou aktualizaci změněné oblasti. Program Myslbek uchovává barvu objektu v různých částech prostoru pomocí dalšího trojrozměrného pole o stejném rozlišení jaké má distanční pole. I toto pole je uloženo na grafickou kartu jako 3D textura. Výpočet barvy bodů na povrchu objektu zasažených vyslanými primárními paprsky je proveden na základě ambientního osvětlení, polohy světel, jejich difúzní a lesklé složky světla, v textuře uložené barvy objektu a pro celý objekt definované lesklosti objektu. V knihovně je také možné povolit vyslání stínových paprsků ke světlům a pokud je mezi vyšetřovaným bodem a zdrojem světla nějaká překážka, světelný příspěvek zdroje se nezapočítá. Tím na povrchu objektu vznikají stíny, díky nimž je dosaženo realističtějšího vzhledu objektu.

Program Myslbek byl upraven, aby mohl být pro testovací účely po připojení Wii ovladače provozován na libovolném počítači. Do této testovací verze byla zobrazovací knihovna přidána. Součástí knihovny jsou kromě funkce generující obraz i funkce pro uložení distančního pole, pole barev a mřížky indikující přítomnost povrchu do paměti grafické karty volané při načítání nového objektu ze souboru. Dále jsou implementovány funkce pro vyprázdnění všech datových struktur i funkce určené pro rychlou aktualizaci změněných částí datových polí. Použitím vytvořené zobrazovací knihovny je obrazový výstup programu Myslbek mnohem kvalitnější oproti původnímu způsobu vizualizace dat, zejména při větším přiblížení obráběného objektu.

Kapitola 5

Výsledky

V této kapitole jsou prezentovány výsledky dosažené pomocí různých nastavení všech implementovaných metod. Měření časů generování jednotlivých snímků bylo provedeno na dvou počítačových sestavách, jejichž parametry jsou uvedeny v tabulce 5.1. Na počítači 1 byl nainstalován operační systém Windows i Linux a díky tomu mohl být změřen vliv použití dvou různých systémů, překladačů a ovladačů grafických karet na výkon programu.

	počítač 1	počítač 2
Operační systém	Windows XP 32-bit Fedora Linux 32-bit	Windows 7 64-bit
Překladač jazyka C++	MS Visual Studio 2008 gcc 4.3	MS Visual Studio 2008
Ovladače grafické karty	197.13 WinXP 32-bit 195.36.15 Linux	197.16 Win7 64-bit notebook
Verze architektury CUDA Verze jazyka OpenCL	3.0 1.0	3.0 1.0
Procesor Počet jader Frekvence procesoru Velikost L2 Cache Velikost operační paměti	Intel Core 2 Quad Q9550 4 2,83 GHz 12 MB 3 GB	Intel Core 2 Duo P8400 2 2,26 GHz 3 MB 4 GB
Grafická karta Compute Capability Počet multiprocesorů Frekvence jádra Frekvence procesorů Frekvence pamětí Šířka paměťové sběrnice Velikost grafické paměti	nVidia GeForce 9800 GX2 1.1 32 (16 na GPU) 600 MHz 1500 MHz 1000 MHz 512-bitů 1 GB (512 MB na GPU)	nVidia GeForce 9600M GT 1.1 4 500 MHz 1250 MHz 800 MHz 128-bitů 512 MB

Tabulka 5.1: Parametry počítačových sestav použitých k testům.

K měření bylo použito tří různých objektů uložených pomocí distančního pole. Konkrétně

se jedná o objekt Stanford Bunny, Stanford Dragon a Stanford Happy Buddha. Model každého z objektů byl k dispozici ve třech rozlišeních. Čas generování obrazu byl tedy celkem měřen na devíti modelech, jejichž ukázky jsou na obrázcích 5.1 a 5.2. Model Bunny byl uložen v distančním poli o rozlišení $64 \times 64 \times 52$, $128 \times 128 \times 100$ a $256 \times 256 \times 200$ voxelů, model Dragon v poli o rozlišení $64 \times 48 \times 32$, $128 \times 92 \times 60$ a $256 \times 186 \times 116$ voxelů a model Buddha v poli o rozlišení $28 \times 64 \times 28$, $56 \times 128 \times 56$ a $108 \times 256 \times 108$ voxelů. Buňky hrubé mřížky všech modelů byly tvořeny $4 \times 4 \times 4$ buňkami jemné mřížky. Obraz byl generován v rozlišení 512×512, 1024×768 a 1280×1024 pixelů. Prezentované časy a počty paprsků jsou průměrem hodnot získaných vygenerováním 36ti snímků, ve kterých model postupně rotuje kolem svislé osy s krokem 10 stupňů.

model	rozlišení obrazu [pixelů]	rozlišení modelu [voxelů]	paprsků celkem abs.	traverz. mřížku abs.	[%]	naléz. povrch abs.	[%]	kroků na paprsek dvouúrov. hrubá	jemná	uniform.
Bunny	512x512	64x64x52	262 144	167 051	64	31 815	12	8,04	3,11	32,62
		128x128x100	262 144	160 986	61	37 006	14	15,07	3,23	60,84
		256x256x200	262 144	160 986	61	42 529	16	28,84	3,28	116,00
	1024x768	64x64x52	786 432	375 898	48	71 586	9	8,04	3,11	32,62
		128x128x100	786 432	362 207	46	83 261	11	15,07	3,23	60,84
		256x256x200	786 432	362 207	46	95 689	12	28,84	3,28	116,01
	1280x1024	64x64x52	1 310 720	668 389	51	127 267	10	8,04	3,11	32,61
		128x128x100	1 310 720	643 912	49	148 021	11	15,07	3,23	60,85
		256x256x200	1 310 720	643 912	49	170 107	13	28,84	3,28	116,02
Dragon	512x512	64x48x32	262 144	85 921	33	17 827	7	6,44	4,04	26,25
		128x92x60	262 144	78 783	30	20 821	8	11,62	4,16	47,13
		256x184x116	262 144	77 150	29	23 830	9	21,75	4,03	87,76
	1024x768	64x48x32	786 432	193 245	25	40 112	5	6,44	4,04	26,26
		128x92x60	786 432	177 285	23	46 844	6	11,62	4,15	47,13
		256x184x116	786 432	173 527	22	53 614	7	21,76	4,03	87,79
	1280x1024	64x48x32	1 310 720	343 561	26	71 314	5	6,45	4,04	26,25
		128x92x60	1 310 720	315 241	24	83 276	6	11,62	4,15	47,11
		256x184x116	1 310 720	308 430	24	95 321	7	21,76	4,03	87,81
Buddha	512x512	28x64x28	262 144	218 831	83	75 077	29	5,33	6,37	22,24
		56x128x56	262 144	218 831	83	82 518	31	10,34	5,28	42,22
		108x256x108	262 144	218 831	83	92 249	35	18,66	5,30	75,66
	1024x768	28x64x28	786 432	492 298	63	168 921	21	5,33	6,37	22,24
		56x128x56	786 432	492 298	63	185 667	24	10,34	5,28	42,22
		108x256x108	786 432	492 298	63	207 556	26	18,66	5,30	75,66
	1280x1024	28x64x28	1 310 720	875 287	67	300 304	23	5,33	6,37	22,24
		56x128x56	1 310 720	875 287	67	330 084	25	10,34	5,28	42,22
		108x256x108	1 310 720	875 287	67	368 994	28	18,66	5,30	75,65

Tabulka 5.2: Přehled počtu paprsků vyslaných, traverzujících mřížku a nalézajících povrch a průměrný počet traverzačních kroků na paprsek traverzující mřížku pro různé modely a různá rozlišení.

5.1 Přehled počtu sledovaných paprsků a počtu traverzačních kroků na paprsek

Tabulka 5.2 poskytuje přehled počtu vyslaných paprsků, který je roven rozlišení obrazu, počtu paprsků, které zasáhnou kvádr obklopující mřížku a z toho důvodu i mřížku traverzují a počtu paprsků, které naleznou povrch. Tyto počty jsou uvedeny pro všechny testované objekty ve všech jejich rozlišeních a pro všechna testovaná rozlišení obrazu. Kromě průměrného počtu paprsků traverzujících mřížku a nalézajících povrch je uveden i počet procent těchto paprsků z celkového počtu vyslaných paprsků.

Dále jsou v tabulce 5.2 uvedeny průměrné počty traverzačních kroků na paprsek traverzující mřížku. Počty kroků jsou uvedeny pro traverzaci uniformní mřížky i pro traverzaci dvouúrovňové mřížky. U traverzace dvouúrovňové mřížky jsou hodnoty rozděleny na počet traverzačních kroků, které paprsek průměrně vykoná v hrubé mřížce a počet kroků, které paprsek průměrně vykoná v jemné mřížce.

5.2 Přehled časů generování obrazu všemi implementovanými metodami

Tabulka 5.3 obsahuje časy generování obrazu traverzací uniformní mřížky algoritmem 3DDDA pro všechny modely ve všech dostupných rozlišeních modelů a všech třech testovaných rozlišeních obrazu na počítači 1 v operačním systému Windows. Pro maximální využití hardwaru jsou použita čtyři programová vlákna. Grafická karta počítače 1 se skládá ze dvou karet, které pro obecné výpočty mohou pracovat pouze samostatně. Proto je pro měření použita jen jedna karta, tedy 16 multiprocesorů. Díky implementaci efektivní paralelizace popsané v kapitole 4.1.5 mohou být u metod traverzace algoritmem 3DDDA nebo CGT na GPU s použitím architektury CUDA za uvedené časy vygenerovány dva obrazy.

Metoda traverzující mřížku algoritmem 3DDDA na CPU bez použití SSE instrukcí slouží jako referenční pro porovnání relativní rychlosti generování obrazu ostatními metodami. V tabulce jsou poté uvedeny i relativní rychlosti jednotlivých metod oproti referenční metodě vyjádřené v procentech. Tabulka 5.4 je pokračováním tabulky 5.3 a obsahuje kromě opětovného uvedení časů referenční metody i časy a relativní rychlosti generování obrazu traverzací uniformní mřížky algoritmem CGT a výsledky metody využívající CPU i GPU současně.

Referenční metoda používá k nalezení průsečíku paprsku s povrchem objektu v buňkách obsahujících povrch metodu lineární interpolace. Gradient je počítán interpolací gradientů v rozích získaných centrální diferencí. Ostatní dvě metody pracující na CPU používají stejné algoritmy, jiné pro ně ani nebyly implementovány. Velikost obrazových dlaždic je vždy 64×64 pixelů. U metody traverzace algoritmem 3DDDA na CPU s použitím SSE instrukcí je vypnuto rozdělení paketu při nižším počtu aktivních paprsků. Velikost paketu u metody traverzace algoritmem CGT na CPU je nastavena na 8×8 paprsků a ořez buněk jemné mřížky je vypnutý.

U metod pracujících na GPU je pro nalezení průsečíku také používána lineární interpolace a gradient je počítán centrální diferencí. Metody traverzace algoritmem 3DDDA na GPU využívající architekturu CUDA a jazyk OpenCL mají velikost bloku, respektive velikost

model	rozlišení obrazu [pixelů]	rozlišení modelu [voxelů]	3DDDA								
			CPU			GPU					
				SSE		CUDA		OpenCL		DFRRT	
			čas [ms]	čas [ms]	rel. [%]	čas [ms]	rel. [%]	čas [ms]	rel. [%]	čas [ms]	rel. [%]
Bunny	512x512	64x64x52	51	17	293	11	460	177	29	11	454
		128x128x100	78	27	292	16	494	369	21	10	753
		256x256x200	140	48	293	36	384	756	18	16	884
	1024x768	64x64x52	123	41	302	24	518	359	34	16	790
		128x128x100	181	60	302	32	559	754	24	19	946
		256x256x200	316	104	303	63	498	1525	21	28	1148
	1280x1024	64x64x52	214	70	306	40	536	534	40	26	823
		128x128x100	316	103	306	53	592	1148	27	30	1060
		256x256x200	552	179	308	94	585	2355	23	41	1347
Dragon	512x512	64x48x32	30	10	298	6	475	82	36	7	423
		128x92x60	38	13	294	7	525	156	24	8	483
		256x184x116	57	20	282	14	407	310	18	12	482
	1024x768	64x48x32	77	25	308	13	581	158	48	13	572
		128x92x60	95	32	301	14	673	312	30	13	705
		256x184x116	135	46	295	26	525	622	22	18	745
	1280x1024	64x48x32	131	42	310	21	622	240	55	19	701
		128x92x60	163	54	304	21	763	486	34	22	758
		256x184x116	234	78	299	39	607	984	24	27	877
Buddha	512x512	28x64x28	55	21	265	8	676	123	44	9	584
		56x128x56	80	29	273	17	472	309	26	13	606
		108x256x108	124	45	277	27	467	626	20	18	701
	1024x768	28x64x28	130	48	270	17	760	234	56	17	754
		56x128x56	186	67	279	37	498	614	30	26	712
		108x256x108	280	99	284	55	508	1228	23	35	792
	1280x1024	28x64x28	226	83	273	27	824	309	73	25	913
		56x128x56	325	114	284	62	527	884	37	39	843
		108x256x108	491	170	289	90	547	1840	27	52	939

Tabulka 5.3: Přehled časů generování obrazu všemi implementovanými metodami při traverzování pouze uniformní mřížky v OS Windows na počítači 1. První část tabulky: Časy při traverzaci algoritmem 3DDDA na CPU s použitím i bez použití SSE instrukcí, na GPU s použitím architektury CUDA i jazyka OpenCL a časy programu DFRRT. Dále také relativní rychlost generování obrazu oproti referenční metodě – traverzování uniformní mřížky algoritmem 3DDDA na CPU bez použití SSE instrukcí.

model	rozlišení obrazu [pixelů]	rozlišení modelu [voxelů]	3D-DDA CPU čas [ms]	CGT CPU SSE čas [ms]	CGT CPU SSE rel. [%]	CGT GPU CUDA čas [ms]	CGT GPU CUDA rel. [%]	CGT GPU OpenCL čas [ms]	CGT GPU OpenCL rel. [%]	CPU+GPU čas [ms]	CPU+GPU rel. [%]
Bunny	512x512	64x64x52	51	10	535	23	226	50	103	8	627
		128x128x100	78	15	518	62	126	154	50	12	665
		256x256x200	140	37	382	241	58	785	18	26	528
	1024x768	64x64x52	123	23	548	33	379	83	150	16	766
		128x128x100	181	30	611	76	239	195	93	21	855
		256x256x200	316	56	564	248	127	763	41	40	789
	1280x1024	64x64x52	214	36	592	44	485	116	183	25	851
		128x128x100	316	46	691	93	339	248	127	33	959
		256x256x200	552	78	712	273	202	833	66	56	983
Dragon	512x512	64x48x32	30	7	414	15	198	27	109	6	518
		128x92x60	38	10	381	38	101	81	47	7	534
		256x184x116	57	22	261	138	41	423	13	14	418
	1024x768	64x48x32	77	18	421	22	352	44	173	11	673
		128x92x60	95	22	434	46	207	91	104	12	766
		256x184x116	135	35	385	141	96	392	35	21	649
	1280x1024	64x48x32	131	30	443	28	473	61	213	17	771
		128x92x60	163	34	476	53	308	105	155	18	920
		256x184x116	234	50	470	146	161	388	60	29	801
Buddha	512x512	28x64x28	55	12	473	18	311	23	238	8	715
		56x128x56	80	15	553	36	221	73	110	12	672
		108x256x108	124	23	542	96	129	215	58	18	696
	1024x768	28x64x28	130	28	469	27	479	38	341	14	918
		56x128x56	186	32	575	50	372	117	160	24	791
		108x256x108	280	44	639	115	243	271	104	33	853
	1280x1024	28x64x28	226	46	490	40	566	58	391	22	1049
		56x128x56	325	52	622	70	465	163	199	37	885
		108x256x108	491	68	727	149	330	352	140	50	981

Tabulka 5.4: Přehled časů generování obrazu všemi implementovanými metodami při traverzování pouze uniformní mřížky v OS Windows na počítači 1. Druhá část tabulky: Časy při traverzaci algoritmem CGT na CPU s použitím SSE instrukcí, na GPU s použitím architektury CUDA i jazyka OpenCL a časy generování obrazu při použití CPU a GPU současně. Dále také relativní rychlost generování obrazu oproti referenční metodě.

model	rozlišení obrazu [pixelů]	rozlišení modelu [voxelů]	3DDDA								
			CPU			GPU					
				SSE		CUDA		OpenCL		DFRRT	
			čas [ms]	čas [ms]	rel. [%]	čas [ms]	rel. [%]	čas [ms]	rel. [%]	čas [ms]	rel. [%]
Bunny	512x512	64x64x52	26	13	199	11	228	149	17	13	205
		128x128x100	29	17	174	18	157	251	11	16	185
		256x256x200	34	23	148	26	127	478	7	22	153
	1024x768	64x64x52	68	31	217	22	311	278	24	22	302
		128x128x100	73	38	192	37	199	514	14	30	241
		256x256x200	83	51	163	54	153	1023	8	42	199
	1280x1024	64x64x52	114	52	218	34	339	434	26	35	328
		128x128x100	124	64	193	59	209	814	15	45	272
		256x256x200	141	86	164	87	162	1638	9	62	227
Dragon	512x512	64x48x32	21	9	235	8	268	95	22	11	194
		128x92x60	21	10	212	10	208	138	16	13	162
		256x184x116	23	13	185	14	164	212	11	15	156
	1024x768	64x48x32	56	22	253	14	401	162	35	18	317
		128x92x60	59	25	235	19	315	250	23	20	286
		256x184x116	61	30	205	26	234	416	15	26	239
	1280x1024	64x48x32	94	37	253	21	458	245	38	26	366
		128x92x60	97	42	231	27	354	393	25	29	340
		256x184x116	103	50	205	40	260	685	15	38	275
Buddha	512x512	28x64x28	39	20	190	14	276	180	22	15	252
		56x128x56	40	24	169	20	199	274	15	21	195
		108x256x108	45	30	151	29	157	436	10	27	165
	1024x768	28x64x28	96	48	202	28	339	344	28	30	317
		56x128x56	99	54	182	40	247	554	18	39	251
		108x256x108	109	67	162	59	186	958	11	54	202
	1280x1024	28x64x28	165	81	204	45	370	519	32	44	375
		56x128x56	169	92	184	64	265	879	19	60	280
		108x256x108	186	113	164	92	202	1491	12	81	229

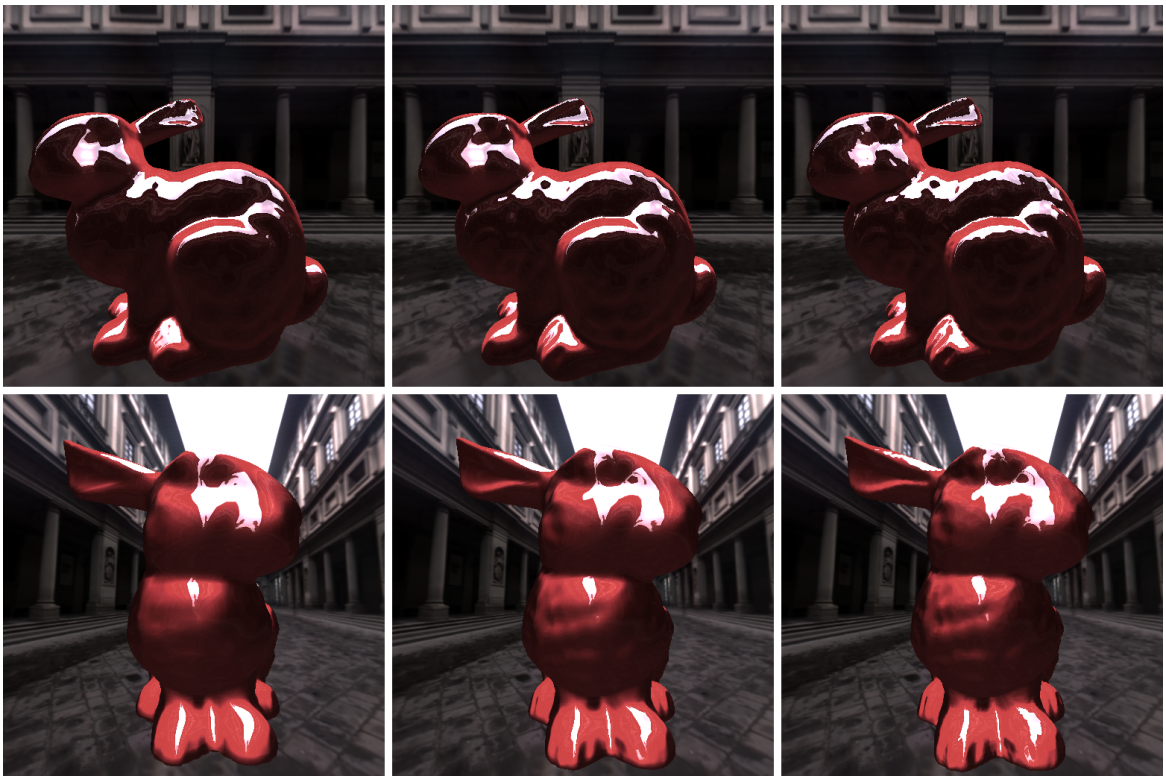
Tabulka 5.5: Přehled časů generování obrazu všemi implementovanými metodami při traverzování dvouúrovňové mřížky v OS Windows na počítači 1. První část tabulky: Časy při traverzaci algoritmem 3DDDA na CPU s použitím i bez použití SSE instrukcí, na GPU s použitím architektury CUDA i jazyka OpenCL a časy programu DFRRT. Dále také relativní rychlost generování obrazu oproti referenční metodě – traverzování dvouúrovňové mřížky algoritmem 3DDDA na CPU bez použití SSE instrukcí.

model	rozlišení obrazu [pixelů]	rozlišení modelu [voxelů]	3D-DDA CPU čas [ms]	CGT CPU SSE čas [ms]	CGT CPU SSE rel. [%]	CGT GPU CUDA čas [ms]	CGT GPU CUDA rel. [%]	CGT GPU OpenCL čas [ms]	CGT GPU OpenCL rel. [%]	CPU+GPU čas [ms]	CPU+GPU rel. [%]
Bunny	512x512	64x64x52	26	9	303	17	150	25	103	7	354
		128x128x100	29	11	250	39	73	67	43	11	272
		256x256x200	34	21	160	117	29	243	14	18	183
	1024x768	64x64x52	68	21	324	26	259	42	163	15	463
		128x128x100	73	24	299	50	145	91	80	19	380
		256x256x200	83	35	237	127	65	258	32	29	282
	1280x1024	64x64x52	114	34	340	35	323	58	196	21	533
		128x128x100	124	38	323	62	199	117	106	30	413
		256x256x200	141	50	280	139	101	278	51	42	338
Dragon	512x512	64x48x32	21	7	305	12	166	17	122	6	368
		128x92x60	21	9	249	27	79	42	52	8	268
		256x184x116	23	15	150	81	29	161	14	12	194
	1024x768	64x48x32	56	17	320	19	300	27	208	11	510
		128x92x60	59	20	296	34	170	50	117	13	445
		256x184x116	61	27	227	86	71	152	40	19	317
	1280x1024	64x48x32	94	28	332	25	383	37	254	16	596
		128x92x60	97	31	311	41	237	60	163	20	491
		256x184x116	103	39	264	90	115	151	69	26	393
Buddha	512x512	28x64x28	39	11	347	17	228	22	180	10	384
		56x128x56	40	13	303	31	129	42	96	12	334
		108x256x108	45	18	244	68	66	97	47	16	275
	1024x768	28x64x28	96	27	354	27	357	38	256	18	523
		56x128x56	99	30	329	44	223	67	149	23	437
		108x256x108	109	37	296	86	128	127	86	30	363
	1280x1024	28x64x28	165	45	366	40	413	56	292	28	581
		56x128x56	169	49	344	61	276	93	181	35	478
		108x256x108	186	57	324	108	172	160	116	46	405

Tabulka 5.6: Přehled časů generování obrazu všemi implementovanými metodami při traverzování dvouúrovňové mřížky v OS Windows na počítači 1. Druhá část tabulky: Časy při traverzaci algoritmem CGT na CPU s použitím SSE instrukcí, na GPU s použitím architektury CUDA i jazyka OpenCL a časy generování obrazu při použití CPU a GPU současně. Dále také relativní rychlost generování obrazu oproti referenční metodě.

pracovní skupiny nastavenu na 16×16 vláken, respektive pracovních jednotek. Pro metody travertace algoritmem CGT na GPU je velikost bloku, respektive pracovní skupiny nastavena na 8×8 vláken, respektive pracovních jednotek.

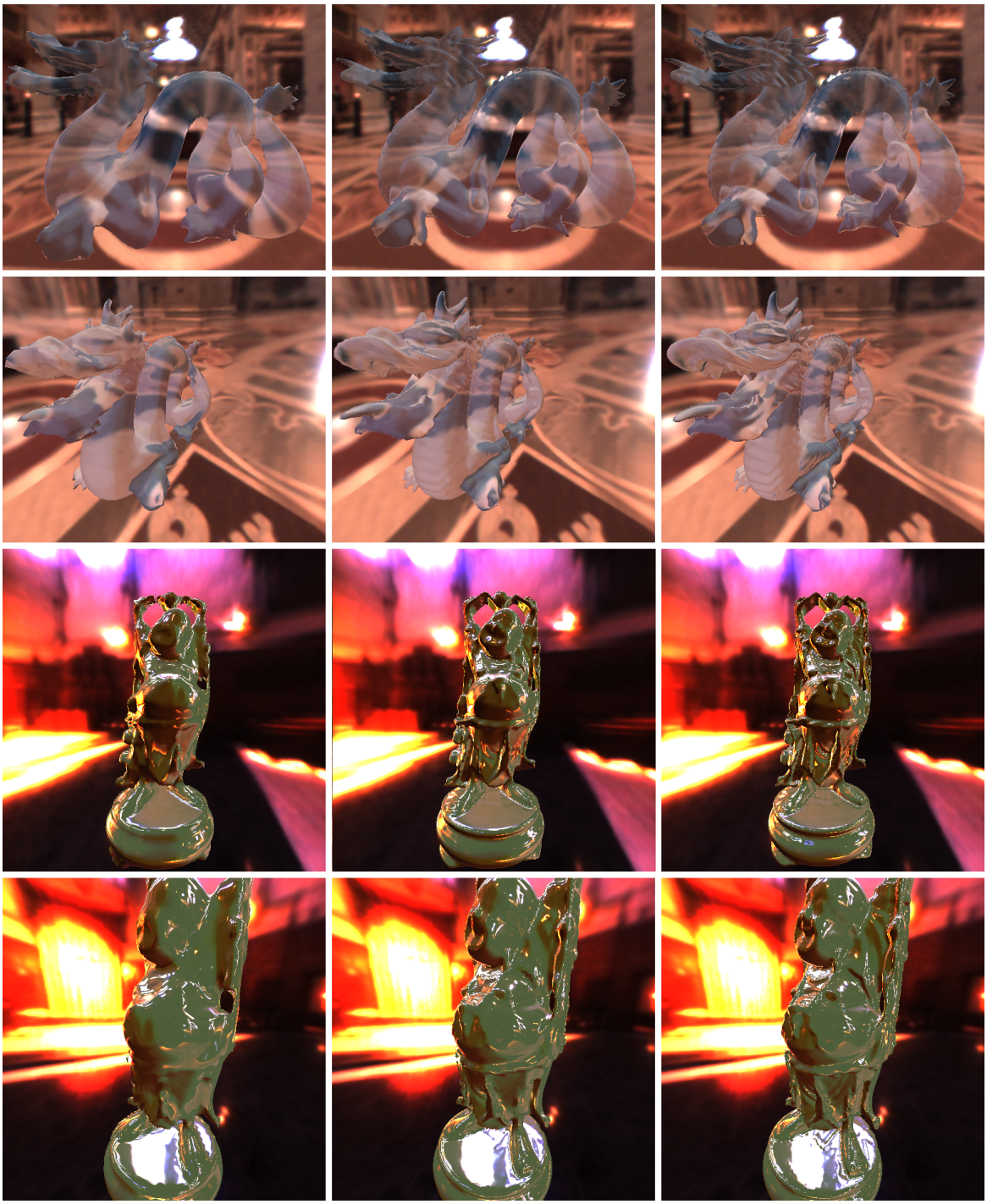
Program DFRRT má velikost bloků nastavenu na 16×16 vláken a rasterizují se buňky hrubé mřížky. Hloubka rekurzivního sledování paprsků je u všech modelů nastavena na hodnotu 1. Generují se tedy pouze primární paprsky. Není-li uvedeno jinak, jsou popsaná nastavení jednotlivých metod použita i pro ostatní měření. Ukázky vygenerovaných obrazů všech tří objektů ve všech třech rozlišeních jsou na obrázcích 5.1 a 5.2.



Obrázek 5.1: Ukázka modelu Bunny v rozlišení $64 \times 64 \times 52$ voxelů (levé dva), v rozlišení $128 \times 128 \times 100$ voxelů (prostřední dva) a v rozlišení $256 \times 256 \times 200$ voxelů při sledování pouze primárních paprsků.

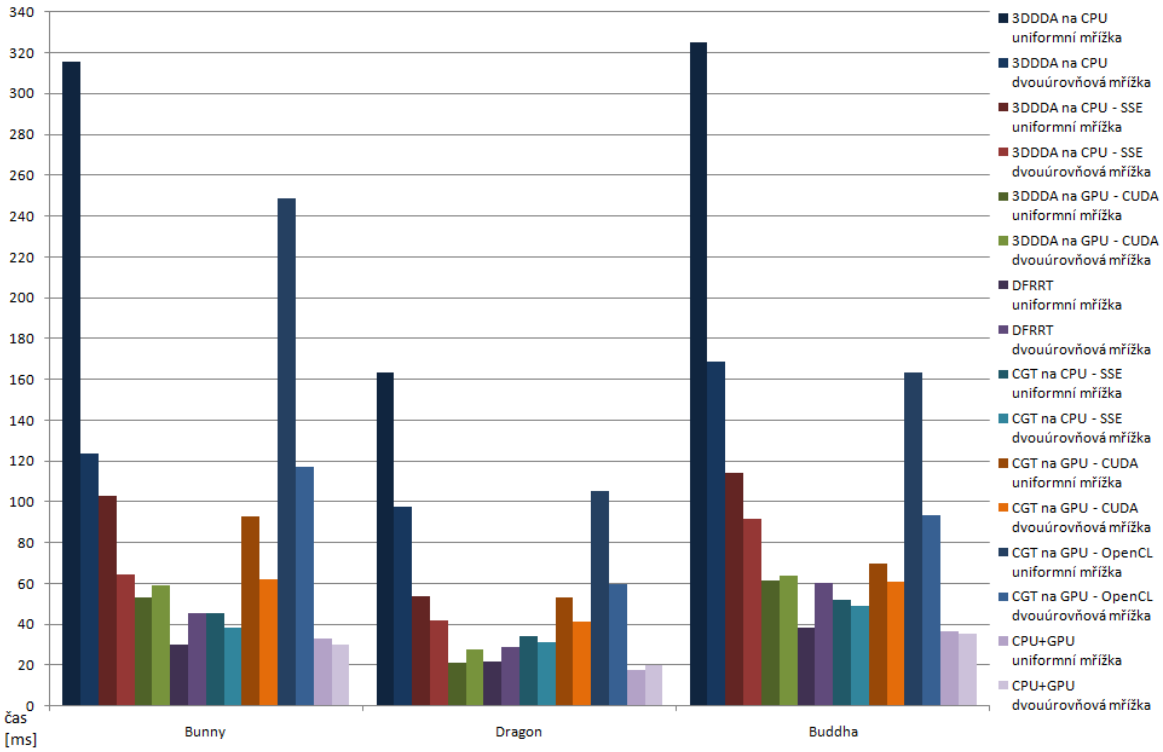
Tabulky 5.5 a 5.6 prezentují časy a relativní rychlosti všech implementovaných metod traverzují-li paprsky dvouúrovňovou mřížku. Graf na obrázku 5.3 poté zobrazuje výsledky všech metod, kromě metody 3DDDA na GPU s použitím OpenCL, jejíž časy jsou enormně vysoké, při traverzaci uniformní i dvouúrovňové mřížky pro všechny modely s rozlišením v hlavní ose 128 voxelů a při rozlišení obrazu 1280×1024 pixelů.

Z tabulek a grafu je zřejmé, že rychlost implementace v jazyku OpenCL je velmi nízká. A to i navzdory optimalizaci kódu a použití nejnovějších verzí překladačů. Nezbývá než doufat, že další verze jazyka a překladačů přinesou zlepšení. Z ostatních metod je nejpomalejší referenční metoda traverzující mřížku algoritmem 3DDDA na CPU bez použití SSE instrukcí. Všechny nové metody, kromě těch využívajících OpenCL, přinášejí oproti referenční metodě



Obrázek 5.2: Ukázka modelu Dragon (horních šest) v rozlišení $64 \times 48 \times 32$ voxelů (levé dva), v rozlišení $128 \times 92 \times 60$ voxelů (prostřední dva) a v rozlišení $256 \times 186 \times 116$ voxelů a modelu Buddha v rozlišení $28 \times 64 \times 28$ voxelů (levé dva), v rozlišení $56 \times 128 \times 56$ voxelů (prostřední dva) a v rozlišení $108 \times 256 \times 108$ voxelů při sledování pouze primárních paprsků

zrychlení. Z metod pracujících na CPU a využívajících SSE instrukce je rychlejší metoda traverzující algoritmem CGT. Algoritmus CGT je ale více závislý na orientaci mřížky. To je patrné zejména u metod využívajících architekturu CUDA. U všech modelů je rychlejší algoritmus 3DDDA, ale u modelu Buddha, který je orientován vertikálně, algoritmus CGT algoritmus 3DDDA téměř dostihuje. Naopak u modelu Dragon, který je orientován horizontálně, je algoritmus 3DDDA téměř dvakrát rychlejší. Metoda omezení intervalu traverzování primárních paprsků, která využívá algoritmus 3DDDA, přináší oproti původní metodě traverzace algoritmem 3DDDA na GPU s použitím architektury CUDA výrazné zrychlení. Je třeba si však uvědomit, že není snadné takovouto metodu efektivně paralelizovat pro více GPU a je časově náročnější aktualizovat při změnách distančního pole polygonální krychle. Nejrychlejší metodou na použitém hardware je metoda využívající CPU a GPU současně.

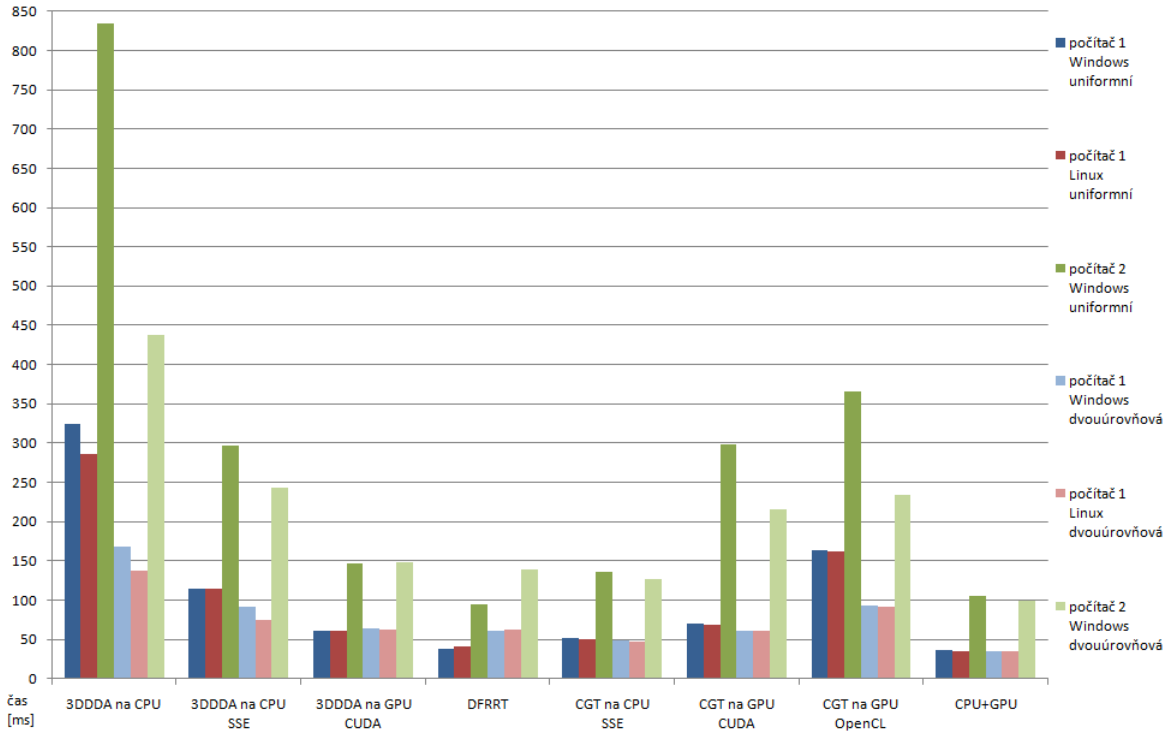


Obrázek 5.3: Porovnání rychlosti všech metod, kromě metody 3DDDA na GPU s OpenCL, pro tři modely s rozlišením v hlavní ose 128 voxelů a při rozlišení obrazu 1280×1024 pixelů.

Graf na obrázku 5.3 dále nabízí porovnání časů traverzace uniformní a dvouúrovňové mřížky. Relativní rychlosti všech metod v procentech pro všechny modely při rozlišení obrazu 1280×1024 pixelů jsou v tabulce 5.7. Obecně platí, že u všech metod je rychlejší traverzace dvouúrovňové mřížky, pouze u metody traverzující algoritmem 3DDDA na GPU využívající architekturu CUDA a tím pádem i u programu DFRRT je rychlejší traverzace uniformní mřížky. Podstatné také je, jak velké zrychlení použití dvouúrovňové mřížky přináší. Zatímco u referenční metody je zrychlení největší, ostatní metody již přinášejí zrychlení menší kvůli sloučení více paprsků dohromady nebo kvůli složitějšímu a delšímu kódu traverzační funkce.

model	rozlišení modelu [voxelů]	3DDDA CPU	SSE	GPU CUDA	GPU OpenCL	GPU DFRRT	CGT CPU SSE	CGT GPU CUDA	CGT GPU OpenCL	CPU+ GPU
Bunny	64x64x52	188	134	119	123	75	108	125	200	118
	128x128x100	255	161	90	141	66	119	150	212	110
	256x256x200	392	208	108	144	66	154	196	300	135
Dragon	64x48x32	139	113	102	98	73	104	113	166	107
	128x92x60	168	127	78	124	75	110	129	176	90
	256x184x116	227	155	97	144	71	127	161	257	111
Buddha	28x64x28	137	102	62	60	56	102	100	103	76
	56x128x56	193	125	97	101	64	107	115	175	104
	108x256x108	264	150	98	123	65	118	137	219	109

Tabulka 5.7: Relativní rychlost generování obrazu v procentech traverzováním dvouúrovňové mřížky oproti traverzování uniformní mřížky při rozlišení obrazu 1280×1024 pixelů v OS Windows na počítači 1.



Obrázek 5.4: Porovnání času generování obrazu v rozlišení 1280×1024 pixelů modelu Buddha v rozlišení $56 \times 128 \times 56$ voxelů na dvou počítačových sestavách, dvou operačních systémech a pro traverzaci uniformní a dvouúrovňové mřížky.

Tabulky 5.8 a 5.9 ukazují zrychlení programu v procentech v operačním systému Linux. Rozlišení obrazu je 1280 × 1024 pixelů. Zrychlení dosahují zejména metody pracující na CPU díky tomu, že si překladač gcc lépe poradí se šablonami funkcí a instrukcemi SSE než překladač od společnosti Microsoft.

model	rozlišení modelu [voxelů]	3DDDA				CGT			CPU+ GPU
		CPU		GPU		CPU	GPU		
			SSE	CUDA	DFRRT	SSE	CUDA	OpenCL	
Bunny	64x64x52	125	102	102	111	116	102	101	111
	128x128x100	116	98	101	107	113	101	101	108
	256x256x200	110	95	101	101	108	100	100	102
Dragon	64x48x32	148	110	103	94	122	103	102	109
	128x92x60	135	105	104	113	117	101	101	106
	256x184x116	123	100	102	99	111	101	100	100
Buddha	28x64x28	120	106	103	91	106	102	103	106
	56x128x56	113	99	101	94	105	101	101	105
	108x256x108	108	93	101	101	95	101	100	104

Tabulka 5.8: Relativní rychlost generování obrazu v procentech traverzací uniformní mřížky v OS Linux oproti generování obrazu v OS Windows při rozlišení obrazu 1280 × 1024 pixelů na počítači 1.

model	rozlišení modelu [voxelů]	3DDDA					CGT			CPU+ GPU
		CPU		GPU			CPU	GPU		
			SSE	CUDA	OpenCL	DFRRT	SSE	CUDA	OpenCL	
Bunny	64x64x52	148	125	102	105	105	117	102	102	109
	128x128x100	135	125	102	106	101	113	101	101	112
	256x256x200	117	125	101	107	99	105	101	101	102
Dragon	64x48x32	175	120	104	104	98	122	103	104	114
	128x92x60	167	127	103	105	92	118	102	102	99
	256x184x116	154	127	102	107	101	110	101	101	97
Buddha	28x64x28	128	120	102	104	94	106	102	102	107
	56x128x56	123	122	101	105	98	105	101	101	102
	108x256x108	114	123	101	106	97	102	101	101	105

Tabulka 5.9: Relativní rychlost generování obrazu v procentech traverzací dvouúrovňové mřížky v OS Linux oproti generování obrazu v OS Windows při rozlišení obrazu 1280 × 1024 pixelů na počítači 1.

Tabulky 5.10 a 5.11 porovnávají rychlost generování obrazu s rozlišením 1280 × 1024 pixelů na počítači 1 a počítači 2. Graf na obrázku 5.4 také porovnává rychlost jednotlivých metod na různých počítačích při nejvyšším rozlišení obrazu na modelu Buddha v rozlišení 56 × 128 × 56 voxelů a dále také rychlost v různých operačních systémech na počítači 1 i rychlost traverzování uniformní a dvouúrovňové mřížky. Zatímco metody pracující na CPU

jsou na počítači 1 vždy vykonávány 2,5 až 2,7 krát rychleji než na počítači 2, rychlost metod využívajících GPU se pohybuje v rozmezí pouze dvojnásobného zrychlení až zrychlení 4,3.

model	rozlišení modelu [voxelů]	3DDDA				CGT			CPU+ GPU
		CPU		GPU		CPU	GPU		
			SSE	CUDA	DFRRT	SSE	CUDA	OpenCL	
Bunny	64x64x52	37	38	44	42	38	25	49	36
	128x128x100	39	39	35	35	39	25	41	32
	256x256x200	39	38	33	41	39	24	39	30
Dragon	64x48x32	38	39	43	37	39	28	47	34
	128x92x60	39	39	31	36	39	29	36	30
	256x184x116	39	39	32	36	39	30	36	31
Buddha	28x64x28	39	38	34	33	38	24	33	32
	56x128x56	39	38	42	41	38	24	45	35
	108x256x108	39	39	36	41	38	23	40	32

Tabulka 5.10: Relativní rychlost generování obrazu v procentech při traverzaci uniformní mřížky na počítači 2 oproti generování obrazu na počítači 1.

model	rozlišení modelu [voxelů]	3DDDA				CGT			CPU+ GPU
		CPU		GPU		CPU	GPU		
			SSE	CUDA	DFRRT	SSE	CUDA	OpenCL	
Bunny	64x64x52	37	38	40	42	37	30	40	33
	128x128x100	39	39	46	43	38	31	45	36
	256x256x200	38	37	44	45	38	33	45	34
Dragon	64x48x32	37	39	37	39	39	32	40	32
	128x92x60	39	39	36	35	38	35	41	34
	256x184x116	39	36	37	37	39	38	44	33
Buddha	28x64x28	38	39	41	40	39	27	36	36
	56x128x56	39	38	43	44	39	28	40	36
	108x256x108	38	39	42	43	38	29	40	35

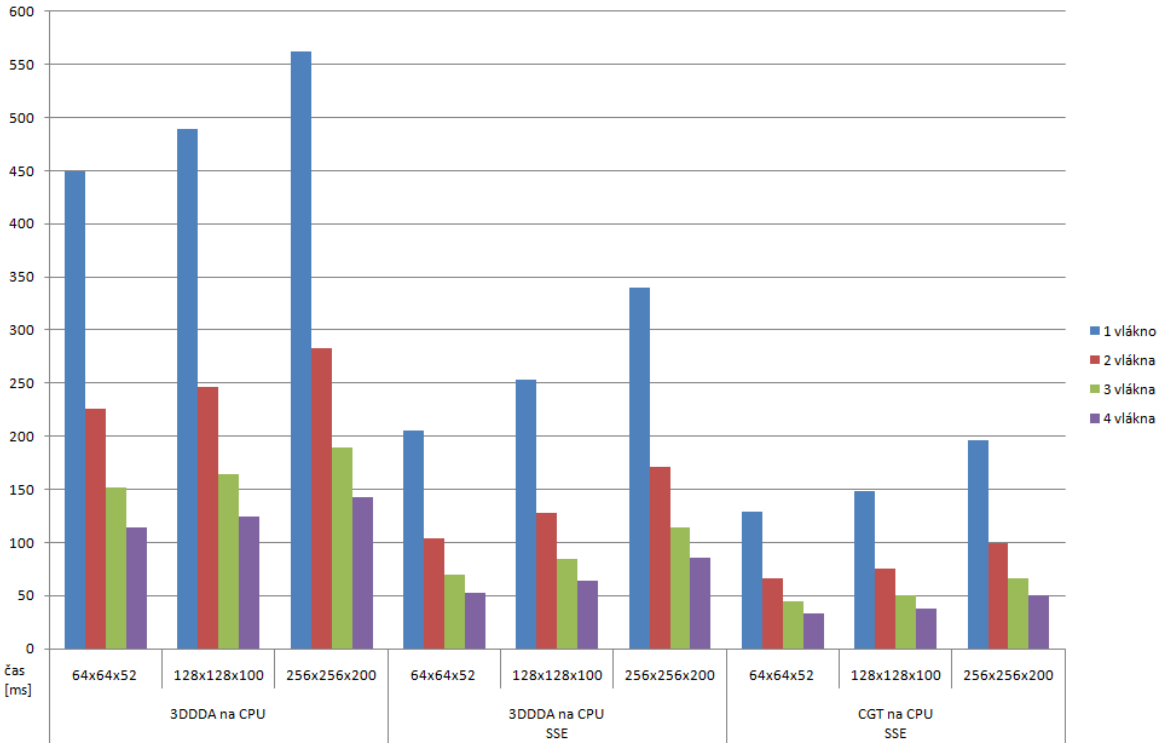
Tabulka 5.11: Relativní rychlost generování obrazu v procentech při traverzaci dvouúrovňové mřížky na počítači 2 oproti generování obrazu na počítači 1.

5.3 Vliv počtu programových vláken na čas výpočtu na CPU

Tabulka 5.12 a graf na obrázku 5.5 ukazují vliv počtu programových vláken na čas výpočtu. Měření bylo provedeno pro model Bunny v různých rozlišeních s velikostí obrazu 1280×1024 pixelů a traverzací dvouúrovňové mřížky. Tabulka kromě absolutních časů udává i relativní rychlost generování v procentech při použití více vláken oproti použití jediného vlákna. Z těchto údajů je zřejmé, že škálovatelnost programu je téměř ideální.

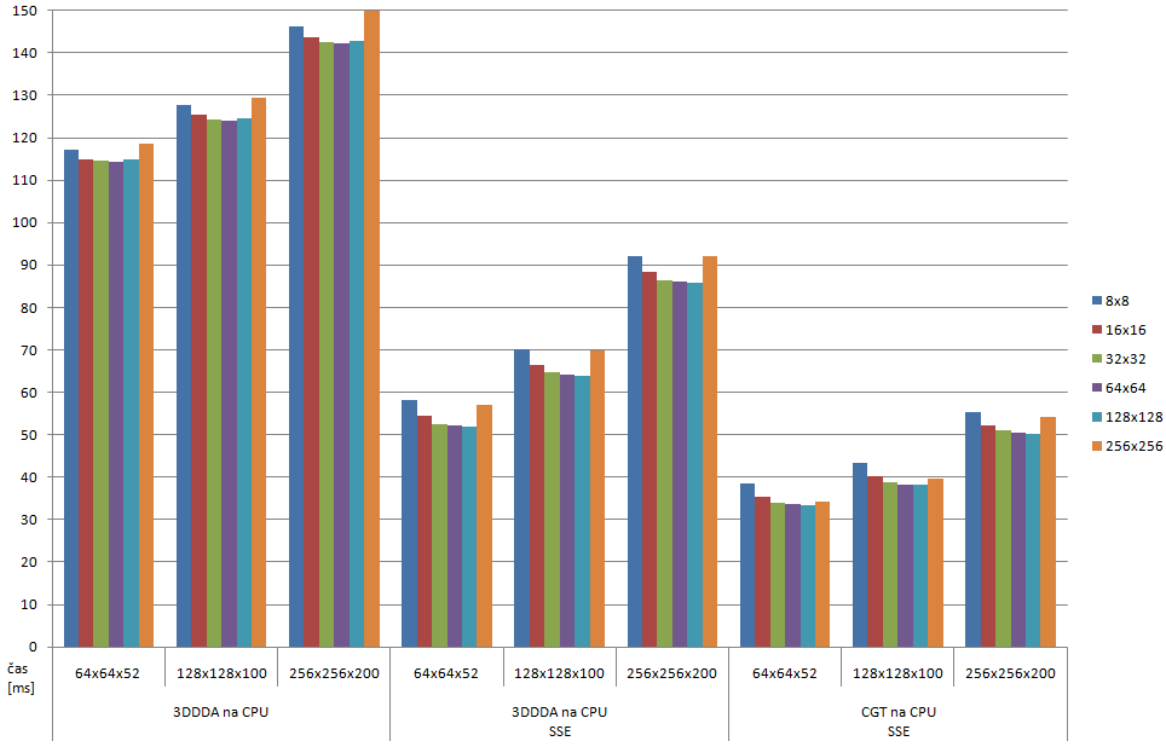
metoda	rozlišení modelu [voxelů]	1 vlákno čas [ms]	2 vlákna čas [ms]	2 vlákna rel. [%]	3 vlákna čas [ms]	3 vlákna rel. [%]	4 vlákna čas [ms]	4 vlákna rel. [%]
3DDDA	64x64x52	449	226	198	151	297	114	393
na CPU	128x128x100	489	246	198	165	297	124	394
bez SSE	256x256x200	562	283	198	189	297	142	395
3DDDA	64x64x52	205	104	198	69	297	52	392
na CPU	128x128x100	253	127	198	85	297	64	394
s SSE	256x256x200	340	171	198	114	298	86	396
CGT	64x64x52	130	66	196	44	292	34	385
na CPU	128x128x100	148	75	197	50	293	38	388
s SSE	256x256x200	196	99	198	67	295	50	391

Tabulka 5.12: Vliv počtu vláken na čas generování obrazu a relativní výkon.



Obrázek 5.5: Škálovatelnost metod pracujících na CPU při rostoucím počtu programových vláken.

5.4 Vliv velikosti obrazových dlaždic na čas výpočtu na CPU



Obrázek 5.6: Závislost času generování obrazu na velikost obrazových dlaždic pro model Bunny v různých rozlišeních při velikosti obrazu 1280×1024 pixelů.

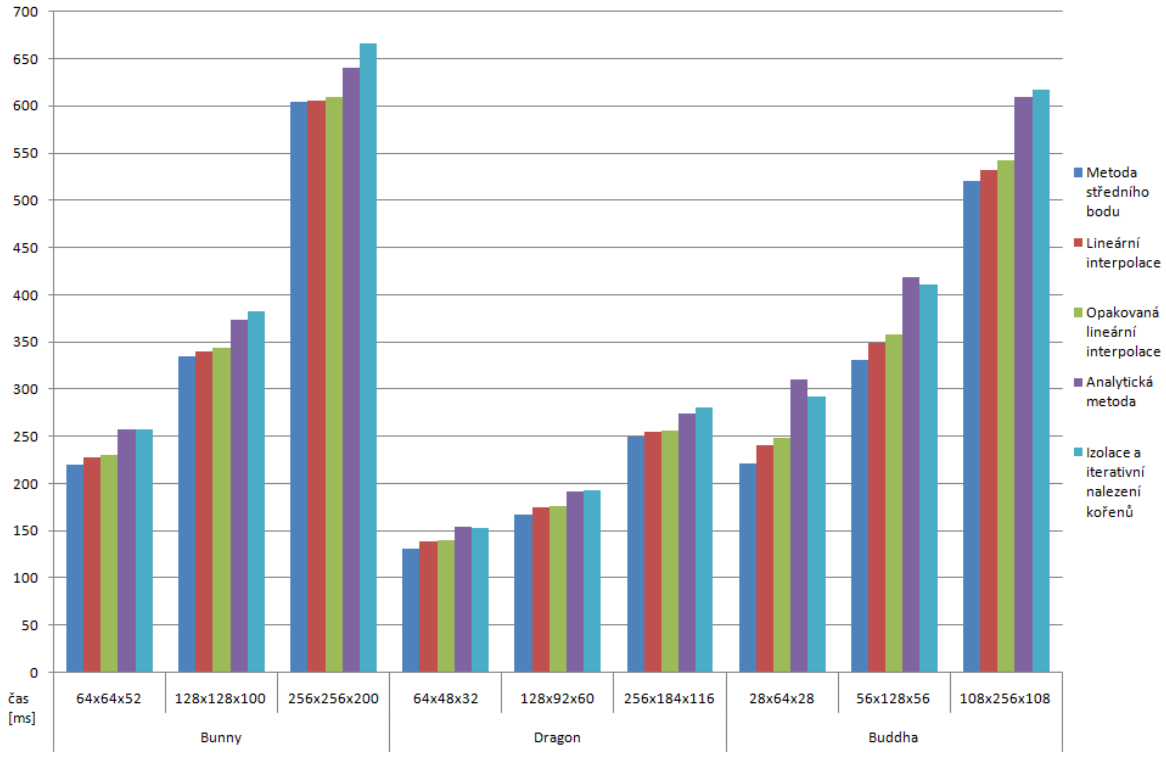
Paralelizace generování obrazu na CPU je provedena na úrovni obrazových dlaždic. Z tabulky 5.13 a grafu na obrázku 5.6 vyplývá, že při rozlišení obrazu 1280×1024 pixelů je nejvhodnější volit velikost čtvercových dlaždic 64×64 nebo 128×128 pixelů. Měření bylo provedeno na modelu Bunny v různých rozlišeních pomocí 4 programových vláken a při traverzování dvouúrovňové mřížky. Vliv velikosti dlaždic je ale velmi malý. Pouze pro metody pracující s pakety paprsků není vhodné volit jejich velikost menší, než je velikost paketu. Také není vhodné volit velikost příliš velkou, neboť rozměry dlaždic nemusí bezezbytku dělit rozměry obrazu, dlaždice mají různé velikosti a škálovatelnost klesá.

5.5 Čas generování obrazu různých metod výpočtu průsečíku

Pro metodu traverzující mřížku algoritmem 3DDDA na CPU bez použití SSE byly implementovány všechny metody výpočtu průsečíku paprsku s povrchem objektu popsané v kapitole 3.4. Celkové časy generování obrazu o rozlišení 1280×1024 pixelů pomocí těchto metod při traverzaci uniformní mřížky jsou uvedeny v tabulce 5.14. Tabulka také udává relativní rychlost v procentech oproti metodě středního bodu, která je nejrychlejším možným řešením daného úkolu. Zjištěné časy jednotlivých metod ukazuje i graf na obrázku 5.7. Vliv použité

metoda	rozlišení modelu [voxelů]	čas [ms] při velikosti dlaždic								
		1x1	2x2	4x4	8x8	16x16	32x32	64x64	128x128	256x256
3DDDA	64x64x52	206	145	124	117	115	114	114	115	119
na CPU	128x128x100	217	156	134	128	126	124	124	125	130
bez SSE	256x256x200	236	174	153	146	144	143	142	143	150
3DDDA	64x64x52	337	87	66	58	54	53	52	52	57
na CPU	128x128x100	381	99	78	70	66	65	64	64	70
s SSE	256x256x200	468	121	100	92	88	87	86	86	92
CGT	64x64x52	2385	598	150	39	35	34	34	34	34
na CPU	128x128x100	2648	665	168	43	40	39	38	38	40
s SSE	256x256x200	3374	846	214	55	52	51	50	50	54

Tabulka 5.13: Vliv velikosti obrazových dlaždic na čas generování obrazu pomocí 4 vláken.



Obrázek 5.7: Závislost času generování obrazu na zvolené metodě výpočtu průsečíku paprsku s povrchem při velikosti obrazu 1280×1024 pixelů.

metody na celkový čas je pouze minimální, přesto je z grafu patrné, že analytická metoda i metoda izolace a iterativního nalezení kořenů celkový čas výrazně prodlužují a hodí se tak spíše pro aplikace, kde je přesnost zobrazení preferována před rychlostí. Metoda opakované lineární interpolace je pouze nepatrně pomalejší než metoda prosté lineární interpolace. Obě metody naleznou průsečík ve stejných případech, opakovanou interpolací je bod průsečíku pouze nalezen přesněji. Jelikož výsledná kvalita obrazu závisí více na směru normálového vektoru než na přesné poloze bodu průsečíku, je pro ostatní metody generování obrazu implementována pouze metoda lineární interpolace.

model	rozlišení modelu [voxelů]	Střední bod čas [ms]	Lineární interp. čas [ms]	Lineární interp. rel. [%]	Opak. lin. interp. čas [ms]	Opak. lin. interp. rel. [%]	Analytická čas [ms]	Analytická rel. [%]	Izolace čas [ms]	Izolace rel. [%]
Bunny	64x64x52	220	228	96	230	95	257	86	257	85
	128x128x100	335	340	98	344	97	373	90	382	88
	256x256x200	604	606	100	610	99	640	94	666	91
Dragon	64x48x32	131	139	95	139	94	154	85	153	86
	128x92x60	167	175	96	176	95	192	87	193	87
	256x184x116	249	255	98	257	97	274	91	281	89
Buddha	28x64x28	221	241	92	249	89	310	71	292	76
	56x128x56	331	349	95	358	93	418	79	411	81
	108x256x108	520	532	98	543	96	609	85	617	84

Tabulka 5.14: Porovnání časů generování obrazu při použití různých metod nalezení průsečíku paprsků s povrchem a relativní rychlost metod oproti metodě středního bodu.

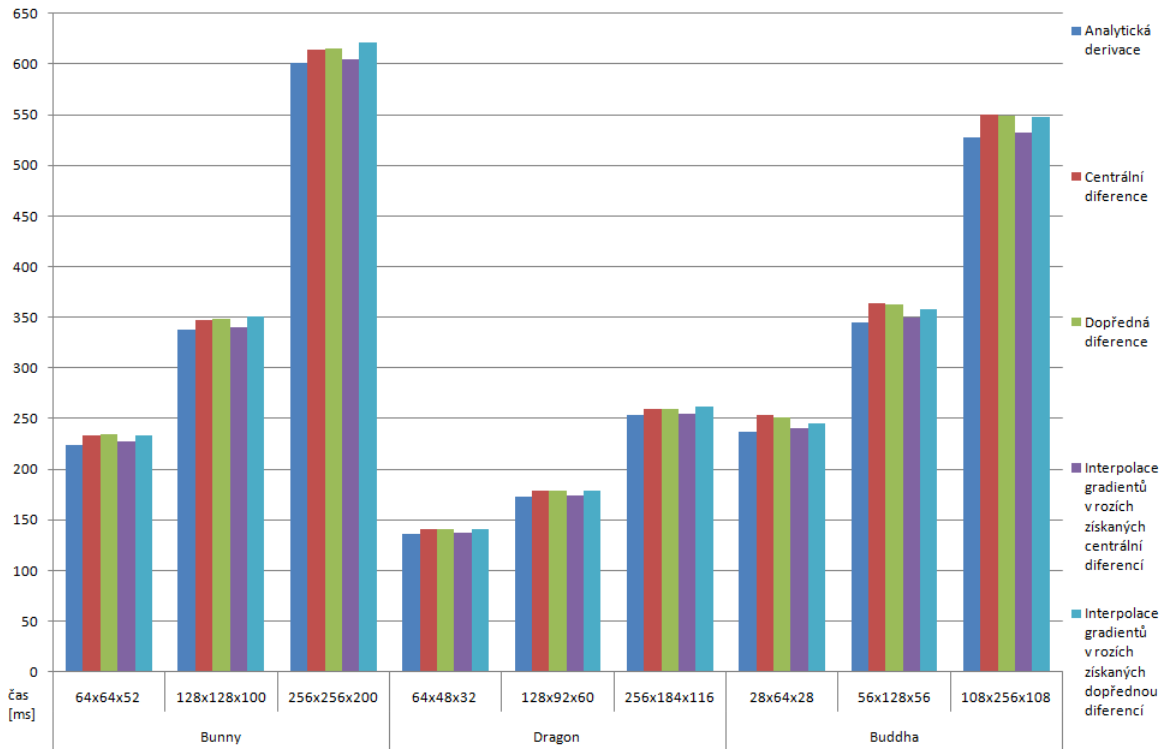
5.6 Čas generování obrazu různých metod výpočtu gradientu

U metody traverzace algoritmem 3DDDA na CPU bez použití SSE je vliv různých metod výpočtu gradientu na čas generování obrazu pouze minimální. Pro měření bylo použito traverzování uniformní mřížky a velikost obrazu byla 1280×1024 pixelů. Tabulka 5.15 a graf na obrázku 5.8 poté ukazují, že nejrychlejší je metoda analytické derivace interpolační funkce. Kvalita obrazu je pro ní ale nedostatečná. Druhou nejrychlejší a zároveň spolu s centrální diferencí nejkvalitnější metodou je interpolace gradientů v rozích získaných centrální diferencí. Metoda je při výpočtu na CPU rychlá, neboť je nutné počítat pouze jednu interpolaci.

Tabulka 5.16 a graf na obrázku 5.9 ukazují, že při implementaci na GPU pomocí architektury CUDA je díky uložení distančního pole do 3D textury rychlejší metoda centrální diference a rozdíl oproti použití interpolace gradientů v rozích počítaných centrální diferencí z hodnot distančního pole uloženého v hlavní paměti grafické karty je velmi výrazný. Při implementaci algoritmu 3DDDA pomocí jazyka OpenCL je situace zcela opačná. Překladač pravděpodobně nedokáže program přizpůsobit k plnému využití možností hardware a čtení interpolovaných dat z textury je tak výrazně pomalejší než čtení dat z pole hodnot a počítání interpolace softwarově.

model	rozlišení modelu [voxelů]	Analytická derivace čas [ms]	Centrální diference čas [ms]	Centrální diference rel. [%]	Dopředná diference čas [ms]	Dopředná diference rel. [%]	Interpol. grad. získaných centrál. dif. čas [ms]	Interpol. grad. získaných centrál. dif. rel. [%]	Interpol. grad. získaných dopřed. dif. čas [ms]	Interpol. grad. získaných dopřed. dif. rel. [%]
Bunny	64x64x52	224	233	96	235	96	227	99	233	96
	128x128x100	337	347	97	349	97	340	99	351	96
	256x256x200	601	614	98	615	98	605	99	621	97
Dragon	64x48x32	137	141	97	141	97	137	99	140	97
	128x92x60	173	178	97	178	97	174	99	178	97
	256x184x116	253	260	98	260	98	255	99	262	97
Buddha	28x64x28	237	253	93	252	94	240	98	245	97
	56x128x56	345	364	95	363	95	349	99	358	96
	108x256x108	527	550	96	549	96	532	99	548	96

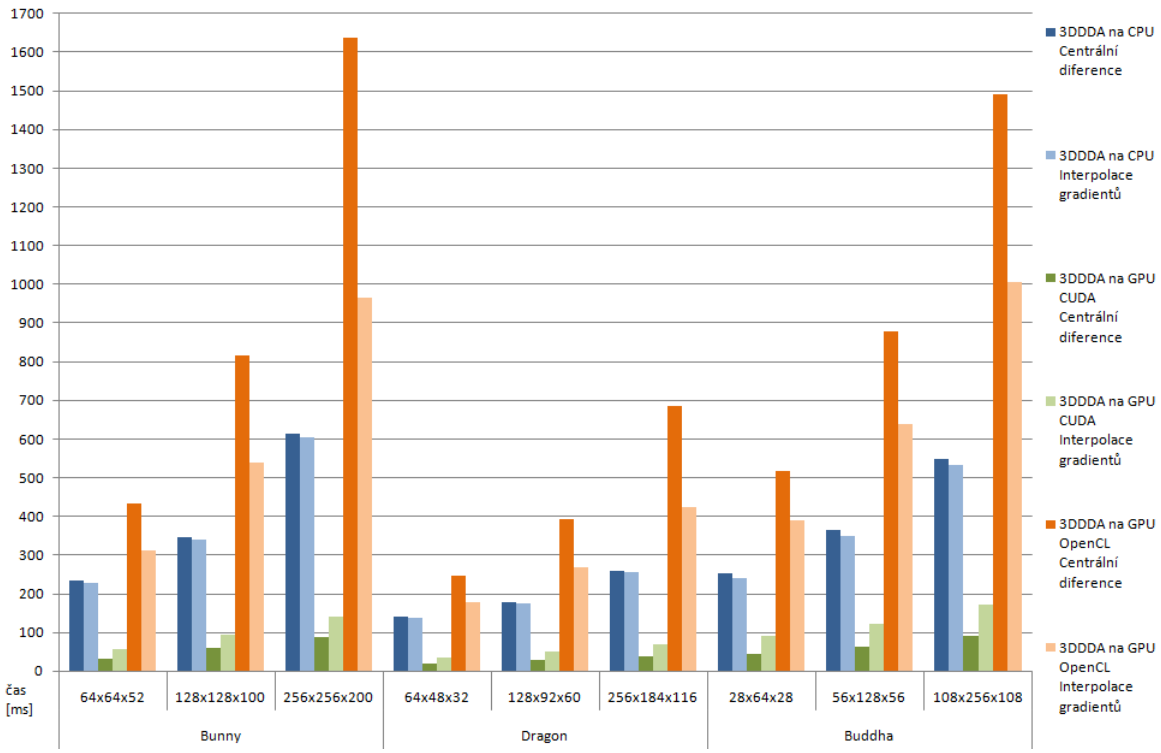
Tabulka 5.15: Porovnání časů generování obrazu při použití různých metod výpočtu gradientu a relativní rychlost metod oproti metodě analytické derivace interpolační funkce.



Obrázek 5.8: Závislost času generování obrazu na zvolené metodě výpočtu gradientu při velikosti obrazu 1280×1024 pixelů.

model	rozlišení modelu [voxelů]	CPU bez SSE		GPU - CUDA		GPU - OpenCL	
		Cent. dif. [ms]	Interp. [ms]	Cent. dif. [ms]	Interp. [ms]	Cent. dif. [ms]	Interp. [ms]
Bunny	64x64x52	233	227	34	58	434	311
	128x128x100	347	340	59	96	814	539
	256x256x200	614	605	87	141	1638	967
Dragon	64x48x32	141	137	21	36	245	178
	128x92x60	178	174	27	49	393	268
	256x184x116	260	255	40	68	685	423
Buddha	28x64x28	253	240	45	92	519	391
	56x128x56	364	349	64	123	879	640
	108x256x108	550	532	92	172	1491	1005

Tabulka 5.16: Porovnání časů generování obrazu u metod traverzujících algoritmem 3DDDA při použití metody výpočtu gradientu centrální diferencí a metody interpolace gradientu v rozích získaných centrální diferencí.



Obrázek 5.9: Čas generování obrazu v rozlišení 1280×1024 pixelů při použití metod výpočtu gradientu centrální diferencí a interpolací gradientů v rozích vypočítaných centrální diferencí.

5.7 Vliv rozdělování paketů na čas

Pro metodu traverzující mřížku algoritmem 3DDDA na CPU s použitím SSE instrukcí byla implementována možnost rozdělit paket o velikosti 2×2 na jednotlivé paprsky, jestliže některé paprsky paketu nemají pokračovat v traverzování. Rozdělení paketu je možné provést při zvoleném počtu zbývajících paprsků, pokud některé paprsky nalezly povrch nebo dokončily traverzování mřížky (situace A), pokud některé paprsky neprotínají obklopující kvádr mřížky (situace B) nebo pokud pro některé paprsky nebyly vygenerovány sekundární paprsky (situace C). Dělení paketu v popsaných situacích je možno libovolně kombinovat. Tabulka 5.17 obsahuje časy generování obrazů o velikosti 1280×1024 pixelů dvou modelů při traverzaci dvouúrovňové mřížky. Zejména byl měřen vliv nastavení různého počtu zbývajících paprsků a také čas při dělení paketu pouze v jedné ze situací. Bylo zjištěno, že dělení paketu zrychlení nepřináší. Naopak může být výsledný čas vyšší kvůli různým testům, zda dělení provádět a kvůli samotnému dělení.

model	rozlišení modelu [voxelů]	bez dělení [ms]	dělení ve všech sit. zbývá-li 1 paprsek [ms]	2 paprsky [ms]	3 paprsky [ms]	zbývají-li 2 paprsky v sit. A [ms]	v sit. B [ms]	v sit. C [ms]
Bunny	64x64x52	53	54	56	58	57	53	53
	128x128x100	64	66	70	72	70	65	65
	256x256x200	86	89	94	97	94	86	86
Dragon	64x48x32	57	59	63	65	62	57	57
	128x92x60	70	73	78	82	78	70	70
	256x184x116	88	93	98	102	98	88	87

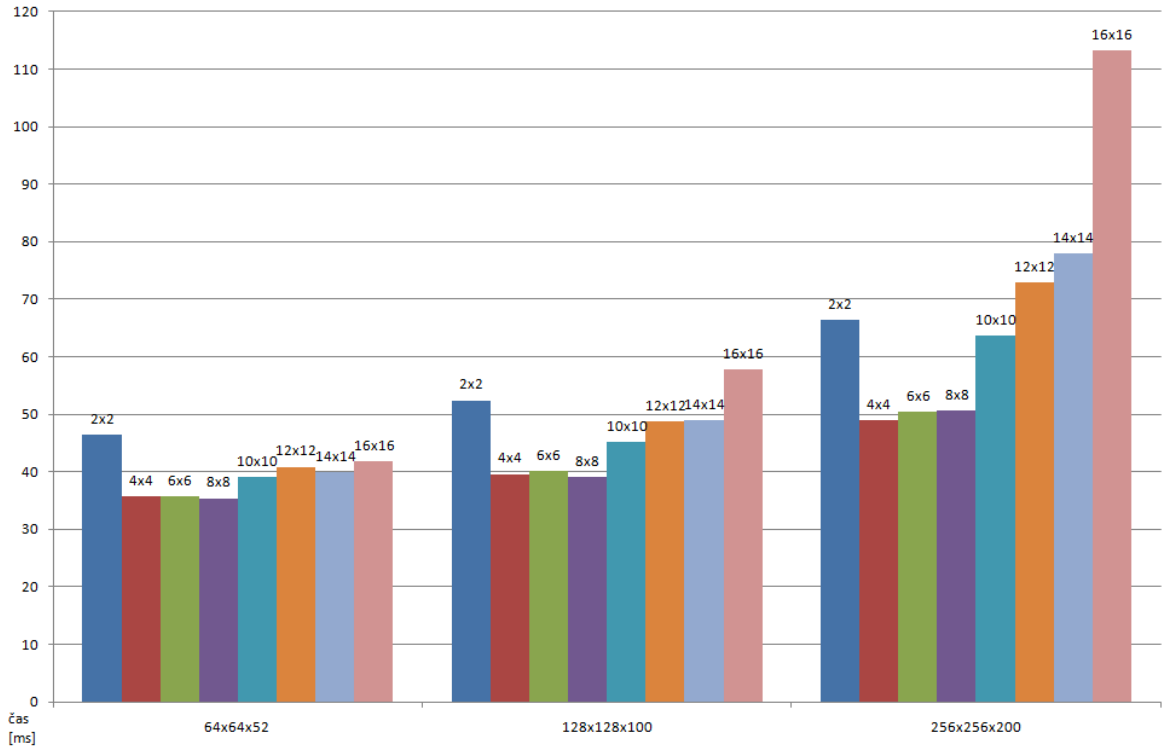
Tabulka 5.17: Testování vlivu rozdělení paketu na jednotlivé paprsky u metody traverzace algoritmem 3DDDA na CPU s použitím SSE instrukcí při různém počtu paprsků a v různých situacích. Situace A – některé paprsky paketu zasáhly povrch nebo dokončily traverzování mřížky. Situace B – některé paprsky paketu nezasáhly obklopující kvádr mřížky. Situace C – pro některé paprsky nebyly vygenerovány sekundární paprsky.

5.8 Závislost času na velikosti paketu algoritmu CGT

Metoda CGT na CPU využívající SSE instrukce dovoluje nastavit různé velikosti paketů paprsků. Jediným omezením je, že se výsledné pakety musí skládat ze sub-paketů o velikosti 2×2 paprsky. Závislost času generování obrazu velikosti 1280×1024 pixelů modelu Bunny v různých rozlišeních při traverzaci dvouúrovňové mřížky na velikosti paketů je v tabulce 5.18 a v grafu na obrázku 5.10. Z nich vyplývá, že vhodná velikost závisí na rozlišení modelu. Dále také závisí na nastavení pohledu na model. Pokud je model dále od kamery, pokrývají pakety velkou část mřížky a čas by byl nižší, pokud by naopak ostatní pakety nemíjely mřížku úplně. Při větším přiblížení modelu protne paket menší množství buněk a čas generování obrazu se při přibližování modelu může snižovat. Z grafu je patrné, že vhodná velikost paketu je 4×4 nebo 8×8 paprsků.

rozlišení modelu [voxelů]	velikost paketu							
	2x2	4x4	6x6	8x8	10x10	12x12	14x14	16x16
64x64x52	46	36	36	35	39	41	40	42
128x128x100	52	40	40	39	45	49	49	58
256x256x200	66	49	50	51	64	73	78	113

Tabulka 5.18: Vliv velikosti paketu paprsků u metody traverzace algoritmem CGT na CPU na čas generování obrazu v milisekundách.



Obrázek 5.10: Vliv velikosti paketu paprsků na čas generování obrazu algoritmem CGT.

5.9 Vliv ořezu buněk jemné mřížky na čas výpočtu

Pro metodu traverzující algoritmem CGT na CPU byl testován vliv ořezu buněk jemné mřížky, pro které při přechodu z traverzování hrubé mřížky na traverzování jemné mřížky buňky hrubé mřížky neindikují přítomnost povrchu. Čas byl měřen pro všechny modely s rozlišením obrazu 1280×1024 pixelů. Tabulka 5.19 prezentuje výsledné časy s ořezem buněk i bez ořezu. Bylo zjištěno, že ořez nemá vliv na rychlost a je tedy třeba ho implementovat pouze pokud nejsou definovány buňky jemné mřížky v místech, kde hrubá mřížka neindikuje přítomnost povrchu.

model	rozlišení modelu [voxelů]	bez ořezu čas [ms]	s ořezem čas [ms]
Bunny	64x64x52	34	34
	128x128x100	39	39
	256x256x200	51	51
Dragon	64x48x32	29	29
	128x92x60	32	32
	256x184x116	39	40
Buddha	28x64x28	45	46
	56x128x56	49	49
	108x256x108	58	58

Tabulka 5.19: Vliv ořezu buněk jemné mřížky při přechodu z traverzování hrubé mřížky u algoritmu CGT na CPU.

5.10 Počty sledovaných paprsků v různé hloubce rekurze a vliv na čas výpočtu

model	rozlišení modelu [voxelů]	počet paprsků v hloubce					
		1	2	3	4	5	6
Bunny	64x64x52	1 310 720	127 267	10 185	695	16	0
	128x128x100	1 310 720	148 021	13 381	1 020	23	0
	256x256x200	1 310 720	170 107	15 965	1 243	29	1
Dragon	64x48x32	1 310 720	71 314	67 302	10 422	9 803	1 854
	128x92x60	1 310 720	83 276	79 343	14 250	13 451	2 547
	256x184x116	1 310 720	95 321	91 012	17 028	16 022	3 025
Buddha	28x64x28	1 310 720	300 304	48 452	21 782	10 861	5 780
	56x128x56	1 310 720	330 084	61 312	23 820	10 970	5 385
	108x256x108	1 310 720	368 994	70 722	26 028	11 789	5 955

Tabulka 5.20: Přehled počtu sledovaných paprsků v různé hloubce stromu paprsků. Pro model Bunny a Buddha generovány pouze odražené paprsky, pro model Dragon pouze lomené.

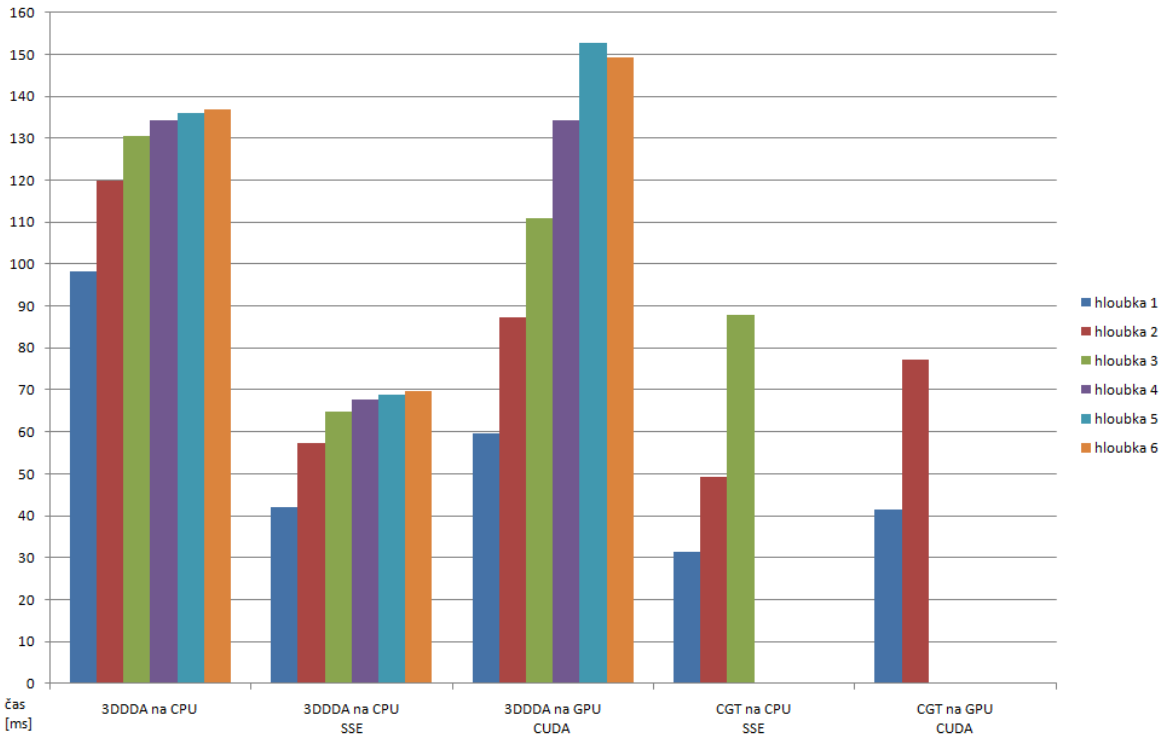
Tabulka 5.20 poskytuje přehled počtu primárních paprsků pro vygenerování obrazu všech modelů o rozlišení 1280×1024 pixelů, průměrný počet sekundárních paprsků v hloubce 2, které vznikají, pokud primární paprsek zasáhne povrch objektu, průměrný počet sekundárních paprsků v hloubce 3, které vznikají, pokud sekundární paprsek v hloubce 2 zasáhne povrch atd. Pro každý paprsek, který zasáhne povrch, je generován pouze jeden další paprsek. U modelu Bunny a Buddha se jedná o odražený paprsek a u modelu Dragon o lomený. Proto počet paprsků postupně klesá a časy generování obrazu získané traverzací dvouúrovňové mřížky uvedené v tabulce 5.21 rostou při zvyšování hloubky rekurze stále méně.

metoda	hloubka rekurze	Bunny 64 x 64 x 52	Bunny 128 x 128 x 100	Bunny 256 x 256 x 200	Dragon 64 x 48 x 32	Dragon 128 x 92 x 60	Dragon 256 x 184 x 116	Buddha 28 x 64 x 28	Buddha 56 x 128 x 56	Buddha 108 x 256 x 108
3DDDA na CPU bez SSE	1	115	125	143	95	98	104	165	170	188
	2	130	143	165	112	120	131	206	217	244
	3	132	145	167	121	131	143	215	230	259
	4	132	145	168	123	134	147	220	235	265
	5	132	145	168	125	136	149	222	238	268
	6	132	145	167	125	137	150	223	239	269
3DDDA na CPU s SSE	1	52	64	86	37	42	50	80	91	113
	2	62	77	103	48	57	71	104	122	153
	3	63	78	105	53	65	80	111	132	166
	4	63	78	106	55	68	84	115	137	173
	5	63	79	106	56	69	86	117	140	176
	6	63	78	106	56	70	87	118	141	177
3DDDA na GPU CUDA	1	70	115	175	44	60	85	103	143	205
	2	98	153	221	59	87	126	153	220	317
	3	105	163	234	74	111	158	173	254	371
	4	122	194	290	88	134	195	203	305	453
	5	134	212	316	98	153	220	231	348	517
	6	122	194	294	94	149	218	217	334	504
CGT na CPU s SSE	1	34	38	50	29	31	39	45	49	57
	2	89	314	3071	38	49	86	113	464	2430
	3	138	631	6446	50	88	217	215	1107	6076
CGT na GPU CUDA	1	36	64	143	24	41	91	42	64	113
	2	75	119	213	45	77	145	101	155	236

Tabulka 5.21: Vliv nastavené hloubky rekurzivního sledování jednoho druhu paprsků na čas generování obrazu v milisekundách. Pro model Bunny a Buddha použity pouze odražené paprsky, pro model Dragon pouze lomené.

Výjimku tvoří metody používající algoritmus CGT, neboť sekundární paprsky nejsou tolik koherentní jako primární paprsky a je třeba traverzovat větší část mřížky, kterou nekoherentní pakety pokrývají. U algoritmu 3DDDA na GPU s architekturou CUDA rostou časy při zvyšování hloubky rekurze více než u algoritmů 3DDDA pracujících na CPU. Z tabulky je přesto patrné, že je u této metody rychlejší generovat obraz s hloubkou rekurze 6 než s hloubkou 5. Důvodem je, že hloubka 6 je maximální hloubka, pro kterou byl program přeložen a překladač kód pro tuto hloubku optimalizoval. Časy generování obrazu modelu Dragon s rozlišením $128 \times 92 \times 60$ voxelů různými metodami znázorňuje i graf na obrázku 5.11.

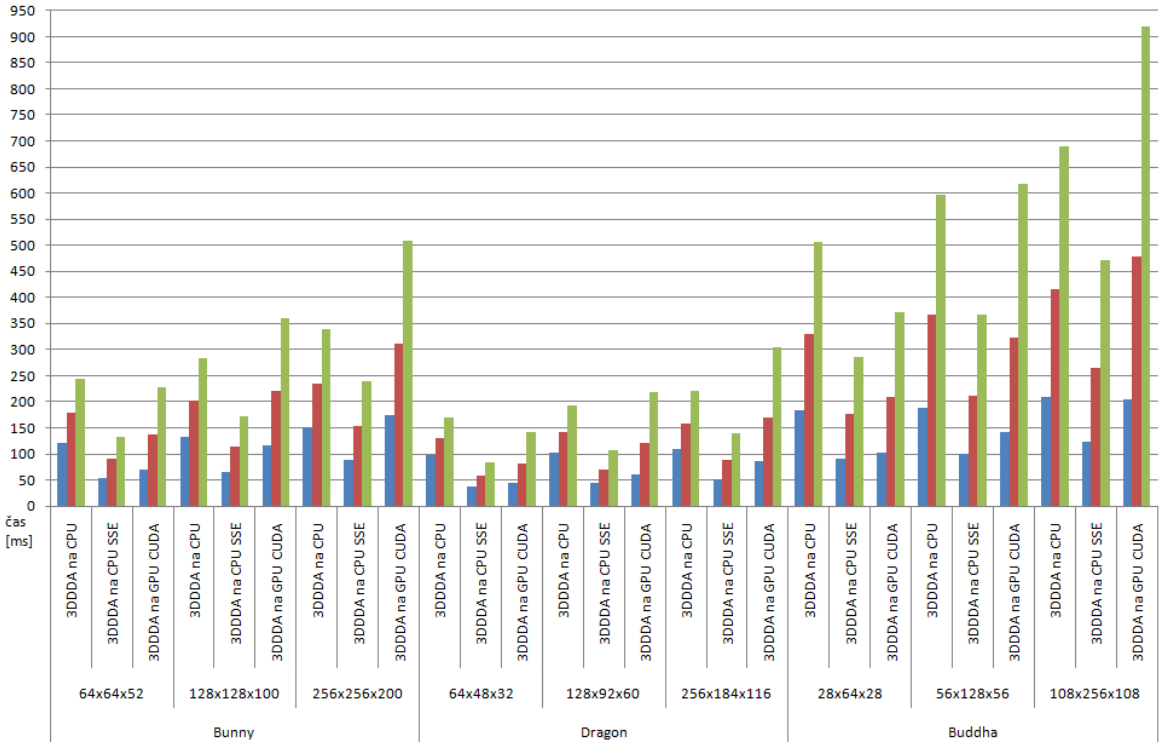
Tabulka 5.22 a graf na obrázku 5.12 ukazují vliv nastavené hloubky rekurzivního sledování paprsků, jsou-li generovány odražené i lomené paprsky současně. Měření bylo provedeno



Obrázek 5.11: Závislost času generování obrazu o velikosti 1280×1024 pixelů na zvolené hloubce rekurzivního sledování paprsků pro model Dragon o rozlišení $128 \times 92 \times 60$ voxelů.

metoda	hloubka rekurze	Bunny			Dragon			Buddha		
		64 x 64 x 52	128 x 128 x 100	256 x 256 x 200	64 x 48 x 32	128 x 92 x 60	256 x 184 x 116	28 x 64 x 28	56 x 128 x 56	108 x 256 x 108
3DDDA na CPU bez SSE	1	120	132	151	98	102	109	183	189	209
	2	179	202	235	130	142	157	330	367	415
	3	244	283	340	170	193	220	507	596	690
3DDDA na CPU s SSE	1	54	66	88	38	43	52	90	101	124
	2	90	113	153	57	70	88	176	212	265
	3	133	173	239	83	106	139	285	368	472
3DDDA na GPU CUDA	1	70	116	175	44	60	85	103	143	205
	2	137	220	312	81	120	169	208	322	478
	3	228	360	508	141	218	305	372	617	919

Tabulka 5.22: Vliv nastavené hloubky rekurzivního sledování odražených i lomených paprsků na čas generování obrazu v milisekundách.

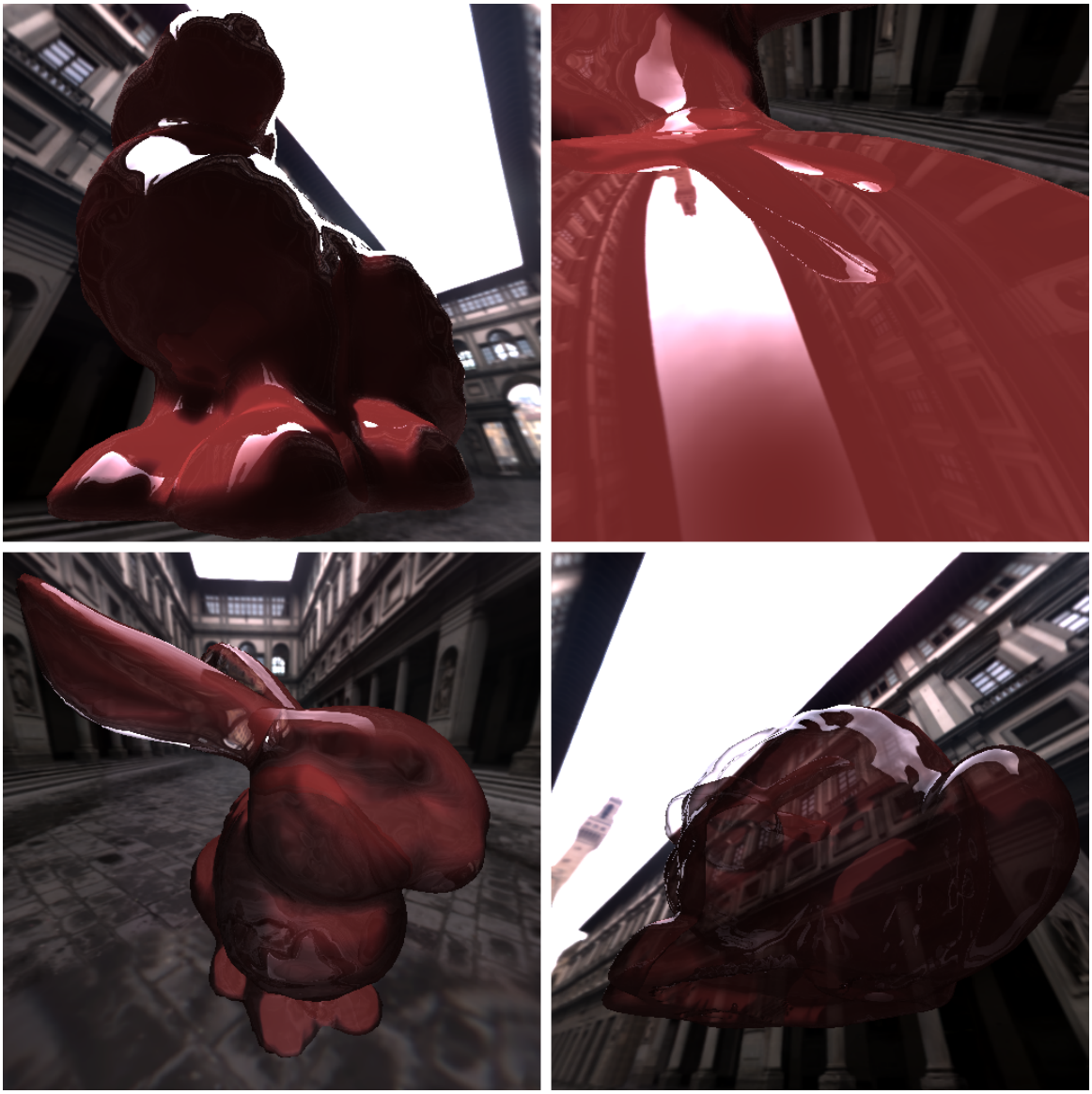


Obrázek 5.12: Závislost času generování obrazu o velikosti 1280×1024 pixelů na zvolené hloubce rekurzivního sledování paprsků pro model Dragon o rozlišení $128 \times 92 \times 60$ voxelů.

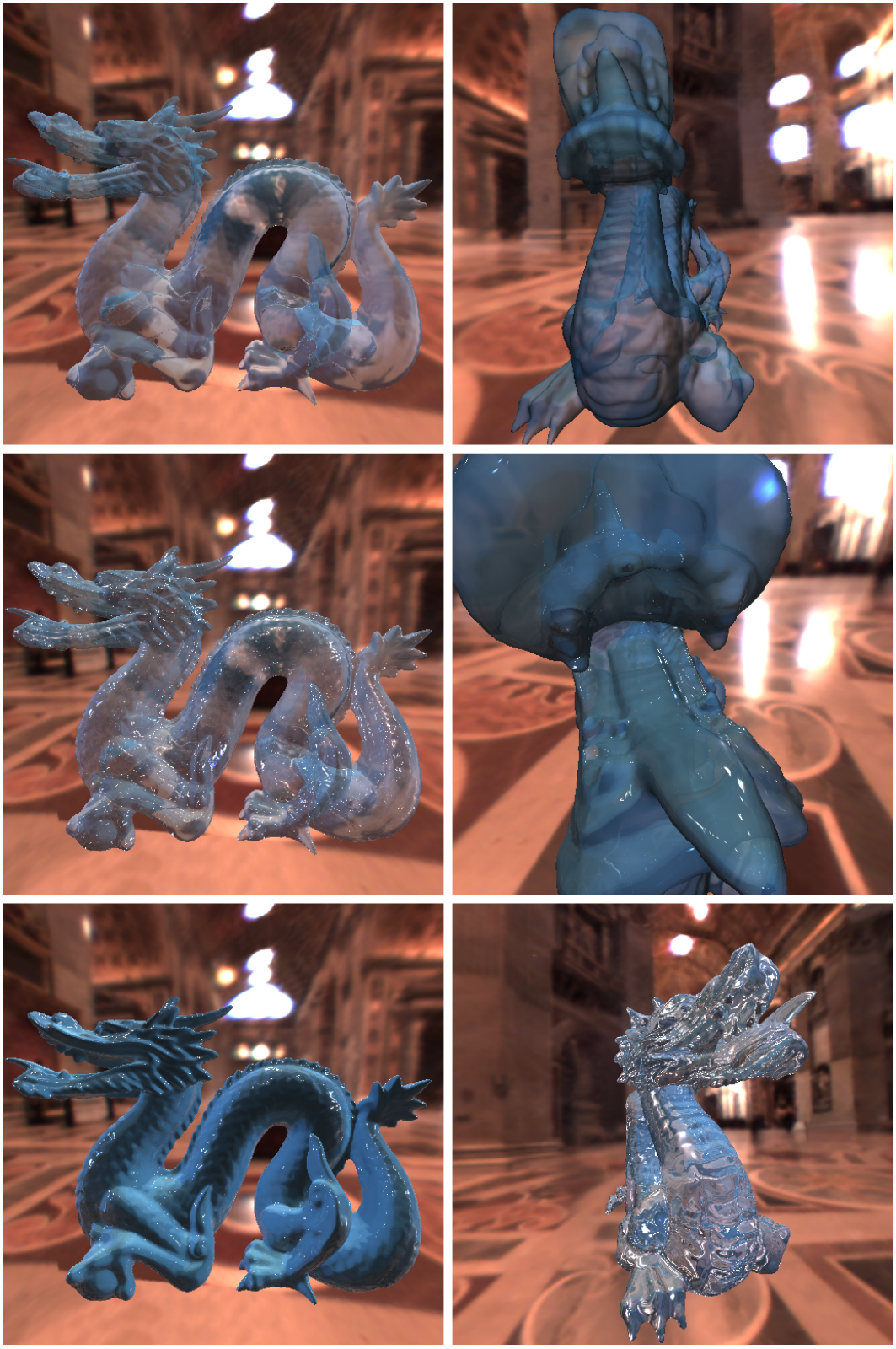
model	rozlišení modelu [voxelů]	počet paprsků v hloubce		
		1	2	3
Bunny	64x64x52	1 310 720	254 534	258 510
	128x128x100	1 310 720	296 041	303 777
	256x256x200	1 310 720	340 215	350 790
Dragon	64x48x32	1 310 720	142 629	164 317
	128x92x60	1 310 720	187 538	221 006
	256x184x116	1 310 720	190 642	226 873
Buddha	28x64x28	1 310 720	600 607	679 849
	56x128x56	1 310 720	660 167	767 574
	108x256x108	1 310 720	737 989	863 727

Tabulka 5.23: Přehled počtu sledovaných paprsků v různé hloubce stromu paprsků. Pro všechny modely generovány odražené i lomené paprsky.

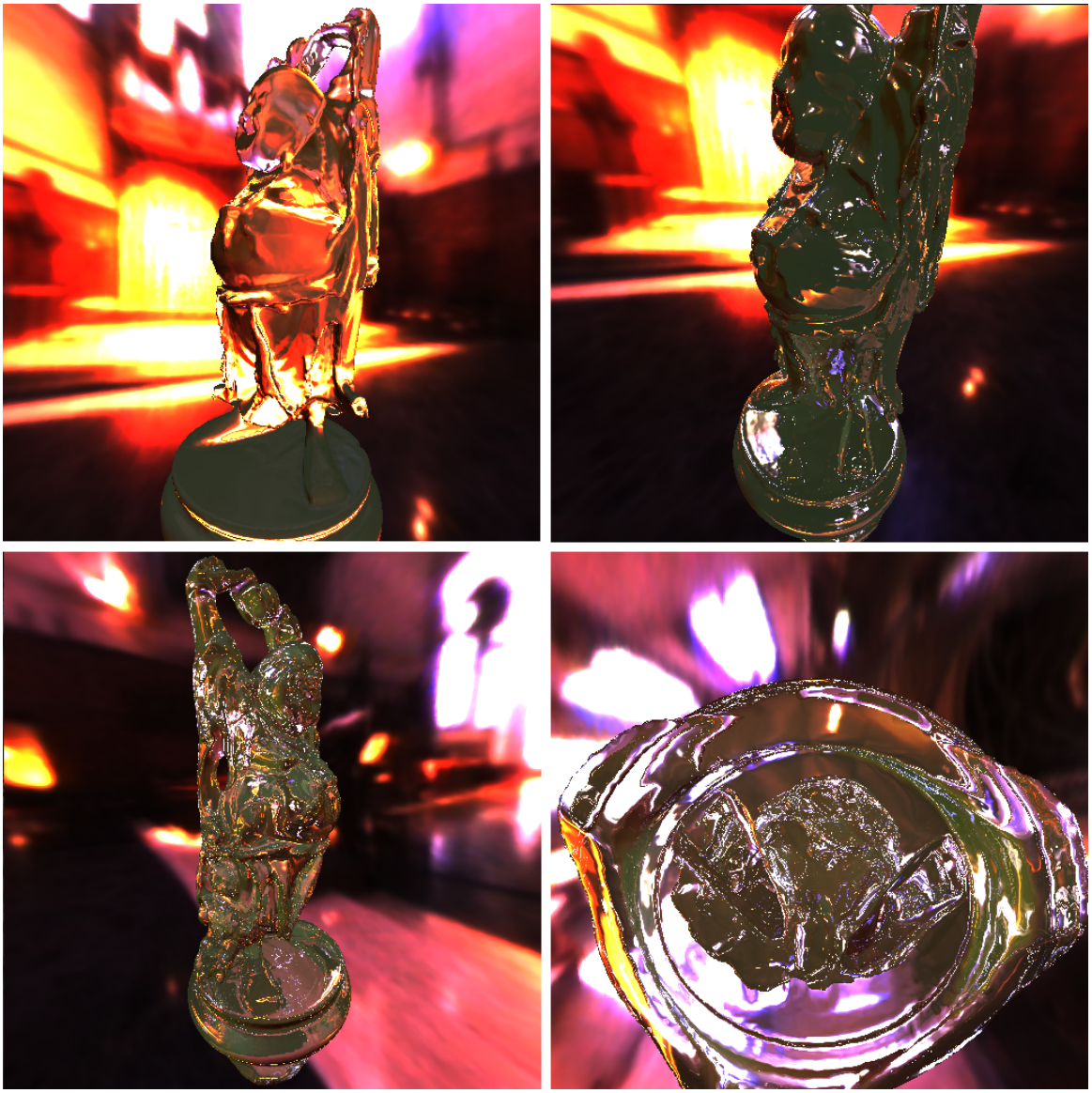
na všach modelech při rozlišení obrazu 1280×1024 pixelů a bylo použito traverzování dvoúrovňové mřížky. Tabulka 5.23 poté ukazuje počty sledovaných paprsků v různých hloubkách rekurze. Jelikož jsou pro každý paprsek, který zasáhne povrch objektu vygenerovány dva další paprsky, množství paprsků při zvyšující se hloubce rekurze vzrůstá a úměrně tomu vzrůstají časy generování obrazu.



Obrázek 5.13: Ukázka modelu Bunny v rozlišení $64 \times 64 \times 52$ voxelů (levé dva) a v rozlišení $128 \times 128 \times 100$ voxelů získaných sledováním i sekundárních paprsků.



Obrázek 5.14: Ukázka modelu Dragon v rozlišení $256 \times 184 \times 116$ voxelů.



Obrázek 5.15: Ukázka modelu Buddha v rozlišení $108 \times 256 \times 108$ voxelů.

Ukázky modelu Bunny získané použitím rekurzivního sledování paprsků jsou na obrázku 5.13. Pro získání horních dvou obrázků byly sledovány pouze odražené paprsky do hloubky 3. U dolních dvou obrázků byly sledovány odražené i lomené paprsky do hloubky 6. Index lomu levého dolního obrázku je nastaven na hodnotu 1,15, index lomu pravého dolního na hodnotu 1,0. Pro všechny obrázky je povoleno použití Fresnelova faktoru.

Na obrázku 5.14 jsou ukázky modelu Dragon. Levé tři obrázky ukazují postupně shora dolů použití pouze lomených paprsků, použití lomených i odražených paprsků a použití pouze odražených paprsků. Hloubka sledování paprsků je vždy nastavena na hodnotu 8. Pravý horní obrázek byl získán povolením Fresnelova faktoru spolu s nastavením indexu lomu na hodnotu 1,0 a jsou sledovány pouze lomené paprsky. U obrázku vpravo uprostřed

je také započítán vliv Fresnelova faktoru a jsou generovány i odražené paprsky. Obrázek vpravo dole ukazuje vzhled modelu při nastavení indexu lomu na hodnotu 1,55 a jsou opět sledovány pouze lomené paprsky.

Obrázek 5.15 obsahuje model Buddha. Pro získání levého horního obrázku bylo použito pouze lomených paprsků při hloubce rekurze 1. Byly tedy vyslány jenom primární paprsky a po nalezení průsečíku byla barva pixelu nastavena na barvu textury okolí ze směru, do kterého by byl vyslán sekundární paprsek. U pravého horního obrázku jsou generovány pouze odražené paprsky a hloubka rekurze byla nastavena na hodnotu 4. Dolní dva obrázky ukazují model získaný sledováním lomených paprsků do hloubky 4 při indexu lomu nastaveném na hodnotu 1,55.

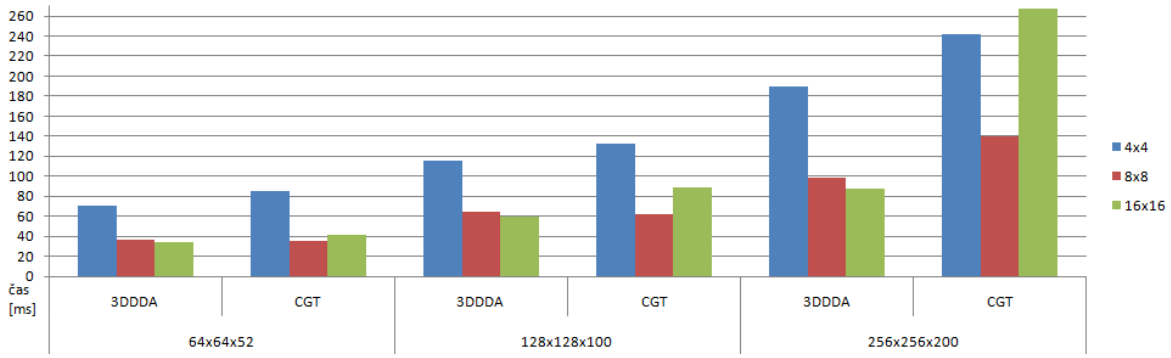
5.11 Závislost velikosti bloků vláken na čas

Rychlost výpočtu na grafické kartě pomocí architektury CUDA nebo jazyka OpenCL je závislá na velikosti lokální práce. U architektury CUDA se hovoří o velikosti bloků vláken, u jazyka OpenCL o počtu pracovních jednotek v pracovní skupině. Tabulka 5.24 spolu s grafy na obrázcích 5.16 a 5.17 ukazuje vliv zvolené velikosti bloků, respektive pracovních skupin na výsledný čas generování obrazu o velikosti 1280×1024 pixelů modelu Bunny.

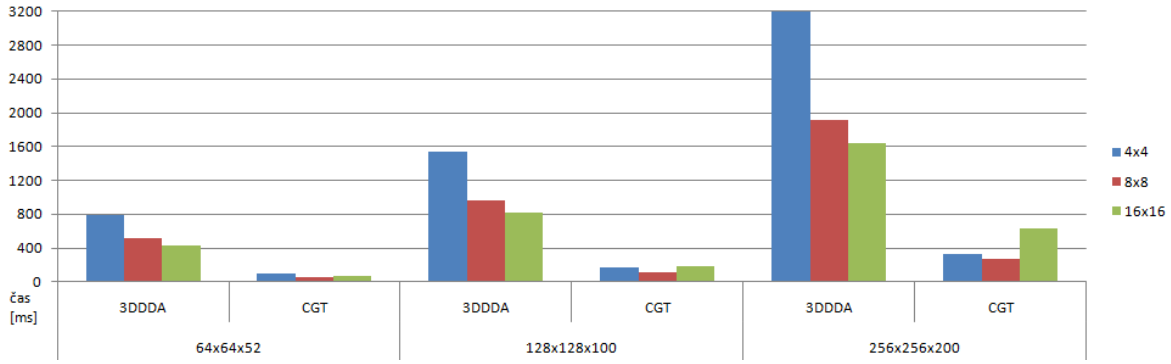
Bylo zjištěno, že pro algoritmus 3DDDA je nejlepší nastavit velikost bloku na hodnotu 16×16 vláken. Naproti tomu algoritmus CGT dosahuje nejlepších výsledků pro bloky o velikosti 8×8 vláken, neboť výsledné pakety poté nepokrývají tak velkou oblast mřížky. Problematika volby vhodné velikosti paketu byla popsána v kapitole 5.8.

rozlišení modelu [voxelů]	velikost bloku [vláken]	CUDA		OpenCL	
		3DDDA	CGT	3DDDA	CGT
64x64x52	4x4	71	85	790	104
	8x8	37	35	514	58
	16x16	34	42	433	74
128x128x100	4x4	116	133	1538	170
	8x8	65	62	965	117
	16x16	59	88	814	184
256x256x200	4x4	190	242	3206	328
	8x8	99	139	1916	278
	16x16	87	267	1638	632

Tabulka 5.24: Závislost času generování obrazu v ms na velikosti bloků, respektive na velikosti pracovních skupin metod generujících obraz na grafické kartě.



Obrázek 5.16: Vliv velikosti bloků na čas generování obrazu na GPU algoritmy 3DDDDA a CGT při použití architektury CUDA.



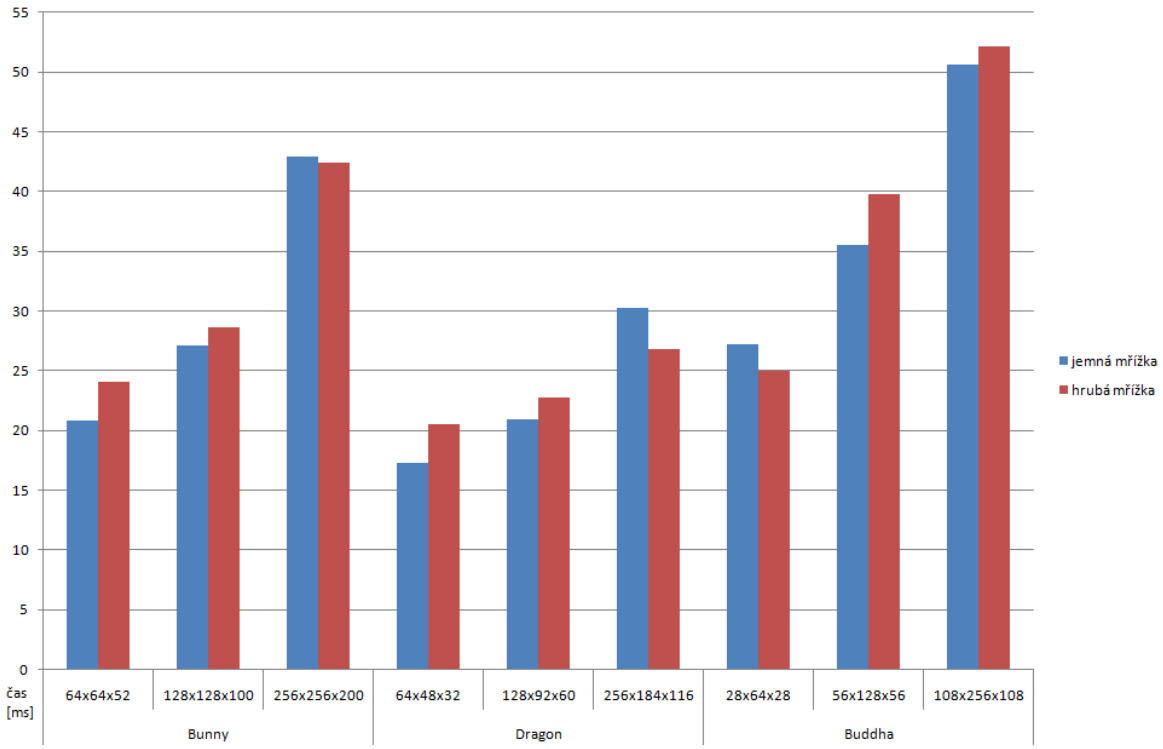
Obrázek 5.17: Vliv velikosti pracovních skupin na čas generování obrazu na GPU algoritmy 3DDDDA a CGT při použití jazyka OpenCL.

5.12 Porovnání časů výpočtu obrazu programem DFRRT při rasterizaci buněk jemné a hrubé mřížky

Program DFRRT umožňuje zvolit, zda mají být do paměti hloubky rasterizovány polygonální krychle představující buňky jemné mřížky obsahující povrch objektu nebo krychle představující buňky hrubé mřížky obsahující povrch objektu. Vliv na čas generování obrazu o rozlišení 1280×1024 pixelů prezentuje tabulka 5.25 a graf na obrázku 5.18. Z nich vyplývá, že záleží na použitém modelu a jeho rozlišení. V praxi může být tedy preferována rasterizace buněk hrubé mřížky, neboť stačí udržovat menší množství polygonálních krychlí.

model	rozlišení modelu [voxelů]	počítač 1 jemná mřížka	počítač 1 hrubá mřížka	počítač 2 jemná mřížka	počítač 2 hrubá mřížka
Bunny	64x64x52	21	24	57	61
	128x128x100	27	29	65	76
	256x256x200	43	42	96	100
Dragon	64x48x32	17	21	45	48
	128x92x60	21	23	52	60
	256x184x116	30	27	71	71
Buddha	28x64x28	27	25	63	72
	56x128x56	35	40	83	95
	108x256x108	51	52	117	125

Tabulka 5.25: Závislost času generování obrazu v ms na rasterizaci buněk jemné a hrubé mřížky metody omezující interval traverzování primárních paprsků.



Obrázek 5.18: Porovnání časů generování obrazu při rasterizaci buněk jemné a hrubé mřížky.

Kapitola 6

Závěr

V práci byly úspěšně implementovány různé algoritmy pro rychlé zobrazování volumetrických dat. Implementace na CPU byly zrychleny použitím SSE instrukcí a paralelizací výpočtu pomocí knihovny OpenMP. Dále byly zobrazovací algoritmy implementovány na GPU pomocí architektury CUDA i jazyka OpenCL. Program vytvořený použitím architektury CUDA umožňuje využít více grafických karet a generovat tak současně několik obrazů, díky čemuž může být stereoskopické zobrazení stejně rychlé jako zobrazování jediného pohledu. Byl také implementován program testující vliv metody omezení intervalu traverzování primárních paprsků rasterizací buněk mřížky obsahujících povrch na rychlost zobrazení, který umožňuje samotný proces výpočtu obrazu dále urychlit za cenu udržování aktuální množiny polygonálních krychlí představujících buňky mřížky obsahující povrch zobrazovaného objektu. Za účelem využití všeho výkonu dostupného hardware pro generování obrazu byla vytvořena i aplikace počítající část obrazu na CPU a část na GPU. Nakonec byla nejvhodnější metoda pracující na GPU upravena do podoby knihovny, kterou může program Myslbek použít jako zobrazovací modul a program Myslbek byl příslušně upraven, aby mohl tuto knihovnu využít.

U implementovaných algoritmů byl testován vliv použití různých metod hledání průsečíku paprsku s povrchem a metod výpočtu gradientu distanční funkce. Nejrychlejší metodou nalezení průsečíku poskytující dobré výsledky je metoda lineární interpolace. Pro výpočet gradientu je u algoritmů pracujících na CPU a algoritmů napsaných v jazyce OpenCL vhodné použít metodu interpolace gradientů v rozích získaných centrální diferencí a u algoritmů využívajících architekturu CUDA je rychlejší počítat gradienty přímo centrální diferencí. Implementované algoritmy lze z hlediska toho, zda používají pro traverzování mřížky algoritmus 3DDDA nebo algoritmus CGT, rozdělit na dvě části. Při generování obrazu na CPU je ve většině případů rychlejší použít algoritmus CGT. Naopak při počítání obrazu na grafické kartě pomocí architektury CUDA je rychlejší algoritmus 3DDDA. Vlivem nedokonalosti překladače jazyka OpenCL je výpočet obrazu na GPU s použitím tohoto jazyka rychlejší při traverzování mřížky algoritmem CGT.

U algoritmu pracujícího na CPU, traverzujícího mřížku algoritmem 3DDDA a využívajícího SSE instrukce jsou vždy 2×2 paprsky sloučeny do jednoho paketu. Oproti algoritmu, který SSE instrukce nevyužívá, je zmíněný algoritmus při traverzování uniformní mřížky 2,7 až 3× rychlejší, ale při traverzování dvouúrovňové mřížky je rychlejší pouze 1,5 až 2,5×. Proto bylo zkoušeno rozdělovat paket při traverzování dvouúrovňové mřížky na jednotlivé paprsky, pokud již všechny paprsky paketu nejsou aktivní. Ukázalo se, že tento mechanismus

další zrychlení nepřináší. Algoritmus pracující na CPU a využívající algoritmus CGT a SSE instrukce je oproti algoritmu používajícímu 3DDDA bez SSE instrukcí rychlejší 2,6 až 7,3× při traverzování uniformní mřížky a 1,5 až 3,7× při traverzování dvouúrovňové mřížky. Bylo zjištěno, že vhodná velikost paketů pro tento algoritmus je velikost 4×4 nebo 8×8 vláken a ukázalo se, že pro dosažení lepších časů tvorby obrazu není nutné vyhýbat se prohledávání buněk jemné mřížky, pro které buňky hrubé mřížky neindikují přítomnost povrchu, při přechodu z traverzování hrubé mřížky na traverzování jemné mřížky. Také bylo potvrzeno, že algoritmus CGT není vhodné používat pro sekundární paprsky, které jsou méně koherentní než primární paprsky.

Algoritmus pracující na GPU využívající architekturu CUDA a traverzující mřížku algoritmem 3DDDA je rychlejší než podobný algoritmus používající jazyk OpenCL i než ostatní algoritmy pracující na GPU a traverzující mřížku algoritmem CGT. Na dostupném hardware nebyl tento algoritmus vždy rychlejší než algoritmus pracující na CPU a využívající pro traverzování mřížky algoritmus CGT. Avšak pomocí grafické karty přítomné v počítači 1, jehož parametry jsou uvedeny v tabulce 5.1, lze za uvedený čas vygenerovat dva obrazy. Nevýhodou metod pracujících na GPU tak zůstává pouze horší odezva na zvyšující se hloubku rekurzivního sledování paprsků.

6.1 Možná pokračování práce

Jedním z dalších pokračování práce by mohlo být otestování algoritmů využívajících architekturu CUDA na nejnovějších kartách nVidia s Compute Capability 2.0, zjištění vlivu použití cache pamětí na celkový výkon a případná optimalizace algoritmů pro tyto karty. Vlastností nových karet by mohl využít zejména program DFRRT, pro který by bylo zajímavé implementovat rasterizaci polygonálních krychlí pomocí architektury CUDA. Díky tomu by zmizel problém s obtížnou paralelizací výpočtu na více grafických karet a bylo by odstraněno i pomalé kopírování dat mezi pamětí hloubky a pamětí, ze které může kernel číst data. Dále by bylo pro tento program potřeba implementovat algoritmus pro rychlou přípravu polygonálních krychlí z distančního pole a celý program by mohl být upraven tak, aby sloužil jako nový, ještě rychlejší zobrazovací modul programu Myslbek.

Literatura

[1] The CImg Library. http://cimg.sourceforge.net. C++ Template Image Processing Toolkit.

[2] Wikipedia, marching cubes. http://en.wikipedia.org/wiki/Marching_cubes.

[3] Wikipedia, ray tracing (graphics). http://en.wikipedia.org/wiki/Ray_tracing_(graphics).

[4] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, Aug. 1987.

[5] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM.

[6] R. Bridson. *Computational aspects of dynamic surfaces*. PhD thesis, Stanford University, 2003.

[7] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[8] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: accelerated ray-tracing system. pages 148–159, 1988.

[9] S. F. F. Gibson. Using distance maps for accurate surface representation in sampled volumes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 23–30, New York, NY, USA, 1998. ACM.

[10] C. P. Gribble, T. Ize, A. Kensler, I. Wald, and S. G. Parker. A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):758–768, 2007.

[11] M. Harris. *Optimizing Parallel Reduction in CUDA*. nVidia.

[12] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, Aug. 12 1985.

[13] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. Chapter 39. Volume rendering techniques. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, 2004.

[14] O. Jamriška. Sledování paprsku pro interaktivní zobrazování povrchu zadaného distanční funkcí. Bachelor's thesis, České vysoké učení technické v Praze, 2009.

[15] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM.

[16] Khronos Group. *The OpenCL Specification*, October 2009. Version: 1.0. Document Revision: 48.

[17] A. M. Knoll, I. Wald, and C. D. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer: International Journal of Computer Graphics*, 25(3):209–225, 2009.

[18] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM.

[19] G. Marmitt, A. Kleer, I. Wald, H. Friedrich, and P. Slusallek. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. pages 429–435, 2004.

[20] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 100–107, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[21] M. E. Mortensen. *Geometric Modeling.* John Wiley & Sons, Inc., New York, NY, USA, 1985.

[22] A. Neubauer, L. Mroz, H. Hauser, and R. Wegenkittl. Cell-based first-hit ray casting. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 77–86, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[23] nVidia. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi.* Whitepaper.

[24] nVidia. *NVIDIA CUDA Programming Guide*, April 2009. Version 2.2.

[25] nVidia. *NVIDIA CUDA Best Practices Guide*, February 2010. Version 3.0.

[26] nVidia. *NVIDIA CUDA Programming Guide*, February 2010. Version 3.0.

[27] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive ray tracing for isosurface rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 233–238, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[28] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[29] J. Žára, B. Beneš, and P. Felkel. *Moderní počítačová grafika.* Computer Press s.r.o, Brno, 1st edition, 1998. In Czech.

[30] J. Schwarze. Cubic and quartic roots. *Graphics gems*, pages 404–407, 1990.

[31] B. Smits. Efficiency issues for ray tracing. *journal of graphics, gpu, and game tools*, 3(2):1–14, 1998.

[32] D. Triolet. nVidia CUDA: practical uses. http://www.behardware.com/articles/678-1/nvidia-cuda-practical-uses.html, August 2007.

[33] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006.

[34] T. Whitted. An improved illumination model for shaded display. *ACM Communication on Graphics*, 23(6):343–349, 1980.

[35] A. Williams, S. Barrus, R. K. Morley, and P. Shirley. An efficient and robust ray-box intersection algorithm. *journal of graphics, gpu, and game tools*, 10(1):49–54, 2005.

[36] M. Zlatuška. Metoda sledování paprsku na grafických akcelerátorech. Master's thesis, České vysoké učení technické v Praze, 2009.

Příloha A

Seznam použitých zkratek

2D Two-Dimensional

3D Three-Dimensional

3DDDA 3D Digital Differential Analyzer

AABB Axis-Aligned Bounding Box

ADF Adaptively sampled Distance Field

CGT Coherent Grid Traversal

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CSG Constructive Solid Geometry

DFRRT Distance Field Rasterization and Ray Tracing

DFRT Distance Field Ray Tracing

GLSL OpenGL Shading Language

GPU Graphic Processing Unit

HLSL High Level Shading Language

LOD Level Of Detail

OpenCL Open Computing Language

OpenGL Open Graphics Library

OpenMP Open Multi-Processing

SIMD Single-Instruction, Multiple-Data

SIMT Single-Instruction, Multiple-Thread

SSE Streaming SIMD Extensions

Příloha B

Instalační a uživatelská příručka

B.1 Program DFRT a DFRRT

Oba programy mohou pracovat v operačním systému Windows i v systému Linux. Pro plnou funkčnost programů je nezbytným vybavením použitého počítače grafická karta nVidia podporující architekturu CUDA. Program DFRT navíc vyžaduje procesor podporující instrukční sadu SSE4. Programy byly testovány ve 32-bitové verzi OS Windows XP s ovladači grafické karty verze 197.13, ve 32-bitové verzi OS Fedora Linux s ovladači 195.36.15 a na notebooku s 64-bitovým OS Windows 7 s ovladači verze 197.16. Pro překlad a spuštění aplikací je vyžadován 32-bitový CUDA Toolkit (testováno s verzí 3.0) a 32-bitový nVidia GPU Computing SDK podporující jazyk OpenCL (testováno s verzí 3.0 podporující OpenCL verze 1.0).

B.1.1 Nastavení parametrů překladu programu DFRT

Mezi zdrojovými soubory programů se nacházejí soubory **setup.h** umožňující snadno nastavit různé možnosti překladu programů. U programu DFRT lze definovat výchozí metodu, kterou bude obraz generován po spuštění programu a je možné zvolit, že bude traverzována pouze uniformní mřížka. Pro metodu traverzující mřížku algoritmem 3DDDA na CPU bez použití SSE instrukcí je možné zvolit metody výpočtu průsečíku paprsku s povrchem objektu a metodu výpočtu gradientu. U metody používající SSE instrukce lze aktivovat dělení paprsku v libovolné kombinaci tří situací a při libovolném počtu zbývajících paprsků. U algoritmu CGT pracujícího na CPU lze nastavit velikost paketu zadáním počtu sub-paketů o velikosti 2×2 paprsky vedle sebe a nad sebou. Také lze povolit ořez buněk jemné mřížky, pro které hrubá mřížka neindikuje přítomnost povrchu.

U metod pracujících na GPU může být výpočet gradientů centrální diferencí nahrazen jejich výpočtem trilineární interpolací gradientů v rozích získaných centrální diferencí a u metody traverzující algoritmem 3DDDA a používající architekturu CUDA i výpočtem gradientů dopřednou diferencí. Může být nastaveno, aby byl program přeložen bez metod využívajících architekturu CUDA nebo bez metod používajících jazyk OpenCL. Lze nastavit počet grafických karet, které mají metody s architekturou CUDA využívat, ale tento počet by neměl být vyšší než v konfiguračním souboru nastavený počet programových vláken. Také je možné povolit překlad metod používajících architekturu CUDA s hloubkou rekurze 2 nebo 6 a lze nastavit velikost bloků vláken.

Pro metody používající jazyk OpenCL je kromě toho, zda má být traverzována uniformní nebo dvouúrovňová mřížka a způsobu výpočtu gradientu možné nastavit ještě velikost pracovních skupin. Tato tři nastavení je třeba provést i v souborech **OpenCL_3DDDA.cl** a **OpenCL_CGT.cl**. Pokud jsou tyto soubory změněny, je třeba odstranit jejich přeložené verze z adresáře `exe`, aby po prvním spuštění programu došlo k jejich novém překladu. Nakonec je možné v souboru **setup.h** zapnout automatické měření času, které začne u zvolené výchozí metody, postupně zapíše do souboru časy generování obrazu měřených metod a skončí u nastavené poslední metody.

B.1.2 Nastavení parametrů překladu programu DFRRT

Hlavičkový soubor **setup.h** programu DFRRT umožňuje zvolit, jestli mají být rasterizovány buňky hrubé nebo jemné mřížky. Pro primární a sekundární paprsky lze zvlášť nastavit, zda mají traverzovat dvouúrovňovou nebo uniformní mřížku. Program může být přeložen pro hloubku rekurzivního sledování paprsků 1, 2 nebo 6. Je možné nastavit velikost bloků vláken a povolit měření a zobrazování časů rasterizační, traverzační a zobrazovací části programu. Stejně jako u programu DFRT lze program přeložit tak, aby proběhlo automatické měření času generování obrazu.

B.1.3 Překlad programů

Ke zdrojovým souborům obou testovacích programů jsou přiloženy soubory projektů vytvořených pomocí MS Visual Studia 2008 i Makefile sloužící pro překlad programů v operačním systému Linux. Před samotným překladem programů ve Visual Studiu 2008 je vhodné zkontrolovat, že je správně nastavena systémová proměnná `CUDA_LIB_PATH` udávající cestu k adresáři `lib` v kořenovém adresáři CUDA Toolkitu a proměnná `NVSDKCOMPUTE_ROOT` obsahující cestu ke kořenovému adresáři nVidia GPU Computing SDK. Dále je vhodné pomocí Visual Studia ve vlastnostech projektů nastavit číslo Compute Capability grafické karty, pro kterou má být program přeložen.

Před překladem programů v operačním systému Linux je třeba v souborech **Makefile** obou programů nastavit proměnnou `ROOTDIR`, aby udávala cestu ke složce `C/common` v kořenovém adresáři obsahujícím nVidia GPU Computing SDK. V souborech **common.mk** je poté nutné zkontrolovat, zda je správně nastavena cesta ke složce obsahující CUDA Toolkit a nakonec je vhodné zadat odpovídající čísla Compute Capability.

B.1.4 Spuštění programů

Přeložené programy by měly být spouštěny z adresáře `exe` nebo pomocí dávkových souborů v adresářích `bat`. Vstupními parametry obou programů jsou šířka obrazu v pixelech, výška obrazu v pixelech, jméno souboru s modelem umístěným v adresáři `data/model` bez přípony, jméno adresáře s texturami okolí umístěným v adresáři `data/cubemap` a jméno konfiguračního souboru umístěného v adresáři `config` bez přípony. Příkaz pro spuštění programu v OS Windows může být například:

```
dfrrt.exe 512 512 bunny_64_4 uffizi bunny
```

a v OS Linux například:

```
./dfrt 1024 768 dragon_256_4 stpeters dragon
```

B.1.5 Struktura konfiguračních souborů

Konfigurační soubory mají pevně danou strukturu. Proto se doporučuje vytvářet nové konfigurační soubory zkopírováním a úpravou souborů stávajících. Struktura souboru spolu s hodnotami použitými pro model Bunny je patrná z následující ukázky.

```
tileWidth       64
tileHeight      64
numThreads      2
objectColorR    0.9
objectColorG    0.3
objectColorB    0.3
specularCoef    0.4
transparentCoef 0.0
FOV             1.0
recursionDepth  1
refractiveIndex 1.1
useFresnel      1
```

První dva parametry udávají velikost obrazových dlaždic použitých pro paralelizaci výpočtu. Následující parametr určuje počet programových vláken použitých pro výpočet. Je vhodné nastavit jejich počet na celkový počet jader použitého počítače. Parametry `objectColor` dovolují specifikovat barvu modelu pomocí složek RGB. Následují hodnoty koeficientů odrazu a průhlednosti, zorné pole kamery, výchozí hloubka rekurzivního sledování paprsků a index lomu materiálu modelu. Nastavením nenulové hodnoty posledního parametru je povoleno použití Fresnelova faktoru pro výpočet osvětlení.

V	Přepínání mezi zobrazováním různých informací
W/S	Změna koeficientu odrazu
E/D	Změna koeficientu průhlednosti
R/F	Změna hloubky rekurzivního sledování paprsků
T/G	Změna indexu lomu
Mezerník	Povolení použití Fresnelova faktoru
Esc, Q	Ukončení programu
X/C	Změna metody výpočtu obrazu
U/J	Nastavení počtu řádků obrazových dlaždic generovaných na GPU
K	Zapnutí automatického nastavení počtu dlaždic generovaných na GPU

Tabulka B.1: Seznam kláves pro ovládání programu DFRT (celá tabulka) a programu DFRRT (pouze první část tabulky).

B.1.6 Ovládání programů

Po spuštění lze programy ovládat pomocí myši a klávesnice. Posunem myši při stisknutém levém tlačítku se mění poloha kamery sledující model. Pomocí kolečka myši se lze k modelu přibližovat nebo se vzdalovat. Seznam kláves sloužících pro změnu některých parametrů a další ovládání programu je v tabulce B.1. Klávesy z druhé části tabulky jsou určeny pro použití pouze v programu DFRT.

B.2 Knihovna GPU DFRT

B.2.1 Překlad knihovny

Ke zdrojovým souborům zobrazovací knihovny je přiložen soubor **Makefile** sloužící pro překlad knihovny v operačním systému Linux. V tomto souboru je třeba nastavit hodnotu proměnné `ROOTDIR` na cestu ke složce `C/common` v adresáři obsahujícím nVidia GPU Computing SDK. V souboru **common.mk** musí být správně nastavena cesta ke složce obsahující CUDA Toolkit a je vhodné nastavit čísla Compute Capability, aby odpovídala parametrům karty, pro kterou je knihovna překládána.

B.2.2 Použití knihovny

V programu, který bude zobrazovací knihovnu využívat, je třeba vytvořit instanci třídy knihovny. Při jejím vytváření se jako parametr předávají rozměry distančního pole, požadované rozlišení obrazu, zorné pole kamery a souřadnice minimálního a maximálního bodu kvádru obklopujícího mřížku. Má-li být obraz generován pomocí několika grafických karet, je nutné vytvořit více instancí třídy, každé pomocí jiného programového vlákna a poté je třeba všechny operace provádět shodně se všemi instancemi pomocí příslušných vláken. Dále je potřeba v knihovně zaregistrovat světla, nastavit úroveň ambientního osvětlení a specifikovat lesklost objektu.

Pro vygenerování obrazu slouží funkce `render_gpu()`, jejímiž parametry jsou pole pro uložení vytvořeného obrazu a aktuální zobrazovací matice velikosti 4×4. Při aktualizaci distančního pole je třeba pro všechny změněné bloky volat funkci `write_surface_gpu()`, `write_inside_gpu()` nebo `write_outside_gpu()` v závislosti na tom, kde se změněný blok nachází. Parametry těchto funkcí jsou souřadnice bloku a u funkce `write_surface_gpu()` i nové distanční pole a pole barev bloku. Vyprázdnit celé distanční pole lze pomocí funkce `clear_gpu()`. Pokud je měněna velká část distančního pole najednou, například při nahrání nového objektu ze souboru, je vhodnější neaktualizovat v grafické paměti každý blok zvlášť, ale použít funkce `set_surface_gpu()`, `set_inside_gpu()` nebo `set_outside_gpu()` a teprve poté aktualizovat data v grafické paměti zavoláním funkce `copy_to_gpu()`. Před ukončením programu je vhodné všechny vytvořené instance třídy uvolnit.

Příloha C

Obsah přiloženého CD

- [CD]
  - [Myslbek]
    - [gpudfrt] — zdrojové soubory zobrazovací knihovny GPU DFRT, Makefile pro systém Linux a přeložená knihovna
    - [Myslbek] — upravená verze programu Myslbek spustitelná bez připojení trackovacích kamer a používající knihovnu GPU DFRT
    - [patch] — nové a upravené soubory programu Myslbek
  - [program]
    - [bat dfrrt] — dávkové soubory pro spuštění programu DFRRT s různými parametry
    - [bat dfrt] — dávkové soubory pro spuštění programu DFRT s různými parametry
    - [config] — konfigurační soubory modelů používané programem DFRT a DFRRT
    - [data]
      - [cubemap] — textury mapy okolí
      - [model] — modely objektů v různém rozlišení uložené v distančním poli
    - [dfrrt] — zdrojové soubory programu DFRRT, projekt pro MS Visual Studio 2008 a Makefile pro systém Linux
      - [gl] — knihovna glew a knihovna glut verze 3.7 podporující kolečko myši
    - [dfrt] — zdrojové soubory programu DFRT, projekt pro MS Visual Studio 2008 a Makefile pro systém Linux
    - [exe] — spustitelné soubory programu DFRT a DFRRT a potřebné dynamické knihovny pro systém Windows
  - [text]
    - [src] — zdrojové soubory textu DP v programu LaTeX
    - [Vavra-thesis-2010.pdf] — text DP ve formátu pdf