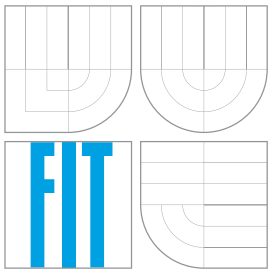
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

PATHTRACING NA GPU

PATHTRACING ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

KAREL BŘEZINA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ POLOK

BRNO 2015

Abstrakt

Tato bakalářská práce se zaměřuje na akceleraci renderovací metody *pathtracing*. Cílem práce je demonstrace a srovnání výkonnosti implementací *pathtracingu* na CPU a GPU. Obě implementace budou využívat akcelerační datové struktury. Výsledky jednotlivých optimalizačních technik budou diskutovány a porovnávány mezi sebou. Dále budou rozebrány možnosti rozšíření stávající aplikace.

Abstract

This bachelor thesis focuses on the acceleration of rendering algorithm pathtracing. Thesis's goal is demonstration and performance comparison between implementation on CPU and GPU. Both implementations will be using accelerated data structures. Results of the specific optimization techniques will be discussed and compared between themselves. Furthermore, will be describe possibilities of extending existing application.

Klíčová slova

Sledování cest, Sledování paprsku, WinAPI, OpenGL, OpenCL, Bounding volume hierarchy, Octree, Uniformní mřížka, GPU, Paprsek.

Keywords

Pathtracing, Raytracing, WinAPI, OpenGL, OpenCL, Bounding volume hierarchy, Octree, Uniform Grid, GPU, Ray.

Citace

Karel Březina: Pathtracing na GPU, bakalářská práce, Brno, FIT VUT v Brně, 2015

Pathtracing na GPU

Prohlášení

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

.....
Karel Březina
14. května 2015

Poděkování

Rád bych poděkoval mému vedoucímu za jeho čas, který věnoval veškerým mým otázkám a vedení této práce.

Obsah

Kapitola 1

Úvod

Myšlenka vytváření fotorealistických scén v reálném čase je stará více než 40 let. V té době ale programátoři museli pracovat (z dnešního pohledu) s nevýkonnými a na výpočetní zdroje omezenými počítacími stroji. Přesto již tehdy přemýšleli o algoritmech, které by je posunuly ke kýženému výsledku.

Pathtracing (nebo-li sledování cest) patří mezi Monte Carlo[1] renderovací techniky. Pomocí této metody lze věrně zobrazit mnoho efektů (např. odlesky, odrazy nebo měkké stíny) dotvářející realističtější vzhled scény. Přesněji se jedná o modifikaci starší metody *raytracing* (detailněji popsanou v kapitole 2.2).

V porovnání s *raytracingem* jsou výsledky této novější metody věrnější realitě (počítá se zde i s vlivem okolního světla nebo s materiály objektů). Naproti těmto pozitivním výsledkům (které by mohli například představovat evoluci v produkci počítačových her) existuje i několik zásadních nedostatků. Hlavním nedostatkem je bezesporu časová náročnost obou algoritmů a částečná zašuměnost prvotních snímků, které metodu *pathtracing* doprovázejí (viz. obrázek 1.1 níže).

Obrázek 1.1: Scéna [11] renderovaná *pathtracingem* obsahující velký počet objektů (758K polygonů/trojúhelníků, mnoho textur) a zobrazující komplexní světelné podmínky (lesklé plochy, kaustiky, silné přímé světlo atd.). Obrázek vlevo byl počítán 50 vteřin a obrázek vpravo 1 hodinu. Standardní obousměrný *pathtracing* potřebuje na výpočet levého obrázku 11 minut a pravého obrázku 13 hodin k dosažení stejného výsledku. Zmíněné časy představují přibližný poměr mezi kvalitou a časem výpočtu.

[1] Monte Carlo metody sdružují širokou škálu algoritmů založených na náhodnosti použité při výpočtu.

Přestože je dnešní výpočetní síla moderních počítačů na vysoké úrovni (a některé jsou schopny vykreslovat *pathtracing* i v reálném čase), stále je potřeba přicházet s novými metodami, které by urychlovaly zpracování (mnohokrát komplexní) scény. Jednou z mnoha možností je zpracovávání *pathtracingu* na grafické kartě (kartách). Tyto karty obsahují několikanásobně více výpočetních jader, které mohou běžet současně (nebo-li paralelně). Některé výpočetní operace jsou rychleji zpracovávané na GPU (*graphic processing unit*) oproti CPU (*central processing unit*), takže je možné dosáhnout až 15x rychlejšího výpočtu [11].

Dalším způsobem urychlení výpočtu je využití tzv. akceleračních datových struktur. Tyto struktury slouží ke snížení počtu operací na testování průniku paprsku s objekty v místech, kde lze predikovat úspěch či neúspěch těchto operací. Uvedeným přístupem lze ušetřit mnoho výpočetního času a zvýšit tak výkonnost celé implementace. Bohužel i jmenované struktury mají svá slabá místa. Vlastnostem akceleračních datových struktur se věnuje kapitola 3.

Cílem projektu je vytvoření aplikace, která bude vhodným způsobem demonstrovat algoritmus *pathtracing* v předem připravené scéně pomocí grafického API OpenGL. K výpočtu tohoto algoritmu budou využity optimalizační techniky spojené s akceleračními datovými strukturami doprovázené porovnáváním výkonnosti jednotlivých struktur zpracovávaných na CPU a GPU. Dále bude převeden algoritmus *pathtracing* do jazyku OpenCL. Kód zapsaný v jazyku OpenCL je určen na programování speciálních výpočetních jednotek, mezi které se řadí i výpočetní jádra grafické karty. Aplikace bude přizpůsobena k přepínání mezi konfiguracemi výpočtu v reálném čase. Na výběr budou všechny typy implementovaných akceleračních datových struktur v kombinaci s výpočtem na CPU nebo GPU. Uživatel bude moci porovnávat rychlost řešení z informačních výpisů, které budou zobrazovány v okně aplikace. V kapitole 2 je popsána metoda *pathtracing* a její nezbytné součásti. Kapitola 3 popisuje použité akcelerační datové struktury. V kapitole 4 jsou představeny existující implementace *pathtracingu* v komerčních i nekomerčních aplikacích. Nástroje použité při implementaci aplikace jsou zmíněny v kapitole 5. Detailnějšímu popisu návrhu a implementace se věnuje kapitola 6. Předposlední kapitola 7 se zaměřuje na testování a vyhodnocení výsledků. V závěru jsou vyhodnoceny výsledky práce a navrhnuty možnosti rozšíření stávající aplikace.

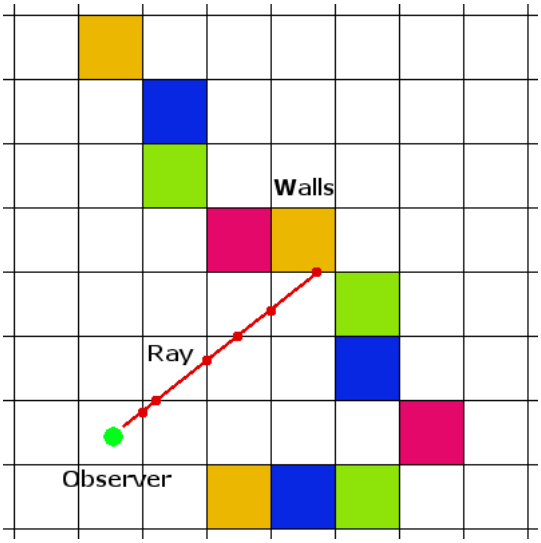
Kapitola 2

Popis metody pathtracing

Jak již bylo zmíněno v úvodu, *pathtracing* je rozšíření metody *raytracing*. Obě tyto varianty vycházejí z podobné metody známé jako *raycasting*. Pro lepší pochopení je vhodné si nejdříve popsat metody, ze kterých *pathtracinsg* vychází.

2.1 Raycasting

Metodu zvanou *raycasting* [1] (nebo-li vrhání paprsku) popsal v roce 1968 Arthur Appel. Metoda znázorňuje techniku vysílání paprsků do prostoru, kde se pro každý pixel obrazu generuje tzv. primární paprsek[2]. Vyslaný paprsek se pohybuje prostorem pomocí modifikovaného algoritmu DDA[3] a po každém posunutí v algoritmu je testováno, zda se paprsek nachází na okraji některého z objektů ve scéně (např. zeď místnosti). Pokud ano, výsledná barva pixelu bude definována podle barvy místa, kde byl objekt protnut. Tato technika je demonstrována na obrázku 2.1 níže.



Obrázek 2.1: Cesta paprsku od pozorovatele směrem do prostoru.

[2] Označuje se tak první paprsek vyslaný od pozice pozorovatele. Všechny následující paprsky jsou sekundární.

[3] *Digital differential analyzer.*

Principy *raycastingu* byly rovněž využity jako zobrazovací technika několika počítačových her v 90. letech minulého století (např. Wolfenstein 3D). *Raycasting* stál na počátku většího zájmu o 3D enginy a stal se základem pro další odvozené metody.

2.2 Raytracing

Raytracing [13] je jeden z prvních algoritmů řešící problém globálního osvětlení scény. Turner Whitted v roce 1979 představil svou rekurzivní implementaci, která vychází z metody *raycasting*. Slovo rekurzivní zde vyjadřuje opětovné vysílání paprsků[4] v místě průniku primárního paprsku s daným objektem. Paprsek zde ale již neprochází prostorem pomocí algoritmu DDA (jako v případě *raycastingu*), ale testuje průnik paprsku s jednotlivými primitivy každého objektu ve scéně. To způsobuje mnohem vyšší zátěž na výpočetní výkon počítače, ale výsledky jsou mnohem realističtější a lze touto metodou získat spoustu grafických efektů (např. měkké stíny nebo kaustiky). *Raytracing* se často používá pro výsledné zobrazení 3D modelů grafických editorů nebo pro tvorbu filmových animací, kde se generují snímky ve vysoké kvalitě. Přesto ani *raytracing* není tím nejlepším, co dokáže počítačová grafika v oblasti fotorealistického zpracování nabídnout.

2.3 Pathtracing

Jedna z nejznámějších modifikací *raytracingu* je metoda *pathtracing* [7]. Tato metoda je stochastická a opět se zabývá globálním osvětlením scény. Na rozdíl od *raytracingu* je po protnutí primárního paprsku s daným objektem náhodně rozhodnuto[5], zda bude paprsek odražen, lomen pod určitým úhlem nebo se bude zjišťovat výsledná barva v bodě průniku. Tato vlastnost *pathtraceru* způsobuje počáteční zašumění několika prvotních snímků. Výsledná barva pixelu se však s přibývajícím počtem vzorků zpřesňuje. Všechny získané vzorky barev pro daný pixel se kombinují poměrnou částí, kdy barevný vzorek získaný později při sledováni trasy paprsku má menší váhu než vzorky získané dříve a neovlivňuje tak výslednou barvu pixelu. V dalších sekcích jsou uvedeny jednotlivé kroky výpočtu barvy.

2.4 Vytvoření a výpočet průniku paprsku

Dříve než začne výpočet, je potřeba vytvořit primární paprsek, jehož trasa bude sledována. Paprsek si lze představit jako polopřímku mající počátek v bodě pozice pozorovatele a směr, kterým směřuje do prostoru, vychází ze zorného pole pozorovatele[6] v kombinaci s rozlišením výsledného snímku. Dále je potřeba specifikovat, do jaké minimální a maximální vzdálenosti bude paprsek vržen do prostoru. Maximální vzdálenost nám definuje hranici, za kterou je vržený paprsek vyhodnocen jako neúspěšný a vrátí definovanou barvu okolí. Teprve po vyplnění všech těchto informací je možné paprsek použít k výpočtu. Pro zjištění pozice paprsku v čase se využívá vzorec 2.1 uvedený níže.

[4] Vysílání sekundárních paprsků pro lom, odraz a tzv. stínový paprsek.
[5] Pomocí vhodného generátoru náhodných čísel.
[6] Běžně se užívá pojem Field of View (FOV).

$$L(t) = O + tv | t > 0 \tag{2.1}$$

kde O je pozice pozorovatele, odkud paprsek vyšel, t je čas, v je směr vrženého paprsku, který spočítáme vzorcem $v = D - O$, kde D je místo průniku paprsku s objektem.

Pokud nabývá parametr t záporné hodnoty, paprsek směřuje za pozorovatele (výsledek nebude pro pozorovatele viditelný), a proto není nutné tento paprsek sledovat. V opačném případě směřuje paprsek před pozorovatele. Dále je nutné zjistit, zda paprsek protne alespoň jeden objekt ve scéně. Vzhledem k tomu, že objekty mohou být složeny z různých typů primitiv (např. koule, trojúhelníky[7], polygony atd.), je potřeba implementovat několik variací testů průniku [5] i v obou *pathtracerech* této bakalářské práce.

Z paprsku, který protne jeden z objektů v prostoru, se vytvoří nový paprsek, jehož počáteční pozice se nachází v bodě průniku a směr paprsku se určí podle toho, o který typ paprsku se jedná. Zde hrají velkou roli vlastnosti materiálů a jejich míra pravděpodobnosti. Pokud dojde k výpočtu emitovaného světla (namísto odražení a lomení paprsku), rozdělí se výpočet na dvě dílčí složky: přímé a nepřímé osvětlení.

Princip přímého osvětlení spočívá ve vyslání tzv. stínového paprsku (v originálu označovaného jako tzv.*shadow ray*), který směřuje přímo ke světelnému zdroji. Pokud paprsek při cestě neprotne žádný objekt, je bod průniku osvětlen světelným zdrojem a získá tak větší množství světla. Přímé osvětlení je třeba spočítat pro všechny světelné zdroje ve scéně a výsledky se následně sčítají.

Výpočtu nepřímého osvětlení se věnuje zobrazovací rovnice v následující sekci 2.5. Výsledky přímého a nepřímého osvětlení jsou nakonec sečteny a dohromady tvoří výsledné množství světla v místě prvního průniku.

2.5 Zobrazovací rovnice

V originálu *Render equation* je integrální rovnice na výpočet množství světla, které bod průniku paprsku s objektem vyzařuje směrem k pozorovateli. Tuto rovnici [7] popsal James Kajiya v roce 1986. Rovnice se skládá ze součtu množství světla, které je vyzařováno z bodu, a rovnicí BRDF na výpočet odrazivosti světla použitého materiálu. Pro výpočet rovnice 2.2 je vhodné použít techniku rekurze.

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_\omega f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t)(\omega_i * n) d\omega_i \tag{2.2}$$

kde λ je vlnová délka světla, t je čas, x je pozice v prostoru, ω_o je směr odchozího světla, ω_i je směr příchozího světla, L_o je celkový součet získaného světla, L_e je vyzařované světlo, L_i je příchozí světlo a f_r představuje BRDF rovnici.

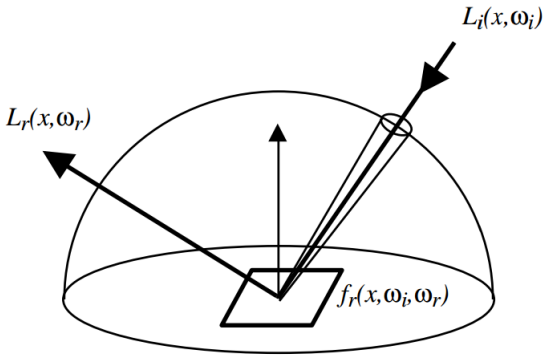
[7] Princip testu průniku paprsku s trojúhelníkem je popsán na webové stránce http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/why-are-triangles-useful

2.6 BRDF

Bidirectional Reflectance Distribution Function reprezentuje způsob, jakým lze vyjádřit světelné vlastnosti povrchu objektu. Tato funkce [10] byla definována Fredem Nicodemusem v roce 1965. Vzorec 2.3 vyjadřuje vztah mezi odraženým a dopadajícím světlem. Tento vztah je znázorněn na obrázku 2.2.

$$f_r(\omega_i, \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)cos(\theta_i \mathrm{d}\omega_i)} \tag{2.3}$$

kde L_r je intenzita odraženého světla, L_i je intenzita dopadajícího světla, ω_i je směr příchozího světla, ω_r je směr odraženého světla a θ_i je úhel mezi normálou povrchu a dopadajícím světlem.

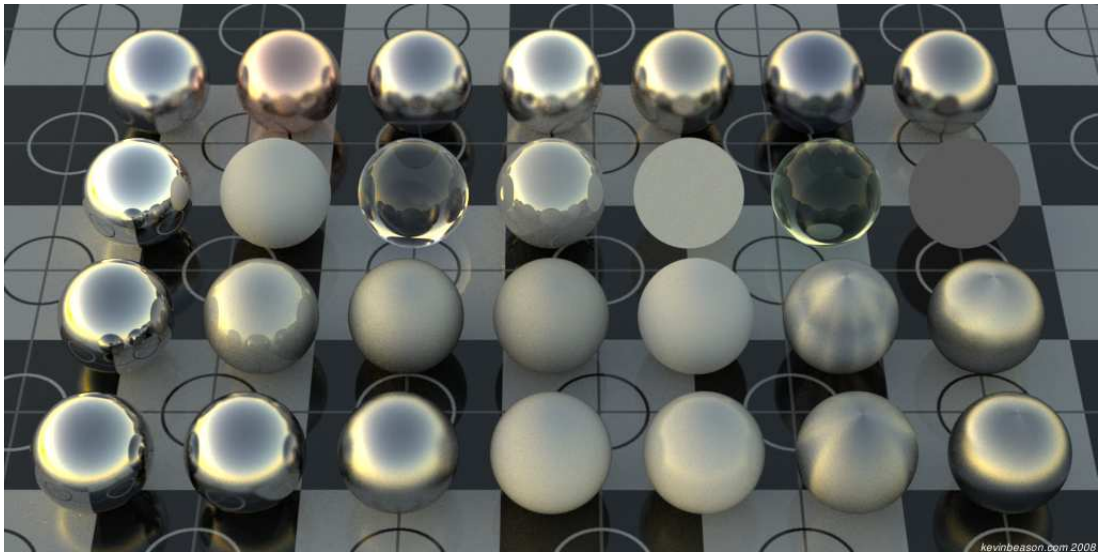


Obrázek 2.2: Množství příchozího a odchozího světla. Převzato z webu[8].

Výpočet rovnice BRDF je možné rozdělit do dvou skupin: izotropní a anizotropní. V první jmenované skupině se jedná o způsob reprezentace odrazových vlastností materiálu. V tomto případě se nebere v potaz rotace ploch okolo normály bodu. Pokud tedy dojde k rotaci ploch okolo normály, výsledná hodnota BRDF bude stále stejná. Izotropním BRDF se vyznačují materiály podobné hladkým plastům. Anizotropní BRDF naproti tomu reaguje na rotaci ploch okolo normály. Mezi materiály s anizotropním BRDF řadíme např. kartáčované kovy, satén nebo vlasy. Na obrázku 2.3 lze vidět některé ze zmíněných materiálů.

Další důležitou vlastností BRDF je její vzájemnost. Ta definuje, že pokud se směr cesty světla obrátí, výsledek zůstane stále stejný. Tento jev je pro *raytracing* (stejně jako pro *pathtracing*) klíčový, protože světelné paprsky jsou obvykle sledovány směrem od pozorovatele[10].

[8] Zdroj: http://cs.brown.edu/courses/cs224/papers/mc_pathtracing.pdf
[9] Zdroj: http://www.kevinbeason.com/worklog/wp-content/uploads/2009/03/brdftestdusk.png
[10] Nikoliv od zdroje světla, jak by mělo správně docházet při výpočtu globálního osvětlení.



Obrázek 2.3: Ukázka BRDF na různých typech materiálů. Převzato z webu[9].

2.7 Russian roulette

Vzhledem k tomu, že počet cest vyslaného paprsku může být potenciálně nekonečný, je nutné určit maximální délku výpočtu. Tuto délku lze zvolit explicitně nebo pomocí techniky zvané *Russian roulette*. Tato technika [3] definuje nenulovou pravděpodobnost k dosažení cílového bodu (bodu průniku) vyslaným paprskem. V opačném případě se zamezí vyslání paprsku (rekurzi algoritmu) a výpočet se ukončí.

Paprsky s kratšími cestami[11] mají větší pravděpodobnost dosažení cílového bodu než paprsky s delšími cestami. Pravděpodobnost ukončení *pathtracingu* vypočteme pomocí vzorce $p = 1 - f_r$ (výpočet f_r je popsán rovnicí 2.3), kde f_r značí odrazivost materiálu v bodu průniku paprsku s objektem a energie získaná ze zbývajících cest je škálovaná poměrem $1/p$. U tmavějších bodů je z důvodu horšího poměru získané informace a výsledné kvality obrazu snížena pravděpodobnost vyslání dalších paprsků. *Russian roulette* tak poskytuje adaptivní způsob ukončení výpočtu metody *pathtracing* s určitou mírou náhodnosti.

[11] Vzdálenost mezi bodem průniku a místem vyslání paprsku.

Kapitola 3

Akcelerační datové struktury

Pro využití metody *pathtracing* v reálných aplikacích je kritická základní rychlost implementace[12]. Jak již bylo řečeno v předchozích kapitole 2, metody založené na *raytracingu* musí provádět mnohačetné testy průniku s objekty ve scéně. Test průniku paprsku, který nezasáhne žádný z objektů scény, je pouze plýtvání výpočetního času počítače. Ten je zákonitě roven lineární časové náročnosti vzhledem k celkovému počtu primitiv ve scéně.

Tento klíčový problém je možné částečně řešit pomocí tzv. sdružování objektů do speciálních datových struktur (podobných binárním stromům), jejichž testování na průnik je mnohem výhodnější[13]. Po implementaci tohoto řešení nemusí vyslaný paprsek testovat všechny primitiva objektů ve scéně, ale pouze ty, které se nacházejí v oblastech, kterými paprsek projde. Ze všech protnutých oblastí se začíná nejdříve testovat obsah té, která je nejblíže pozici pozorovatele (resp. místo odkud byl paprsek vržen). Následující sekce popisují několik typů akceleračních datových struktur. Výsledky testování jednotlivých akceleračních datových struktur budou rozebrány v kapitole 7.

3.1 Bounding volume hierarchy

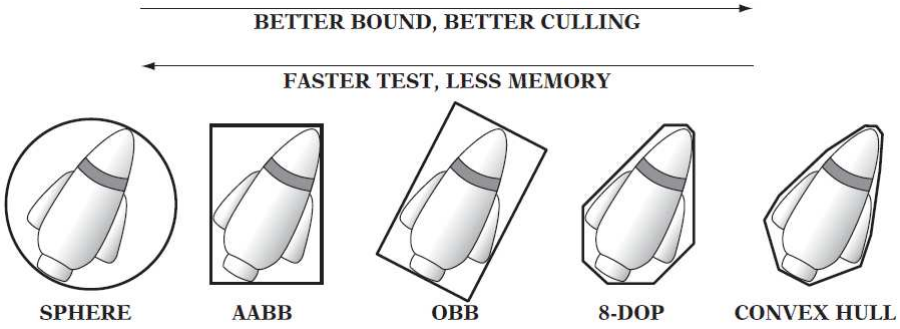
Bounding volume hierarchy [5] (nebo-li zkráceně BVH) tvoří stromovou strukturu sdružující objekty v tzv. *bounding volumes*. *Bounding volumes* jednoduchým způsobem umožňují zapouzdřit detailnější objekty (obsahující více primitiv/polygonů) do struktury, která se snáze testuje na průnik[14]. Rozměry jsou určovány dle maximálních rozměrů jednotlivých os vybraných objektů (osy X,Y a Z) v trojrozměrném prostoru. Způsob ohraničení se může lišit podle tvaru struktury (viz. obrázek 3.1).

Výběr je řízen poměrem mezi rychlostí testování s menšími nároky na paměť a lepšímu ohraničení objektu vedoucí k přesnějšímu vyhodnocování pravděpodobnosti průniku paprsku s objektem. Režie spojená s vytvářením struktury *bounding volume* je oproti testování polygonů všech objektů minimální.

[12] Také označovaná jako naivní.

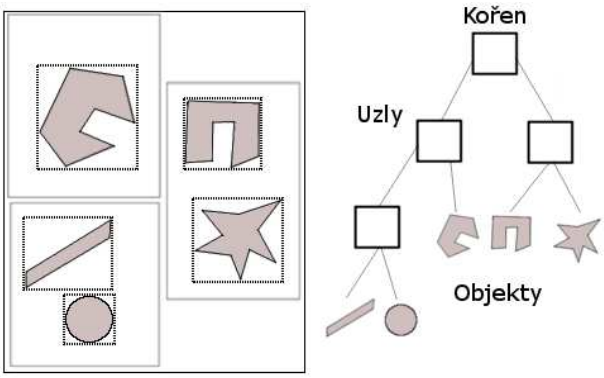
[13] Tato výhoda se projevuje s větším počtem primitiv objektů ve scéně.

[14] Princip průniku paprsku s AABB je vysvětlen na webové stránce https://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm



Obrázek 3.1: Různé typy *bounding volumes* [5].

Kořen BVH stromu je tvořen dvěma uzly, na které se navazují další uzly nebo koncové *bounding volume* struktury (tvořící listy uzlu). Pro zjištění průniku paprsku jsou rekurzivně testovány všechny uzly navázané v kořeni stromu. Pokud se tato operace úspěšně dostane až k listu stromu (k *bounding volume*), je otestován obsah této struktury na průnik paprsku s primitivy. Na obrázku 3.2 je znázorněn příklad stromové hierarchie.



Obrázek 3.2: Stromové zapouzdření objektů do *bounding volumes*.

Tento přístup dokáže ušetřit mnoho výpočetního času, protože není třeba testovat všechny primitiva objektů ve scéně, ale pouze ty, které mohou být reálně protnuty paprskem. Časová složitost tohoto řešení je v nejhorším případě logaritmická.

Důležitým aspektem při vytváření BVH struktury je její časová náročnost a volba správné heuristiky pro dělení objektů do větví BVH stromu. Použití nevhodné heuristiky může vést k horší vyváženosti BVH stromu a nižší výkonnosti struktury. Bohužel neexistuje univerzální heuristika pro všechny typy renderovaných scén, a proto je vhodná implementace funkce pro výpočet heuristiky na základě zobrazované scény (jako v případě [4]).

3.2 Octree

Datová struktura *Octree* [5] je částečně podobná BVH stromu. Každý uzel (včetně kořene) stromu se skládá přesně z 8 potomků (kombinace uzlů a listů). K dělení prostoru dochází do té doby, dokud není dosaženo maximální hloubky vytváření *octree* uzlů nebo definované hranice, kolik objektů ze scény může být umístěno do jednoho ohraničujícího *octree* uzlu. Každé nové dělení zajišťuje, že objekty nacházející se uvnitř uzlu budou umístěny minimálně do jednoho z 8 potomků tohoto uzlu. Rozměry každého nového uzlu jsou poloviční oproti rozměrům rodičovského uzlu. Výjimku tvoří kořen *octree* stromu, jehož rozměry jsou vždy minimální, ale zároveň dostatečné pro to, aby obsáhly všechny objekty nacházející se ve scéně.

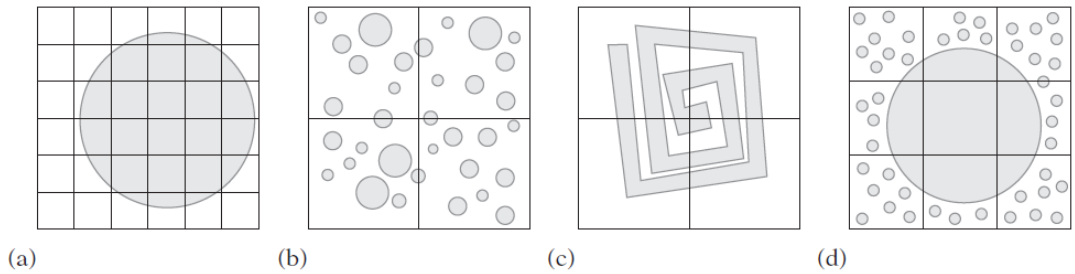
Použití datové struktury *octree* může v některých případech vést k problému, kdy se většina objektů ve scéně nachází na menším prostoru scény a v kombinaci s nižší maximální hloubkou vytvářené *Octree* struktury zůstanou objekty v jediném uzlu. Pokud by vyslaný paprsek při cestě narazil na tento uzel, musel by otestovat všechny objekty, které byly do uzlu vloženy. V tomto případě by výsledné zrychlení nemuselo být tak výrazné oproti jiným akceleračním datovým strukturám. Naproti tomu lze zlepšit výkonnost jednoduchým uchováním informace o tom, který objekt již byl v průběhu výpočtu testován. Stejně jako u ostatních akceleračních datových struktur, i zde je třeba počítat s operacemi na aktualizaci struktury při použití v dynamické scéně. Dále je také třeba provádět vyvažování celého stromu, které by mělo zachovávat vyšší výkonnost při testech průniku. Ve 2D prostoru existuje obdoba *octree* nazývaná *quadtree*, která má pouze čtyři potomky a typicky se používá k akceleraci renderování ve 2D prostoru.

3.3 Uniform grid

Tato metoda zapouzdřuje celou scénu do pravidelné mřížky, ve které je každý objekt asociován s určitým počtem voxelů[15] nazývaných (*grid cells*), v nichž se nachází. Díky pravidelnosti každé buňky a snadné přístupnosti k ní[16] je tato metoda velmi efektivní pro výpočet testu průniku. Vyslaný paprsek se pohybuje pomocí 3D DDA algoritmu [14] a po každém posunutí v mřížce se otestují všechny objekty, které se v dané buňce nachází.

Avšak i zde se vyskytuje několik případů, kdy se zvyšuje režie spojená s vytvářením a aktualizací indexů zaznamenaných objektů, a tím se snižuje výkonnost výpočtu. V prvním případě se jedná o čas strávený při inicializaci proměnných uniformní mřížky s vyšším počtem buněk a objektů s většími rozměry, které padnou do většího množství těchto buněk. S tím je také spojeno rozhodnutí, jaké rozměry voxelů zvolit pro dosažení optimálního poměru přesnost/výkon. Podle [6] uniformní mřížky vytvořené s větším množstvím voxelů vykazují výborné výsledky v testech SPD[17] scén. Na obrázku 3.3 lze vidět případy, kdy není použití uniformní mřížky optimální.

[15] Jedná se v podstatě o 3D pixel.
[16] Index lze získat podílem rozměrů *World coordinates* a šířkou jedné buňky uniformní mřížky.
[17] *Standard Procedural Databases*.



Obrázek 3.3: Problémy spojené s rozlišením mřížky [5]. (a) Mřížka je příliš jemná. (b) Mřížka je příliš hrubá. (c) Mřížka je příliš hrubá, vzhledem ke složitosti zobrazovaného objektu. (d) Mřížka je hrubá i jemná vzhledem k objektům ve scéně.

Pokud je naše scéna dynamická (objekty v ní mění svoji pozici), je třeba počítat s aktualizací indexů rozmístěných objektů v uniformní mřížce. Tato operace může být časově náročná při použití vyššího rozlišení mřížky. V některých případech je z hlediska výkonu lepší vytvořit novou mřížku než aktualizovat stávající. Stejně jako u *octree* i zde je vhodné použít značení objektů, které již byly testovány v průběhu výpočtu na průnik.

Kapitola 4

Existující řešení

V této kapitole budou představeny již existující implementace algoritmu *pathtracing* v komerčních i nekomerčních aplikacích. Tyto aplikace obvykle podporují mnohem více technologií, a proto je třeba brát následující informace jako pouze orientační.

4.1 Nvidia Iray

Nvidia Iray je robustní C++ API na programování fotorealistických scén. Je také dostupný jako renderovací engine do mnoha aplikací[18] pro tvorbu 3D obsahu. Dále nabízí nepřeberné množství grafických efektů a také nástroje pro souběžné zpracovávání většího počtu operací. K tomu je možné využít i nasazení většího počtu GPU od firmy Nvidia.

Iray rozlišuje tři módy renderování: fotorealistické, interaktivní a renderování v reálném čase. Každý z uvedených módů si lze upravit podle konkrétních požadavků uživatele. Fotorealistický mód přináší veškeré efekty, kterými Iray disponuje pro věrnější zobrazení scény. Interaktivní mód klade důraz na maximální výkon při zachování pokročilých technik napodobujících realitu. Renderování v reálném čase pak již používá standardní rasterizační techniky pro renderování v reálném čase. Avšak tento mód nedosahuje přesnosti předchozích dvou módů v simulaci globálního osvětlení. Zobrazení využívá API OpenGL ve verzi 3.3 včetně několika rozšíření od Nvidie.

4.2 Furryball

Furryball (aktuálně ve verzi 4.8) je nástroj, který se zaměřuje na renderování fotorealistických scén pomocí GPU (jsou podporovány pouze GPU s podporou technologie CUDA). Software je dostupný jako plugin pro grafické nástroje Maya, 3DS Max a Cinema 4D. Mimo implementaci *pathtracingu* podporuje také efekty jako jsou např. *Motion blur*, *Multisampling*, *Ambient Oclusion* a mnohé další. Uživatel má také možnost přepínat mezi dvěma způsoby renderování (rasterizace vs *raytracing*). Na oficiálních stránkách výrobce jsou zveřejněny testy na různých testovacích sestavách.

Vlastní testování neproběhlo z důvodu absence grafické karty podporující technologii CUDA. V tabulce níže je uvedeno několik testovacích sestav a jejich výkon v renderování pomocí rasterizace a *raytracingu*. Převzato z webové stránky výrobce[19]. Výsledky jsou spíše

[18] Např. 3ds Max, Cinema 4D nebo Autodesk Maya.

[19] Zdroj: http://www.aaa-studio.cz/furrybench/benchResults4.php

orientační, ale bližším prozkoumáním tabulky 4.1 lze vypozorovat, že doba renderování jednoho snímku pomocí rasterizace a *raytracingu* se pohybuje přibližně v poměru 1:7.

CPU	RAM	GPU	Rasterization	Raytracing
Intel Core i7-5960X @ 3.00GHz (16 CPUs)	32GB	NVIDIA GeForce GTX 980 (4GB)	3.78 s	40.81 s
Intel Core i7-2600K @ 3.40GHz (8 CPUs)	16GB	NVIDIA GeForce GTX 580 (1.5GB)	10.24 s	62.72 s
AMD Phenom II X4 955 @ 3.20GHz (4 CPUs)	8GB	NVIDIA GeForce GTX 750 Ti (2GB)	13.92 s	92.77 s
Intel Core i5-4200M @ 2.50GHz (2 CPUs)	4GB	NVIDIA GeForce GT 750M (2GB)	33.36 s	230.57 s

Tabulka 4.1: Porovnání výkonu renderování mezi rasterizací a *raytracingem*. Konfigurace testovaných počítačů běží na 64-bitových systémech Windows 7/8.

4.3 Octane Render

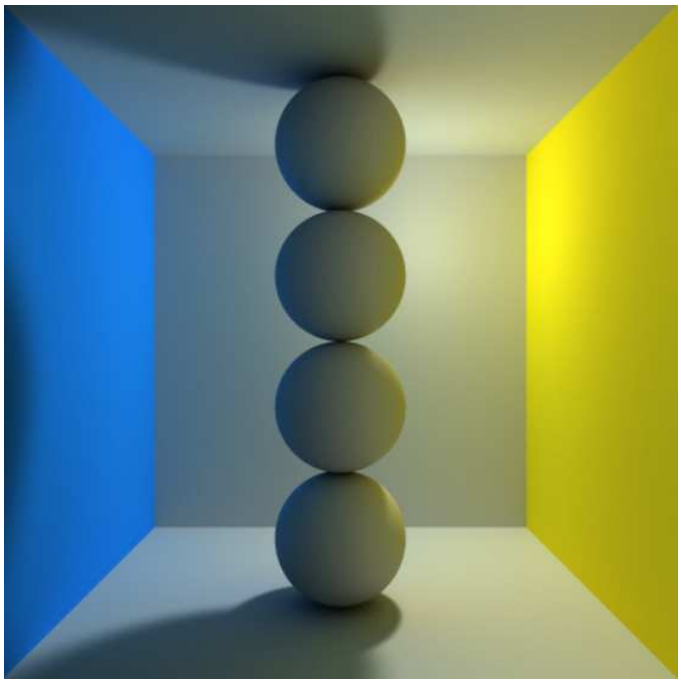
Dalším komerčním produktem je Octane Render. Ten je schopen v reálném čase vykreslovat scény pomocí *pathtracingu* zpracovávaném pouze na jednotkách GPU[20] s technologii CUDA. Octane Render je označován jako *unbiased renderer*. Slovo unbiased značí, že technika renderování by neměla generovat systematické chyby nebo *bias*[21] při výpočtu barvy pixelu. Mimo tyto vlastnosti jsou k dispozici také nástroje pro správu textur, světel nebo renderování přes síť[22]. Renderer lze taktéž použít jako plugin do stávajících 3D editorů (např. Blender, CINEMA 4D nebo 3ds Max). Obrázek 4.1 ukazuje kvalitu v detailním zobrazení modelu.

Obrázek 4.1: Snímek vygenerovaná Octane Render. Převzato z webové stránky autora[23].

[20] Podporuje výpočet na větším počtu grafických karet.
[21] Rozdíl mezi předpokládanou hodnotou a skutečnou hodnotou.
[22] Renderování na vzdálených počítačích.
[23] Zdroj: http://render.otoy.com/gallery2.php

4.4 WebGL Path Tracing

WebGL Path Tracing patří mezi projekty programátora Evana Wallace. Z názvu lze odvodit, že tento *pathtracer* využívá pro výpočet API WebGL. Zbytek aplikace je implementována v jazyku Javascript. Zobrazovaná scéna je dynamicky překládaná do GLSL *shaderu*, ve kterém probíhá výpočet. Pro dostatečně věrné zobrazení je nastavena maximální hloubka sledování paprsku na hodnotu 5. Aplikace umožňuje uživateli interakci se zobrazovanými prvky, kam patří jednotlivá primitiva nebo světlo. Na výběr je několik připravených scén, ale lze libovolně přidávat i další objekty a přesouvat je v prostoru. Dále si může uživatel zvolit z několika variant materiálů objektů (matný, lesklý, zrcadlový) a barev okolních stěn. Na obrázku 4.2 je zobrazena jedna z připravených variant scén.



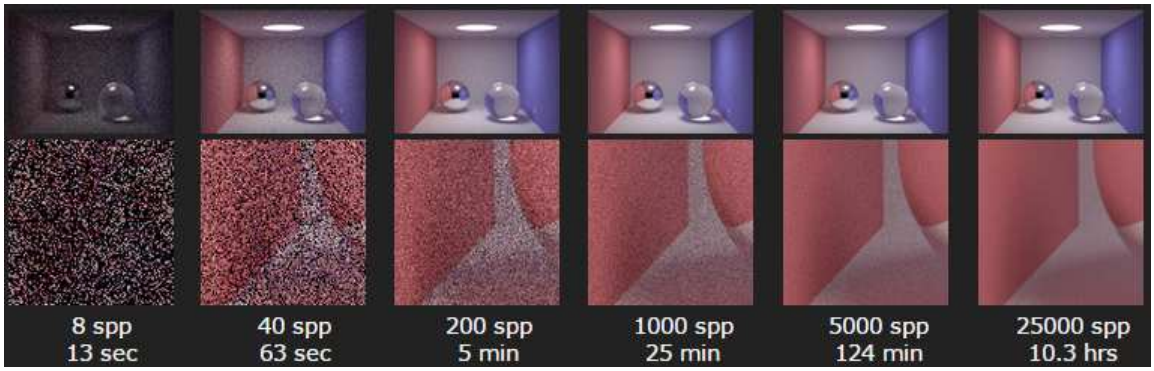
Obrázek 4.2: Snímek z WebGL Path Tracing. Převzato z webové stránky autora[24].

Aplikaci si lze vyzkoušet na webové stránce autora[24]. Zdrojové kódy aplikace jsou zveřejněny na webové stránce služby GitHub uživatele evanw[25] pod licencí MIT. Podmínkou pro spuštění výpočtu je podpora rozšíření `OES_texture_float`.

4.5 Smallpt

Smallpt patří mezi zástupce nekomerčních aplikací implementující Monte Carlo *pathtracing* s algoritmem *Russian roulette*. Dále podporuje vícevláknové zpracování s využitím knihovny OpenMP, měkké stíny nebo *antialising* pomocí techniky *super-sampling*. Výpočet algoritmu *pathtracing* probíhá pouze na CPU. Na obrázku 4.3 je zobrazena vizuální kvalita snímků po jednotlivých časových intervalech.

[24] Zdroj: http://madebyevan.com/webgl-path-tracing/
[25] Zdroj: https://github.com/evanw/webgl-path-tracing



Obrázek 4.3: Náhled na detail snímku ze Smallpt. spp(samples per pixel) značí počet vzorků na pixel. Převzato z webové stránky autora[26].

Celá implementace je napsána do 99 řádků kódu. Malý rozsah zdrojového kódu vyvažuje nízká míra optimalizace a výpočet tak trvá delší dobu. Projekt poskytuje zdrojové kódy napsané v jazyce C++ pod licencí MIT, která je otevřená a kompatibilní s GPL[27].

4.5.1 SmallptGPU2

Díky otevřenosti zdrojových kódu vznikají mnohé modifikace, mezi které patří i SmallptGPU2[28]. SmallptGPU2 převádí implementaci Smallpt do jazyka OpenCL pro výpočet na CPU a GPU současně. Aplikace dovoluje nastavit, v jakém poměru bude zpracování mezi oběma výpočetníma jednotkami. Nástroj byl otestován na následujících konfiguracích testovaných počítačů uvedených v tabulce 4.2.

#	OS	CPU	RAM	GPU
1	Windows 8.1 64-bit	Intel Core i5-2410M @ 2.30GHz (4 cores)	6GB	AMD Radeon HD 6490M
2	Windows 8.1 64-bit	Intel Pentium 2117U @ 1.80GHz (2 cores)	4GB	Intel HD Graphics (GT1)

Tabulka 4.2: Specifikace testovacích počítačů

Výkon byl srovnáván na obou počítačích ve třech módech nastavení aplikace.

1. Výpočet probíhá pouze na CPU.
2. Výpočet probíhá na CPU a GPU zároveň v poměru 1:1.
3. Výpočet probíhá pouze na GPU.

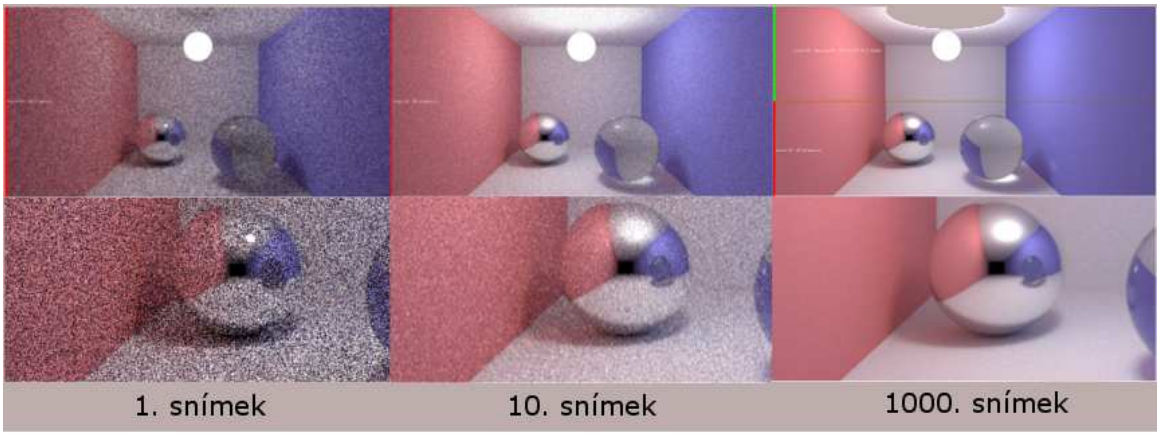
[26] Zdroj: http://www.kevinbeason.com/smallpt/#moreinfo
[27] *General Public License.*
[28] Aplikaci lze stáhnout na webových stránkách: http://davibu.interfree.it/opencl/smallptgpu2/smallptGPU2.html

4.5.2 Výsledky testování

Testování probíhalo pomocí experimentálního měření na zmíněných sestavách. V tabulkách 4.3 a 4.4 jsou uvedeny výsledky testování na první a druhé sestavě a na obrázku 4.4 lze vidět získanou kvalitu pro daný počet vzorků na pixel.

#	Konfigurace	Jeden snímek	10 snímků	30 snímků	100 snímků	300 snímků	1000 snímků
1	Pouze CPU	0.940s	10.661s	34.955s	1:54.499s	5:30.157s	17:14.295s
1	CPU a GPU	0.453s	4.458s	13.960s	47.199s	2:20.341s	7:46.451s
1	Pouze GPU	0.640s	5.802s	19.200s	1:5.853s	3:19.645s	11:8.316s

Tabulka 4.3: Výsledky získané z testování na počítači číslo 1.



Obrázek 4.4: Porovnání mezi kvalitou jednotlivých snímků.

#	Konfigurace	Jeden snímek	10 snímků	30 snímků	100 snímků	300 snímků	1000 snímků
2	Pouze CPU	1.580s	14.485s	47.114s	2:40.260s	8:20.339s	27:46.609s
2	CPU a GPU	0.770s	7.120s	24.250s	1:21.761s	4:12.507s	13:43.708s
2	Pouze GPU	0.470s	4.177	14.262s	48.922s	2:25.688s	8:03.227s

Tabulka 4.4: Výsledky získané z testování na počítači číslo 2.

Kapitola 5

Použité technologie

V této kapitole jsou popsány všechny nástroje a jejich vlastnosti využité při implementaci výsledné aplikace. Převážná část aplikace bude vytvořena v programovacím jazyce C/C++, zajišťující výhodný poměr výkon/abstrakce. Pro vytvoření grafického kontextu bude použito API OpenGL (ve verzi 3.0+ s dopřednou kompatibilitou).

Vzhledem k tomu, že samotné OpenGL nedisponuje nástroji pro vytvoření okna aplikace, je pro tento účel použito aplikační prostředí WinAPI. Toto API bude dále zprostředkovávat také uživatelské vstupy. K implementaci algoritmu *pathtracing* na GPU bude využita knihovna OpenCL (ve verzi 1.2), která je nejčastěji podporovaná v dnešních běžných grafických akcelerátorech.

Vzhledem k implementaci CPU i GPU *pathtracingu* a doprovodné funkčnosti (např. tlačítko pauzy nebo save) je potřeba synchronizovat některé operace před tím, než dojde k přepnutí výpočtu na druhý renderer. Tuto funkčnost zajišťují semafory z knihovny SDL, které jsou velice jednoduché a intuitivní na použití.

5.1 WinAPI

WinAPI (nebo-li Windows API) je aplikační prostředí[29] poskytující vývojáři nízkoúrovňový přístup k sadě služeb dostupných pod systémem Microsoft Windows. Tyto služby mimo jiné zajišťují odchytávání událostí uživatelského vstupu (např. myš, klávesnice ad.) a tvorbu grafického uživatelského prostředí. Pokročilejší tvorbu grafického obsahu pak přenechává na aplikačních prostředích jako je DirectX nebo OpenGL.

WinAPI je primárně určeno pro využití v jazyku C/C++, ale existují také implementace pro použití v jazyku Java. Použitelné alternativy k WinAPI jsou knihovny SDL (Simple Direct Media), GLFW nebo GLUT (resp. freeGLUT), které nabízejí jednodušší způsob použití. WinAPI je použito k vytvoření aplikačního okna a zajištění obsluhy interakce uživatele s aplikací. Rozhodnutí vybrat toto aplikační prostředí vychází z důvodu bezproblémové podpory interoperability mezi knihovnami OpenGL a OpenCL (dále bude diskutováno v sekci 6.4), která se u ostatních knihoven nepodařila zajistit bez obtíží.

[29] WinAPI existuje již z dob Microsoft Windows 95.

5.2 Knihovna OpenGL

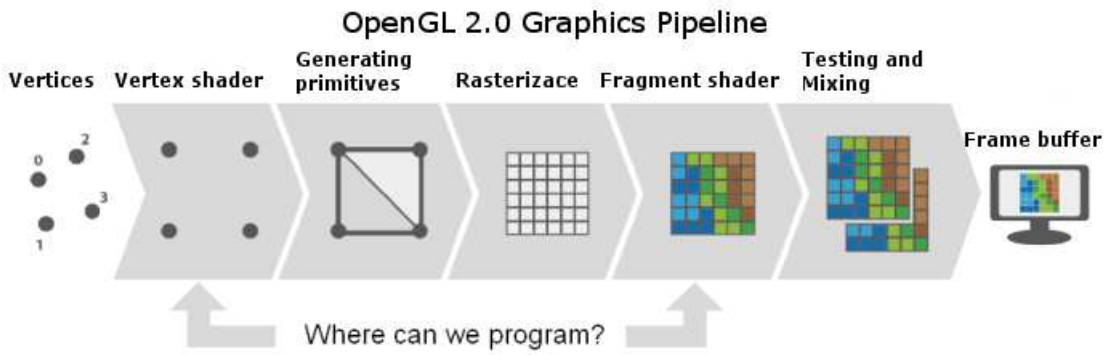
Knihovna OpenGL (nebo-li *Open Graphics Library*) [12] je popisována jako nízkoúrovňové grafické API, pomocí kterého lze vytvářet dvou (2D) nebo třírozměrnou (3D) grafiku. Ta je akcelerována pomocí grafické karty (výrobce musí toto API podporovat). Dnes se používá převážně v tvorbě grafických aplikací (např. GIMP, Adobe Photoshop), počítačových her (Doom 3, Angry Birds) nebo v různých oblastech inženýrství (AutoCAD, architektura).

OpenGL lze využít v mnoha běžných programovacích jazycích, jako je např. C/C++, Java (knihovna JOGL) a dalších, na mnoha běžně používaných platformách, např. Windows, Linux nebo MacOSX. Také je toto API (v podobě OpenGL ES) oblíbené při tvorbě pokročilé grafiky na mobilních platformách nebo pro tvorbu 3D grafiky na internetu (v případě WebGL).

V této bakalářské práci je OpenGL využito jako zobrazovací prostředek výsledků výpočtu metody *pathtracing* a jednoduchého grafického uživatelského prostředí pro ovládání výpočtu. Také jsou v aplikaci zobrazovány dodatečné informace pro okamžitou uživatelskou kontrolu (např. aktuální počet snímků nebo čas výpočtu pro zvolenou akcelerační datovou strukturu a renderer). Další využité vlastnosti OpenGL budou rozebrány v kapitole 6.1 popisující implementaci zobrazení.

5.3 Shadery

Shadery byly navrženy k tomu, aby zpracovávaly relativně malou množinu grafických atributů scény [2]. Mezi tyto grafické atributy lze zařadit barvy, pozice, normály nebo matice jako vstupní data. *Shadery* si lze představit jako programovatelné části grafické *pipeline* (viz. obrázek 5.1). Možnost programování vlastních shaderů přišla s API DirectX 8 a OpenGL 2.0. Od té doby se hojně využívají především ve videoherním průmyslu, kde zajišťují tvorbu nespočet grafických efektů.



Obrázek 5.1: Grafická *pipeline* OpenGL 2.0. Převzato z webové stránky[30].

Shadery se dělí na několik typů:

Vertex Shader je první část *pipeline*, která obvykle pracuje s jednotlivými vertexy vykreslovaných primitiv. Obvykle se zde upravují pozice vertexů, které nemění originální data, která jsou do *shaderu* vkládána. Tato vlastnost zrychluje generování pokročilých efektů simulujících nerovnosti vykreslovaného objektu nebo jednoduché animace.

[30] Zdroj: http://www.rastergrid.com/blog/wp-content/uploads/2010/09/sm50_tess.png

Geometry shader se vyskytuje v grafické pipeline OpenGL od verze 3.2. Účel tohoto *shaderu* spočívá v možnosti generovat nové vertexy na základě již existujících a měnit tak vzhled výsledného objektu.

Fragment shader se zabývá zpracováváním barevné složky vertexů. Na každý vertex lze „natáhnout“ zadanou barvu nebo určit, která část použité textury se zde bude vykreslovat. Pomocí *fragment shaderu* lze tak tvořit nejrůznější filtry, mixování více textur a mnohé další.

Compute shader je „novinka“, která podobně jako OpenCL nebo CUDA slouží k všeobecným výpočtům, které lze paralelizovat. Tento *shader* se objevil v OpenGL až od verze 4.3, a proto není zdaleka tak využitelný u většiny uživatelů vlastnících grafické karty starší více než 3 roky.

Jednotlivé *shadery* je možné implementovat přímo v knihovně OpenGL (jazyk GLSL) nebo DirectX (jazyk HLSL). Jazyky GLSL/HLSL se používají častěji pro dosažení akcelerované grafiky a grafických efektů, než k obecným výpočtům. Ze všech zmíněných *shaderů* by se dal pro účely implementace *pathtracingu* využít *compute shader*, ale ten nemusí být prozatím dostupný na mnoha běžných systémech. Z tohoto důvodu bude v práci využita knihovna OpenCL, která existuje delší dobu a je tak více rozšířená než *compute shader*. Pro zobrazení obsahu okna budou použity *vertex* a *fragment shadery*.

5.4 Knihovna OpenCL

OpenCL [9] je průmyslová standardní knihovna na programování aplikací, které jsou zpracovávány v kombinaci hlavního procesoru (CPU) a procesorů grafické karty (GPU). Tato knihovna je stále relativně mladá (první revize vznikla v roce 2008), ale i tak ji lze použít k efektivnímu využití potenciálu specializovaných čipů (v případě této práce výpočetních jader grafické karty). Programový kód napsaný v OpenCL je nezávislý na použitém hardwaru, a dá se tak jednoduše upravovat a používat na široké škále čipů (CPU, GPU, DSP, FPGA), které standard OpenCL podporují. Základ tohoto jazyka vychází ze standardu C99 jazyka C rozšířeného o speciální datové struktury včetně příslušných metod s několika konstrukčními omezeními.

Nové datové struktury a operace nad nimi jsou určené pro využití v paralelním zpracování (jako jsou třeba vektory, buffery nebo 1/2/3D obrazové typy). OpenCL poskytuje také nástroje pro synchronizaci vláken a atomicitu operací v případě, kdy vlákno potřebuje pracovat s daty získanými v průběhu výpočtu jiného vlákna.

Omezení, které přináší nezávislost jazyka OpenCL na použitém čipu, bylo nutné zavést z důvodu potenciálně odlišného chování hardwarové implementace některých operací. V této práci se omezení týká především dynamické alokace paměti, využití rekurze (resp. správné použití zásobníku při rekurzi) nebo využití reference na objekt.

Operace by měli být programovány v jazyku OpenCL pouze tehdy, pokud jsou vykonávány v cyklech. Implementovaný algoritmus by měl obsahovat co nejmenší počet datových závislostí mezi vlákny výpočtu, aby nedocházelo ke zpomalování výpočtu ostatních vláken. V této práci je algoritmus *pathtracing* implementován v jazyku OpenCL (ve verzi 1.2) a výpočet probíhá paralelně pro každý pixel výsledného snímku. Detailnější popis výpočtu je uveden v podsekci 6.4.1.

Kapitola 6

Implementace

Tato kapitola se zaměřuje na popis vlastní implementace výsledné aplikace. Ta je určena pouze pro demonstraci výkonnosti jednotlivých akceleračních datových struktur v kombinaci s CPU/GPU pathtracerem. Aplikace je přizpůsobená ke spouštění na počítači s operačním systémem Microsoft Windows.

6.1 Zobrazení aplikace

Pro zobrazení okna aplikace jsem zvolil aplikační prostředí WinAPI[31]. Toto prostředí je roky prověřené a disponuje širokou škálou dostupných nástrojů. V aplikaci je WinAPI využité pouze pro vytvoření základního okna (v rozlišení 800x600) a zachytávání událostí vstupních zařízení, jako je myš nebo klávesové zkratky. Veškerý obsah okna je vykreslován pomocí „moderního“ OpenGL verze 3.0+ s dopřednou kompatibilitou. Inicializace OpenGL je prováděna skrz knihovnu p. Ing. Lukáše Poloka.

Vykreslování jednotlivých prvků aplikace (funkční tlačítka, animace stavu výpočtu apod.) je prováděno pomocí Vertex Buffer Object (zkráceně VBO) a Vertex Array Object (zkráceně VAO). V případě první jmenované datové struktury se jedná o buffer, do které jsou ukládány veškeré vertexy určené k vykreslení. Pokud bude potřeba s tímto bufferem dále pracovat, je nutné ho nejdříve nabindovat (pomocí funkce `glBindBuffer()`). Poté se lze na data v nabindovaném VBO odkazovat. Vertex Array Object je datová struktura, která sdružuje posloupnost OpenGL příkazů, které by bylo nutné jinak manuálně psát na všechna místa použití (dalo by se říci, že se jedná o jistý druh makra). S využitím VAO není nutné psát u každé vykreslovací rutiny všechny potřebné příkazy, ale jednoduše stačí nabindovat požadované VAO. Tento přístup dovoluje dynamické přepínání mezi vícero VAO struktur. V aplikaci je využito čtyř VBO a VAO.

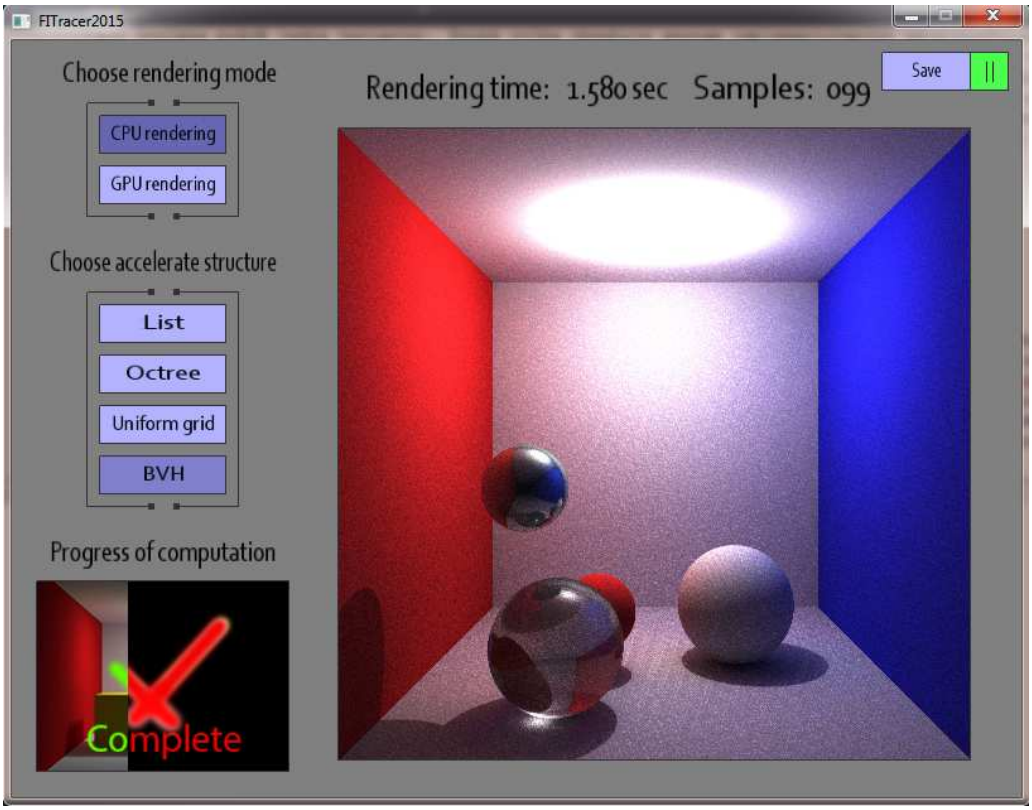
Po úspěšném vytvoření a inicializaci VAO a VBO struktur je možné začít vykreslovat. Pro vykreslení jsou použity funkce `glDrawElements` a `glDrawArrays`, kde první zmíněná funkce vyžaduje seznam prvků, které budou v daném pořadí vykreslovány. Tento přístup je ekonomický, neboť umožňuje využít stejné vertexy vícekrát. Naproti tomu `glDrawArrays` vykresluje každý vertex pouze jednou a v pořadí, v jakém jsou uloženy ve VBO. V aplikaci jsou využity jednoduché *vertex* a *fragment shadery*. Jak již bylo zmíněno v sekci 5.3, *vertex shader* je použit pro vykreslení jednotlivých vertexů z nabindovaného VBO. U *fragment*

[31] I přes snahu využít modernějších API pro tvorbu aplikačního okna doprovázející lepší provázání s knihovnou OpenGL (např. GLFW nebo SDL) se ukázalo být plně funkční pouze WinAPI, které jako jediné dokázalo bezproblémově spolupracovat s požadovanou funkcí OpenGL/CL interoperability.

shaderu se přepíná mezi vykreslením definované barvy (v případě ovládacích a dekoračních prvků aplikace) a vykreslování textury (v případě výsledných snímků výpočtu). Data určující barvu prvků v okně jsou nahrávány do tzv. uniformních proměnných, které jsou nastavovány v průběhu běžící aplikace na danou hodnotu. Vykreslování vypočtených snímků probíhá nad dvěma 2D texturama, kdy v jeden moment je jedna textura určena pouze pro vykreslování a druhá je uzamknutá pro využití ve výpočtu. Podrobnější popis přepínání textur je popsán v sekci 6.4. Operace nad OpenGL jsou implementovány ve třídě `SDLGLContext`, která obsahuje také informace o průběhu výpočtu a synchronizační prvky SDL semaforů.

6.2 Ovládání aplikace

Aplikace disponuje základní sestavou ovládacích prvků, kterými lze zvolit požadovanou konfiguraci výpočtu. První volba spočívá ve výběru rendereru. Uživatel má na výběr mezi výpočtem na CPU nebo na GPU (vždy musí být zvolena právě jedna varianta). Druhá volba obsahuje seznam implementovaných akceleračních datových struktur[32], a uživatel tak může volit, která z implementovaných akceleračních datových struktur se má použít. Po spuštění aplikace nejsou vybrány žádné volby a tlačítko pro spuštění výpočtu je zablokované, dokud uživatel nevybere počáteční konfiguraci. Na obrázku 6.1 je ukázka zobrazující jednotlivé prvky demonstrační aplikace.



Obrázek 6.1: Snímek z běžící aplikace.

[32]Volba LIST znamená výpočet bez akcelerační datové struktury.

Mezi jednotlivými konfiguracemi lze libovolně přepínat v průběhu výpočtu, ale změny se aplikují vždy až po dokončení výpočtu aktuálního snímku. Přepínat lze i pomocí klávesových zkratek, kde první písmeno funkčního prvku odpovídá přiřazené klávesové zkratce (např. písmeno 'o' aktivuje tlačítko Octree). Po spuštění aplikace bude uživatel dotázán v příkazové řádce, které z dostupných zařízení (podporující standard OpenCL) bude použito pro výpočet. V případě absence zařízení podporující standard OpenCL bude uživatel o dané situaci informován a aplikace se ukončí.

V případě CPU *pathtraceru* je uživatel informován o průběhu výpočtu animací v okně aplikace nebo procentuálním výpisem v příkazové řádce. Uživatel má také možnost výpočet pozastavit[33] a provést případně patřičné změny v konfiguraci výpočtu. Aktuální snímek si lze uložit do složky, kde se aplikace nachází[34] ve formátu PNG. Implementace ovládacích prvků se nachází ve třídách `GLObject`, `GLButton`, `GLText` a jejich obsluha v aplikaci je prováděna ve třídě `SDLGLContext`.

6.3 Scéna

Pro demonstraci výpočtu *pathtracingu* je připravena variace tzv. Cornell boxu, která se skládá z několika stěn a koulí. Vytvoření objektu koule zajišťuje třída `Sphere` a načítání 3D modelů třídy `Mesh` a `Triangle`. Mesh je datová struktura, pomocí které lze načítat složitější objekty složené z mnoha trojúhelníků. Aplikace podporuje načítání 3D modelů z formátu OBJ[35]. Pro lepší demonstraci vlastností *pathtracerů* se každá koule odlišuje použitým materiálem. Ve vlastnostech materiálu je možné nastavit průhlednost objektu, míru lomu příchozího světla (v našem případě příchozího paprsku), odrazivost povrchu objektu nebo základní barvu. Tyto vlastnosti zajišťuje třída `Material`. Při inicializaci scény jsou nejprve vytvořeny různé druhy materiálů a ty jsou použity pro vytvoření objektu. Aplikace podporuje načítání souborů ve formátu MTL, které obsahují informace o materiálu pro jednotlivé části daného 3D modelu. Dále jsou inicializovány entity zastupující roli světelných zdrojů pomocí třídy `Light`. Instanci třídy lze nastavit barvu a intenzitu vyzařovaného světla. Každý ze zmíněných prvků scény lze libovolně přesouvat ve scéně pomocí funkcí `setScale`, `setRotation` a `setTranslation`. Dále je každý objekt označen jedinečným indexem z důvodu rozlišení dat při exportu pro GPU *pathtracer*.

Při inicializaci CPU *pathtraceru* je scéna vytvořena pro každou akcelerační datovou strukturu znova. Díky tomu není nutné při každé změně konfigurace výpočtu rušit a opět vytvářet celou scénu (vzhledem k jednoduchosti zobrazované scény jsou potřebné paměťové nároky relativně nízké). Na obrázku 6.2 je zobrazen snímek složený ze 150 vzorků na pixel v rozlišení 500x500. Snímek obsahuje několik světelných zdrojů, které demonstrují různobarevnost zobrazených stínů a jejich vzájemné překrývání.

[33] CPU: pozastaví výpočet okamžitě, GPU: pozastaví výpočet po dokončení výpočtu aktuálního snímku.
[34] Název vygenerovaného souboru je FITracer2015output.png.
[35] Stěna Cornell boxu je načítána ze souboru plane.obj.

Obrázek 6.2: Výsledný snímek scény Cornell boxu.

6.4 Implementace pathtracerů

Implementace obou *pathtracerů* zachovává stejné vlastnosti výpočtu. Jedinou odlišností CPU a GPU *pathtraceru* je optimalizace Octree a Uniform grid struktur. CPU implementace obsahuje optimalizační techniku v podobě „paměti“, která si uchovává informaci, zda aktuálně sledovaný paprsek již daný objekt testoval. Tato „paměť“ si uchovává ID paprsku, které je unikátní pro každý nově vytvořený paprsek. GPU implementace obou struktur tuto vlastnost nepodporuje z důvodu vyšších nároků na lokální paměť výpočetních jader[36]. Použití paměti objektů v globální paměti výpočetních jader se negativně projevilo v časovém zpoždění při častém čtení z této paměti. Celkový čas výpočtu na GPU byl tak pomalejší než bez využití této techniky.

Oba renderery využívají pro zápis získaného snímku OpenGL 2D textury. Implementace GPU *pathtraceru* je napsána v jazyku OpenCL 1.2. Tato verze OpenCL nepodporuje režim čtení–zápis nad proměnou typu `image2d_t`, proto bylo třeba použít jednu texturu pro čtení a druhou pro zápis. Z tohoto důvodu je nutné přepínat mezi texturami po každém vypočteném snímku (toto přepínání připomíná techniku zvanou *ping pongping*). Pro využití OpenGL textury v OpenCL kernelu je potřeba OpenCL 2D texturu nejdříve vytvořit pomocí funkce `clCreateFromGLTexture()` a před spuštěním kernelu uzamknout texturu proti

[36]Tyto paměťové nároky jsou vysoké i při použití 3D modelů s menším počtem trojúhelníků (přibližně 3k trojúhelníků).

úpravám (`clEnqueueAcquireGLObjects()`). Po dokončení výpočtu je nutné opět texturu odemknout (`clEnqueueReleaseGLObjects()`), aby bylo možné s ní pracovat v OpenGL. Pro získání barvy pixelu z textury je nutné využít tzv. *sampleru* a s nastavením filtrovacího módu na nejbližší vzorek, který nezkresluje výsledný snímek. U CPU implementace je možné přistupovat přímo k pixelům textury přes index šířky a výšky.

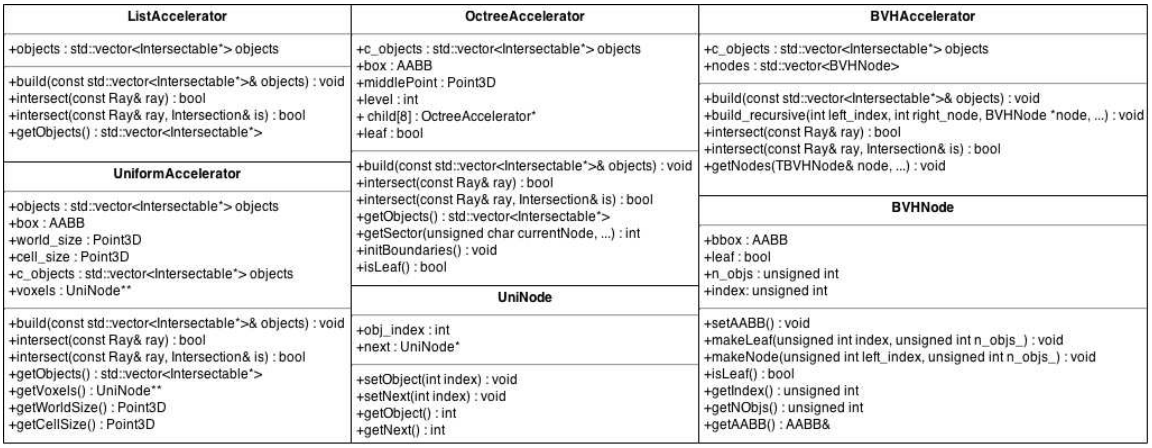
Po výpočtu každého snímku se indikuje dokončení výpočtu a prohození aktivní textury pro vykreslení v okně aplikace. Dále se zkontroluje zda uživatel aktivoval tlačítko pauzy (u GPU *pathtraceru*), změnil nastavení stávající konfigurace. Pokud ano, dojde k přepnutí aktivního kernelu na zvolenou volbu u GPU *pathtraceru* a nahrání dané akcelerační datové struktury do CPU *pathtraceru*. Pokud uživatel uzavřel okno aplikace, dojde k ukončení výpočtu aplikace se ukončí.

6.4.1 CPU pathtracer

Kostra implementace CPU *pathtraceru* vychází z materiálů kurzu EDAN30 Photorealistic Computer Graphics na Lund University[37] (použité zdrojové kódy jsou označeny komentářem v hlavičce souboru). Tuto základní implementaci jsem následně upravoval pro potřeby této bakalářské práce. Výsledná CPU implementace podporuje vykreslování *pathtracingu* ve čtyřech možných konfiguracích. Mezi ty se řadí všechny popsané akcelerační datové struktury[38] popsané v kapitole 3. Implementaci metody LIST zajišťuje třída `ListAccelerator`, která představuje výpočet průniku paprsku s všemi objekty ve scéně. Tato metoda je bez akcelerace a demonstruje nejjednodušší způsob testu průniku. Metoda OCTREE je implementovaná ve třídě `OctreeAccelerator`, která definuje algoritmus vytvoření datové struktury Octree a test průniku nad touto strukturou. Podobně jsou na tom struktury UNIFORM GRID a BVH, jejichž implementaci lze najít ve třídách `UniformAccelerator` a `BVHAccelerator`. Všechny třídy implementují dvě varianty na test průniku paprsku s objektem. První varianta testu se zaměřuje pouze na zjištění, zda paprsek protne libovolný objekt v prostoru. Tento test je primárně určen pro výpočet stínového paprsku. Druhá varianta testu navíc sbírá informace o protnutí s nejbližším objektem v cestě paprsku. Tento test je použit pro standardní sledování cest paprsků. Na obrázku 6.3 je diagram tříd, které reprezentují použité akcelerační datové struktury.

Pro výpočet snímků pomocí CPU je třeba ověřit, zda je zvolena volba "CPU renderer". Vlastnosti výpočtu nelze ovlivnit přímo v aplikaci, ale jsou pevně stanoveny a možnost dynamické úpravy těchto vlastností by byly předmětem budoucího rozšíření. Jedná se především o určení maximální hloubky sledování trasy paprsku[39]. Po získání výsledné barvy pro daný paprsek je tato hodnota přidána do aktuálně aktivní textury jako další vzorek. Jak již bylo dříve zmíněno, přidávání vzorků funguje na principu poměrné váhy s ostatními vzorky v textuře. Proto, čím déle běží výpočet, tím je váha každého dalšího získaného vzorku nižší a nepředstavuje výraznější změny ve výsledném snímku. Dochází tak ke zpřesňování výsledků výpočtu.

[37] Stránka kurzu: http://cs.lth.se/edan30/
[38] Volba LIST nevyužívá žádné akcelerační datové struktury.
[39] Tato hodnota je experimentálně nastavena na hloubku 4

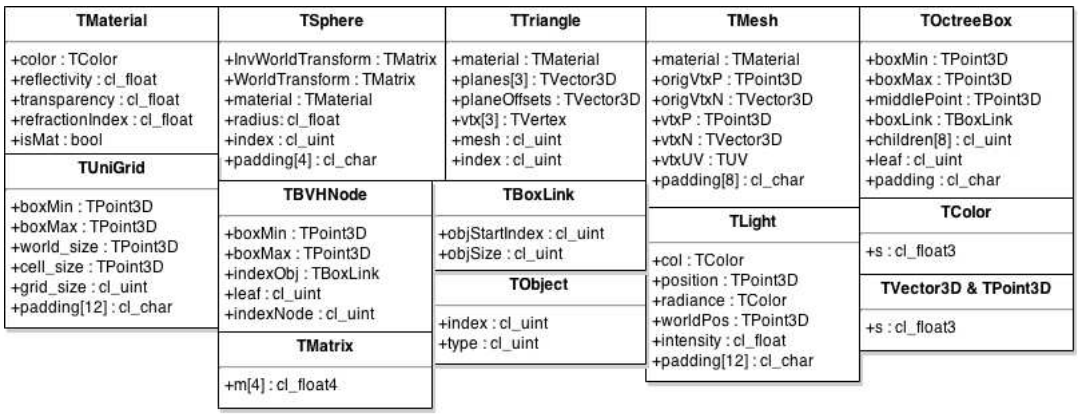


Obrázek 6.3: Diagram všech akceleračních datových struktur.

6.4.2 GPU pathtracer

Při implementaci GPU *pathtraceru* byly postupně zjišťovány omezení jazyka OpenCL. Z tohoto důvodu nešlo o přímočarou konverzi již existujícího řešení, které ve velké míře využívají funkčnosti jazyka C++ nebo technik jako je rekurze, dynamické alokace paměti a reference na objekt. Implementaci bylo proto třeba přizpůsobit nastaveným omezením a některé postupy simulovat (např. použití statických polí jako zásobníků).

Před zahájením vlastního výpočtu je nutné si připravit potřebná data z již existujících datových struktur CPU implementace a poté je nahrát do zvoleného kernelu[40] v podobě parametrů. Na obrázku 6.4 je zobrazeno několik struktur reprezentujících exportované data. Pokud potřebujeme vyexportovat větší množství dat (např. seznam proměnných), je nutné tato data vložit do bufferů s využitím funkcí `clCreateBuffer()`(pro vytvoření bufferu) a `clEnqueueWriteBuffer()`(pro zapsání dat do bufferu). Vytvořený buffer poté přiřadíme do kernelu jako parametr. Z kernelu lze přistupovat k jednotlivým instancím přes index (podobně jako přístup do pole).



Obrázek 6.4: Diagram datových struktur pro export do kernelu.

[40] Označení pro OpenCL program.

U některých datových struktur pro export dat bylo nutné provést navýšení paměťové náročnosti (proměnná padding) z důvodu kompatibility s datovými strukturami používanými v kernelu. Z důvodu efektivního využití paměti na čipu jsou datové struktury vytvořené v OpenCL zarovnávány a nemusí tak odpovídat zarovnání datových struktur v jiných jazycích. Proto bylo potřeba ověřit shodnost paměťové náročnosti datových struktur vytvořených pro export dat a pro příjem dat na straně kernelu. V opačném případě by mohlo dojít ke špatnému přiřazení vstupních dat a výpočet by neproběhl korektně.

GPU *pathtracer* je založen na algoritmech CPU implementace, kde pro každou konfiguraci výpočtu existuje samostatný OpenCL kernel (`gpu_pt_list`, `gpu_pt_octree`, `gpu_pt_unigrid` a `gpu_pt_bvh`). Tyto kernely se odlišují pouze funkcemi pro zjištění testů průniku s danou akcelerační datovou strukturou. Mimo tyto funkce je jádro výpočtu a pomocné metody pro všechny kernely stejné. Pro výpočet prvního snímku pomocí GPU *pathtraceru* jsou určeny kernely s příponou `*_first`, u kterých se vypočítané vzorky přiřadí do textury.

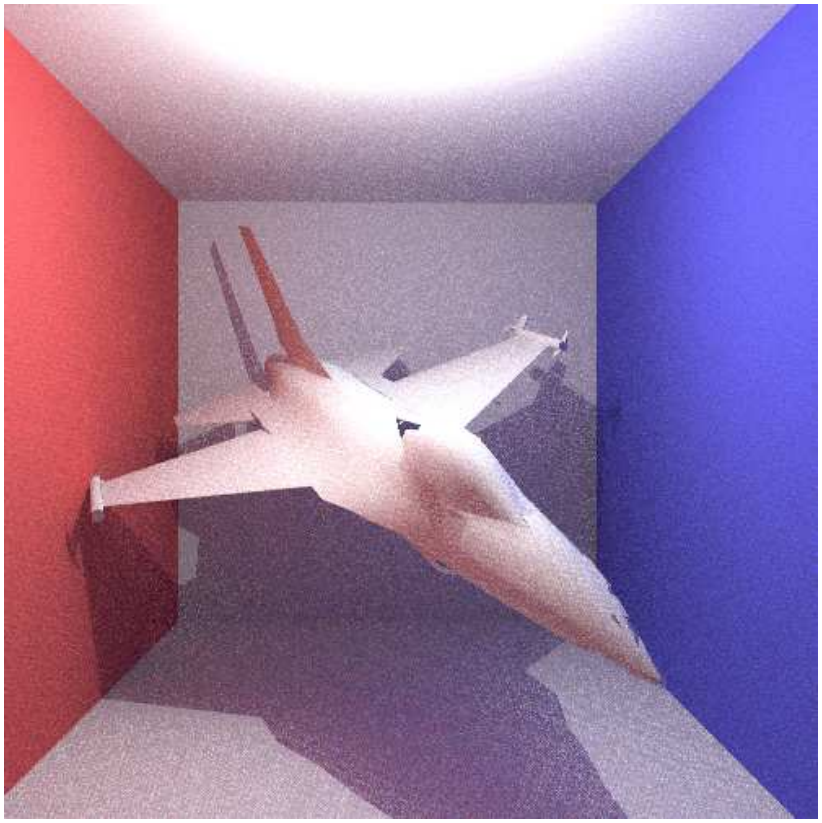
Jednotlivé metody CPU *pathtraceru* byly tak převedeny do (identické) statické podoby. Sledování cest paprsků je zde stejně jako u CPU implementace programově omezen na hloubku 4. Při zvyšování hloubky výpočtu lze narazit na problém s pevně definovanou velikostí statických polí, které jsou použity namísto datové struktury zásobníků u CPU implementace, které mohou být teoreticky nekonečné. Do tohoto pseudozásobníku se ukládající průběžné výsledky výpočtu, které jsou po dosažení maximální hloubky sledování cest paprsků použity pro finální výpočet barvy pro daný pixel. Pokud by bylo nutné zvýšit hloubku výpočtu, je potřeba ověřit, zda nedochází k přetečení daného pseudozásobníku a případně tak zvýšit kapacitu statického pole[41].

[41] Kapacita pole musí být vždy násobkem dvou.

Kapitola 7

Testování

Testování aplikace probíhalo na předem připravené scéně Cornell boxu, která obsahuje 5 stěn (načtených ze souboru plane.obj ze složky data), které tvoří hranice viditelné části projekce. Mezi stěnami je umístěn 3D model letounu f-16, který je složen z 4592 trojuhelníků. Celá scéna je osvětlena jediným světlem bílé barvy. Na obrázku 7.1 je ukázka testované scény. Výsledné časy odpovídají výpočtu jednoho snímku v rozlišení 500x500 pro danou konfiguraci. Výsledky zahrnují všechny možné kombinace konfigurace výpočtu. V tabulce 7.1 jsou uvedeny sestavy, na kterých byla implementace testována. V případě sestav 1 a 4 byly pro výpočet *pathtracingu* použity výpočetní jádra grafické karty. U sestav 2 a 3 byly pro výpočet vybrány výpočetní jádra procesoru.



Obrázek 7.1: Použitá scéna - Cornell box

#	OS	CPU	RAM	GPU
1	Windows 7 64-bit	Intel Core i5 661 @ 3.33GHz (2 cores)	2GB	Nvidia GTX 580
2	Windows 7 64-bit	Intel Core i5-2410M @ 2.30GHz (2 cores)	6GB	AMD Radeon 6490M
3	Windows 8.1 64-bit	Intel Pentium 2117U @ 1.80GHz (2 cores)	4GB	Intel Graphics HD
4	Windows 10 64-bit	AMD X6 FX-6300 @ 3.50GHz (6 cores)	4GB	Nvidia GTX 750 Ti

Tabulka 7.1: Specifikace testovacích sestav.

Výsledky měření jsou uvedeny v tabulkách 7.2 pro CPU implementaci a 7.3 pro GPU implementaci *pathtraceru*. Lze z nich odvodit, že poměry výkonnosti akceleračních datových struktur u GPU a CPU implementace *pathtracingu* jsou odlišné. U obou *pathtracerů* se ukazuje být nejpomalejší metoda LIST, tedy metoda bez akcelerační datové struktury. U metod OCTREE, UNIFORM GRID a BVH je kritická část volby rozsahu dělící funkce. U Octree rozhoduje volba maximální hloubky dělení prostoru a maximální počet objektů v uzlu před dosažením maximální hloubky uzlu. Pro Uniform grid je potřeba nastavit počet voxelů, které budou scénu ohraničovat. U Bounding volume hierarchy se nastavují stejné vlastnosti jako u Octree.

U každé sestavy bylo experimentálně měřeno, při kterém nastavení uvedených vlastností je výkonnost akcelerační datové struktury nejvyšší (tyto vlastnosti jsou uvedeny v tabulkách pro danou strukturu a testovací sestavu). Z výsledků je také patrný rozdíl mezi výpočtem GPU *pathtraceru* na CPU a GPU, kde grafická karta disponuje větším počtem výpočetních jader.

#	LIST	OCTREE	UNIFORM	BVH
1	2614.376s	35.945s (hloubka 7)	54.077s (70 buněk)	29.485s (hloubka 20)
2	3187.936s	41.981s (hloubka 7)	65.010s (80 buněk)	36.296s (hloubka 20)
3	4505.673s	60.506s (hloubka 7)	92.381s (70 buněk)	52.448s (hloubka 20)
4	3698.364s	49.102s (hloubka 7)	65.734s (100 buněk)	40.043s (hloubka 20)

Tabulka 7.2: Výsledky CPU implementace *pathtracingu*. Hodnoty jsou uvedeny pro výpočet 1 snímku v rozlišení 500x500.

#	LIST	OCTREE	UNIFORM	BVH
1	58.640s	5.640s (hloubka 7)	6.431s (110 buněk)	9.644s (hloubka 20)
2	561.167s	5.073s (hloubka 7)	13.881s (110 buněk)	22.850s (hloubka 20)
3	699.091s	6.793s (hloubka 7)	18.991s (110 buněk)	31.475s (hloubka 20)
4	59.899s	2.469s (hloubka 7)	5.532s (120 buněk)	8.732s (hloubka 20)

Tabulka 7.3: Výsledky GPU implementace *pathtracingu*. Hodnoty jsou uvedeny pro výpočet 1 snímku v rozlišení 500x500.

Kapitola 8

Závěr

Cílem této bakalářské práce bylo nastudování algoritmu *pathtracing* a jeho implementace na GPU. Tento cíl zahrnovalo nastudování a aplikaci akceleračních datových struktur pro urychlení výpočtu. Tato práce byla zpracována z pohledu testování rozdílu výkonnosti při použití CPU a GPU varianty *pathtracerů* v kombinaci s akceleračními datovými strukturami. Při testování bylo zjištěno, že všechny implementované struktury metody výrazně urychlují výpočet *pathtracingu* u obou implementací. Dále byl proveden návrh a implementace demonstrační aplikace, která zajišťuje názornou ukázku vypočtených snímků.

Během návrhu a implementace aplikace bylo potřeba prostudovat knihovnu OpenGL a *shadery*, pomocí kterých bylo vytvořeno prostředí aplikace a vykreslení vypočtených snímků metody *pathtracing*. Dále bylo potřeba se obeznámit s problematikou programování grafických procesorů s využitím jazyka OpenCL a všech jeho vlastností.

Výsledek implementace je aplikace schopná provádět výpočet snímků ve fotorealistické kvalitě. Kvalita výsledného snímku s co nejmenším vlivem „šumu“ se odvíjí od počtu získaných vzorků na pixel. Výsledný snímek lze v průběhu výpočtu uložit na disk.

Stávající aplikaci by bylo možné dále rozšířit v navazující magisterské práci například o implementaci metody oboustranného *pathtracingu* (nebo-li *Bidirectional pathtracing* [8]) nebo implementaci dalších (resp. optimalizaci stávajících) akceleračních datových struktur. Pro lepší srovnání výkonnosti implementací by bylo vhodné poskytnout více připravených scén, které by se lišili množstvím zobrazovaných objektů (a jejich složitostí) a uživatel by měl tak větší volnost ve výběru zobrazované scény.

Dále by bylo vhodné poskytnout uživateli větší míru interaktivity s aplikací. Pro lepší pochopení metody *pathtracing* by uživatel mohl mít možnost nastavit pozici pozorovatele ve scéně nebo mít možnost upravovat vlastnosti zobrazovaných objektů (barva, materiál, pozice) a světel. Intuitivním doplňkem by mohla být možnost se „projít“ po scéně stejně, jako je tomu u dnešních FPS počítačových her. Aby byl pohyb v prostoru plynulejší, bylo by vhodné implementovat jednoduchý renderer. Uživatel by tak po přepnutí na tento renderer mohl dynamicky upravovat pozici pozorovatele. Zdrojové kódy aplikace jsou k dispozici na přiloženém CD nebo v repozitáři služby GitHub[42].

Oblast zpracovávání fotorealistických scén je v době psaní této práce aktuální téma a každým rokem[43] představují výrobci grafických karet stále dokonalejší prototypy demonstrující výpočetně náročné metody na novém grafickém hardwaru.

[42] Webová stránka: https://github.com/karelbre/FITracer2015

[43] Např. konference ACM SIGGRAPH.

Literatura

[1] Appel, A.: Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), New York, NY, USA: ACM, 1968, s. 37–45, doi:10.1145/1468075.1468082.
URL http://doi.acm.org/10.1145/1468075.1468082

[2] Bailey, M.: Using GPU Shaders for Visualization, Part 2. *IEEE Computer Graphics and Applications*, ročník 31, č. 2, 2011: s. 67–73.
URL http://dblp.uni-trier.de/db/journals/cga/cga31.html#Bailey11

[3] Bikker, J.: *Ray tracing in real-time games : proefschrift*. Delft: Technische Universiteit, 2012, ISBN 978-90-5335-595-4.

[4] Bittner, J.; Hapala, M.; Havran, V.: Fast Insertion-Based Optimization of Bounding Volume Hierarchies. *Computer Graphics Forum*, ročník 32, č. 1, 2013: s. 85–100, ISSN 1467-8659, doi:10.1111/cgf.12000.
URL http://dx.doi.org/10.1111/cgf.12000

[5] Ericson, C.: *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, ISBN 1558607323.

[6] Havran Vlastimil, F. S.: Comparison of Hierarchical Grids. *Ray Tracing News*, ročník 12, č. 1, jun 1999.

[7] Kajiya, J. T.: The Rendering Equation. *SIGGRAPH Comput. Graph.*, ročník 20, č. 4, Srpen 1986: s. 143–150, ISSN 0097-8930, doi:10.1145/15886.15902.
URL http://doi.acm.org/10.1145/15886.15902

[8] Lafortune, E. P.; Willems, Y. D.: Bi-Directional Path Tracing. In *PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS '93)*, 1993, s. 145–153.

[9] Munshi, A.; Gaster, B.; Mattson, T. G.; aj.: *OpenCL Programming Guide*. Addison-Wesley Professional, první vydání, 2011, ISBN 0321749642, 9780321749642.

[10] Nicodemus, F. E.: Directional Reflectance and Emissivity of an Opaque Surface. *Applied Optics*, ročník 4, č. 7, Červenec 1965: s. 767–775.
URL http://www.opticsinfobase.org/abstract.cfm?URI=ao-4-7-767

[11] Pajot, A.; Barthe, L.; Paulin, M.; aj.: Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering. *Computer Graphics Forum*, ročník 30, č. 2, Duben 2011: s. 315–324.

[12] Shreiner, D.: *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., třetí vydání, 1999, ISBN 0201657651.

[13] Whitted, T.: An Improved Illumination Model for Shaded Display. *Commun. ACM*, ročník 23, č. 6, Červen 1980: s. 343–349, ISSN 0001-0782, doi:10.1145/358876.358882. URL http://doi.acm.org/10.1145/358876.358882

[14] Wu, J.-H.; Ohnishi, Y.; Shi, G.-H.; aj.: Theory of Three-Dimensional Discontinuous Deformation Analysis and Its Application to a Slope Toppling at Amatoribashi, Japan. *International Journal of Geomechanics*, ročník 5, č. 3, 2005: s. 179–195, doi:10.1061/(ASCE)1532-3641(2005)5:3(179), http://dx.doi.org/10.1061/(ASCE)1532-3641(2005)5:3(179). URL http://dx.doi.org/10.1061/(ASCE)1532-3641(2005)5:3(179)