VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RAY TRACING PRO GPUENGINE

RAY TRACING FOR GPUENGINE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. DAVID NOVÁK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ STARKA**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Novák David, Bc.**
Program: Informační technologie Obor: Počítačová grafika a multimédia
Název: **Raytracing pro GPUEngine**
Raytracing for GPUEngine
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte teorii týkající se raytracingu a jeho optimalizací pro GPU.
2. Navrhněte, které části je vhodné přenést na GPU vzhledem k urychlení výpočtů, a navrhněte algoritmy.
3. Doplňte potřebné části do knihovny GPUEngine.
4. Vytvořte demonstrační aplikaci raytracingu s využitím navržených GPU algoritmů/optimalizací.
5. Zhodnoťte dosažené výsledky
6. Vytvořte video prezentující práci.

Literatura:

- Dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Starka Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 6. listopadu 2018

Abstrakt

Hlavním cílem práce je optimalizace metody sledování paprsku, konkrétně pomocí akcelearčních datových struktur. Zaměřeno bude především na zamyšlení nad různými strategiemi stavby této struktury a jejího průchodu. V rámci práce budou implementovány a porovnány algoritmy běžící na CPU a na GPU, přesněji rychlost stavby a výsledná kvalita mající přímý vliv na rychlost výpočtu samotného sledování paprsku. K otestování kvality akcelerační struktury bude sloužit aplikace počítající zobrazování scény metodou sledování paprsku. Část stavby akceleračních struktur bude přidána do knihovny GPUEngine.

Abstract

The main goal of this thesis is ray tracing optimization, especially with the use of acceleration data structure. It'll be focused on discretion about various structure build strategies and their traversal. Different algorithms on the CPU and on the GPU will be implemented and compared in the thesis, specifically will be compared the speed of build and final structure quality, which have a direct influence on ray tracing performance. A ray tracing application will be implemented for the purpose of the acceleration structure quality test. A part with acceleration structure building will be added to GPUEngine library.

Klíčová slova

Sledování paprsku, hierarchie obalových těles, OpenGL, optimalizace sledování paprsku

Keywords

Ray tracing, Bounding volume hierarchy, OpenGL, Ray tracing optimization

Citace

NOVÁK, David. *Ray Tracing pro GPUEngine*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Starka

Ray Tracing pro GPUEngine

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Starky.

.....
David Novák
16. května 2019

Poděkování

Děkuji svému vedoucímu práce za jeho vedení a poskytnuté rady a svým blízkým za podporu.

Obsah

Kapitola 1

Úvod

V oblasti počítačové grafiky existuje celá řada způsobu zobrazování scény nebo jinými slovy počítání viditelnosti. V současnosti bezesporu vládne rasterizace, jež je dnes hardwarově implementována na každé grafické kartě. Dalo by se říci, že tato metoda je nejvhodnějším řešení zobrazování vzhledem k poměrně snadné implementaci a výsledné rychlosti. Existuje však řada oblastí, kde není rasterizace nejvhodnějším řešením. Zde lze hovořit například o filmovém průmyslu, kde jsou velmi vysoké požadavky na vizuální kvalitu výsledného snímku. V tomto bodě však přichází komplikace, jelikož při využití rasterizace je problémem korektní zobrazování stínů, odlesků, případně lomu světla. S těmito problémy se postupem času podařilo vypořádat, kdy například stíny lze počítat metodami Shadow Mapping, Shadow Volumes, případně pokročilejšími technikami. Co se však vizuální kvality týče, lepších výsledků dosahuje metoda sledování paprsků neboli Ray tracing. Tato metoda je již poměrně stará, vychází z článku Turnera Whitteda z 80. let minulého století. Bezespornou výhodou sledování paprsku je možnost aproximovat chování paprsku v reálném světě a tím pádem možnost modelovat různé optické jevy (stíny, odrazy světla, lom světla). Největší nevýhodou však zůstává výkonnost, která dalece zaostává za výkonností klasické rasterizace. V již zmíněném filmovém průmyslu to však tolik na škodu není, zde je rozhodujícím faktorem vizuální kvalita, nikoli výkonnost algoritmu.

V současné době se začíná ukazovat, že je možné využití této techniky i v herním průmyslu, konkrétně právě na řešení stínů a odrazů světla. Jako příklady je možné zmínit hry Shadow of the Tomb Raider a Battlefield V, které poskytují možnost zobrazování těchto optických efektů metodou Ray Tracing (v tomto případě je využito tzv. RT Cores vyskytujících se na čipech společnosti nVidia od architektury Turing). Zde již však nastává problém, kdy vzniká požadavek na určitou vizuální kvalitu a zároveň rychlost výpočtu, tedy souběh dvou protichůdných parametrů. Pokud se podíváme na náročnost vykreslování dříve zmíněných her (viz benchmarkové testy ukazující signifikantní vliv na výkon aplikace[1]), kdy je využito sledování paprsku, je patrná náročnost tohoto algoritmu a je potřeba si uvědomit, že je využita hardwarová akcelerace a není řešena celá viditelnost, ale pouze některé optické jevy.

Způsobů akcelerace sledování paprsku existuje několik, za zmínku však stojí metody cullingu, tedy zahazování geometrie nepotřebné pro vykreslování, a využití akceleračních struktur. Těch existuje celá řada a každá má své výhody a nevýhody. Urychlení tímto způsobem spočívá v ideji, že je nepodstatné zabývat se geometrií, jež se nachází mimo oblast, kam směřuje paprsek. Jinými slovy akcelerační struktury různým způsobem obalují

[1] Benchmark test hry Battlefield V

geometrii scény a při výpočtu konkrétního paprsku se pouze zabýváme geometrií, jež náleží zasaženému obalu (tudíž zbylým primitivům se již nevěnujeme). Tato oblast je již v dnešní době poměrně rozsáhlá, ať už co se týče konceptu struktury, její stavby nebo algoritmů průchodu. Zmínit lze jednodušší strukturu jako je uniformní mřížka až po pokročilejší jako je například hierarchie obalových těles, avšak tomuto tématu bude věnována detailnější diskuze v následujících kapitolách.

V rámci této práce bude nejprve po teoretické stránce vysvětlen algoritmus sledování paprsku, tedy řešení viditelnosti, jeho vlastnosti včetně tzv. úzkých hrdel algoritmu. V další části budou představeny akcelerační struktury, způsoby jejich vytváření, a to jak na CPU, tak i paralelizované verze na GPU a v neposlední řadě způsoby průchodu strukturou. Dále bude vysvětlen návrh vyvíjené aplikace a diskutovány použité technologie a principy. Na závěr bude provedeno testování výkonnosti vykreslování vytvořenou aplikací a zhodnocení dosažení cílů práce.

Kapitola 2

Ray Tracing

Tato kapitola se věnuje představení a teoretickému rozboru metody sledování paprsku, rozšířené metody distribuovaného sledování paprsku, vlastnostem těchto algoritmů a také matematickému aparátu využitém v této metodě.

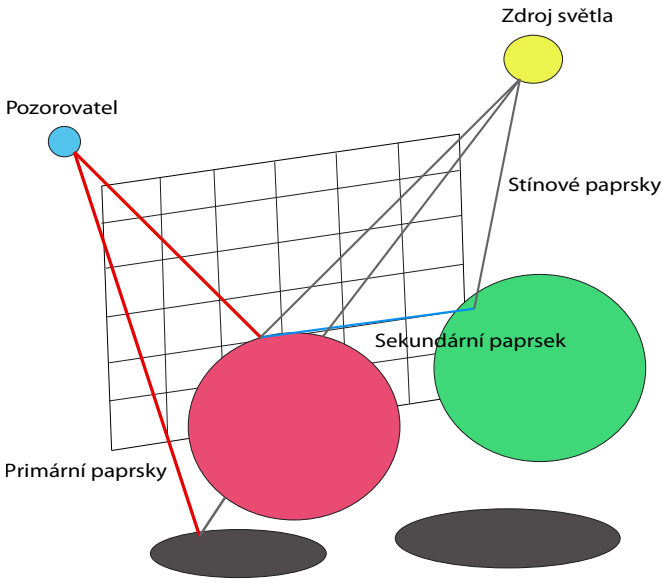
2.1 Princip sledování paprsku

Jedná se o metodu zobrazování scény, chcete-li způsob řešení viditelnosti. Založená je na metodě vrhání paprsku [4] (Ray Casting) představené Arthurem Appelem již v 60. letech, používané například při zobrazování volumetrických dat. Tuto metodu o několik let později rozšířil Turner Whitted do podoby sledování paprsku [20]. Jak již název říká viditelnost se řeší sledováním množství paprsků a jejich interakcí se sledovaným prostředím. Tento přístup velmi dobrým způsobem umožňuje modelovat světlo z pohledu geometrické optiky (lom, odraz), umožňuje zároveň korektní počítání tvrdých stínů.

Princip metody je vcelku prostý, do scény jsou od pozorovatele skrze průmětnu vysílány paprsky (ilustrováno na obrázku 2.1). Pokud se konkrétní paprsek protne s geometrií scény, dojde k vyslání stínového paprsku do zdroje světla (obecně do všech zdrojů světla) a vyhodnotí se osvětlení v daném bodě. Další chování paprsku záleží na charakteristice materiálu v daném bodě (tím může být například klasický index lomu). Může se provést odraz, lom nebo ukončit sledování. I v případě odrazu či lomu však po několika kolizích dochází k ukončení sledování, jelikož při každé interakci s geometrií scény je část světla odražena respektive pohlcena. Toto je možné modelovat různými způsoby více či méně fyzikálně korektně.

Původní idea sledování paprsku byla lehce odlišná, konkrétně neprobíhalo sledování paprsku z pozice pozorovatele nýbrž ze zdroje světla. Ač je tento přístup z fyzikálního hlediska správný, jeho vadou je, že velké množství paprsků nedojde k pozorovateli a na výsledném snímku se nepodílejí, což zároveň znamená vyslání většího množství paprsků nebo případnou rekonstrukci finálního snímku s použitím menšího počtu paprsků.

Sledování paprsků má však určitá omezení, konkrétně se jedná o nemožnost počítání měkkých stínů, nepřímého osvětlení, propagaci světla a několik dalších efektů. Sama metoda sledování paprsků bývá označována za metodu řešení globálního osvětlení scény, avšak právě z této metody vychází další přístupy, které řeší globální osvětlení lépe. Zmínit lze například distribuovaný Ray Tracing, Path Tracing nebo Photon Mapping. Tyto metody dokáží řešit lépe problém globálního osvětlení, ale za cenu mnohdy vyšší výpočetní náročnosti a za faktu, že každá má specifické problémy, jenž se obvykle projevují viditelným šumem.



Obrázek 2.1: Ilustrace principu sledování cest. Paprsky vychází od pozorovatele skrze průmětnu do scény, kde koliduje s objekty. Z těchto bodů jsou dále vrhány stínové paprsky ke zdroji světla, případně odražené sekundární paprsky. Zdroj: vlastní.

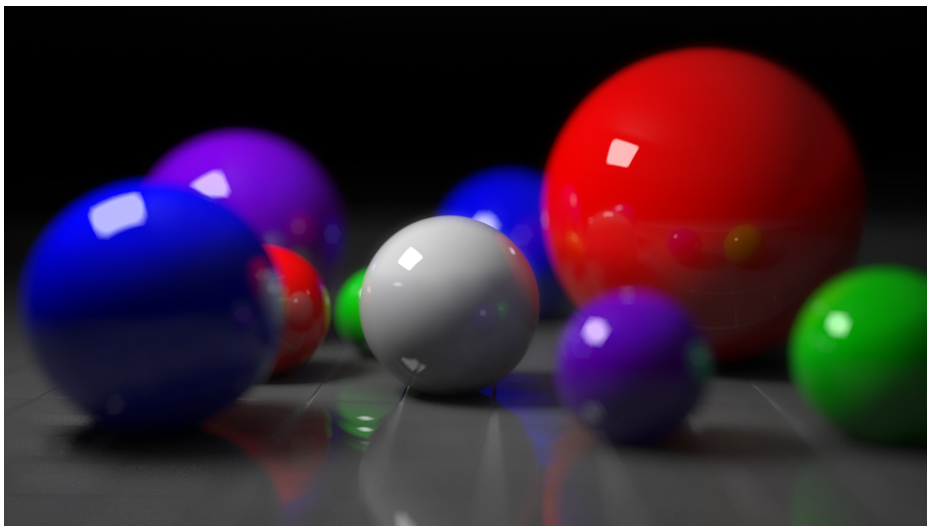
Úzkým hrdlem algoritmu sledování cest z hlediska výkonu je hledání nejbližší kolize paprsku s geometrií. Ostatní části algoritmu obvykle nebývají náročné, tedy až na případy, kdy je počítáno osvětlení v bodě nějakým komplexnějším přístupem nebo kdy je chování světla vyhodnocováno přesnými fyzikálními vztahy. Stále však zůstává v platnosti tvrzení, že hledání oné kolize je významně náročnější a s komplexitou zobrazované scény tato náročnost nadále vzrůstá. V zásadě není příliš způsobů akcelerace, nejvýznamnějším bývá snížení výpočetní složitosti za pomocí akceleračních datových struktur. Toto téma je však detailněji rozebráno v následující kapitole.

2.2 Distribuované sledování paprsku

Distribuované sledování paprsku [6] je určitým pokračovatelem metody sledování paprsku. Původní metoda sice uměla dobře modelovat odlesky, lom světla a podobné optické jevy, problémem byl však, že nedokáže zobrazovat měkké stíny, simulovat matné povrchy nebo například simulovat hloubku pole.

S těmito nedostatky si dokáže poradit právě distribuované sledování paprsku. Princip je obdobný a to, že jsou vysílány do scény paprsky a hledá se kolize s geometrií scény. Z daného bodu kolize poté není vyslán pouze jeden, ale několik paprsků s náhodně vybraným směrem. Tyto paprsky jsou ideálně rovnoměrně rozprostřené na jednotkové hemisféře okolo daného bodu. Takto lze modelovat optické jevy jako měkké stíny, které více odpovídají reálnému chování, jelikož bodové zdroje světla (a s nimi spojené tvrdé stíny) se obvykle nevyskytují. Dále je možné takto modelovat i měkké odrazy, což opět více odpovídá realitě,

kdy se běžně nevyskytují materiály, které by měly dokonale hladký povrch, a tím pádem odrážely světlo vždy jedním směrem. Není však nutné, aby byly paprsky vysílané jen z bodu kolize. Je možné provádět distribuci přes pixel, tím pádem lze získat efekt antialiasingu. Dále je možné provádět distribuci přes čočku (tedy z pozice pozorovatele), a tím pádem lze simulovat jev hloubky pole (nebo-li tzv. Depth of field), kdy se vzdálenější objekty jeví více rozostřené, jelikož je vzhledem k větší vzdálenosti mine více paprsků. Dále je možné zároveň provádět distribuci v čase, kdy je stejný paprsek vyslán do scény v jiný čas a tím lze modelovat efekt rozostření pohybu. Snímek zobrazený metodou distribuovaného sledování paprsku lze vidět na obrázku 2.2.



Obrázek 2.2: Scéna zobrazená pomocí metody distribuovaného sledování cest. Na snímku je možné vidět optické jevy jako jsou měkké stíny, měkké odlesky od povrchu nebo rozostření okolí. Převzato z webu[2].

Díky této metodě tedy můžeme získávat lepší výsledky, ale zároveň je také třeba počítat s určitými problémy. Prvním takovým problémem může být počet vyslaných paprsků. Konkrétně řečeno, pokud jsou například modelovány měkké stíny, tak nízký počet vyslaných paprsků způsobí viditelný šum, pro lepší výsledek je tedy nutné zvýšit počet vyslaných paprsků. To se však logicky musí projevit na celkovém výkonu, jelikož právě výpočet paprsků je nejdražší částí tohoto algoritmu. Tento problém se značně projevuje při modelování hloubky pole nebo antialiasingu, kdy je v podstatě celá scéna vykreslována několikrát. Dalším problémem je nutnost kvalitního generátoru náhodných čísel, kdy použití méně kvalitního generátoru může znatelně degradovat výslednou kvalitu snímku. V tomto případě je možné použít klasický lineární kongruentní generátor, který by neměl mít větší vliv na výkonnost, popřípadě kvalitnější Merssenne twister. Jako problém se to však může jevit při akcelerované implementaci na GPU, kdy je potřeba implementovat generátor generující nezávisle na konkrétní výpočetní invokaci. V opačném případě mohou být na výsledku patrné určité artefakty.

Celkově vzato tato metoda zvyšuje možnosti klasického sledování paprsku, avšak za cenu výše zmíněných problémů, což je však v některých případech pouze marginální záležitost. Pokud bychom však chtěli modelovat globální osvětlení realističtějším způsobem, je možné

[2] http://www.wikiwand.com/en/Ray_tracing_(graphics)

použít pokročilé metody založené na sledování paprsku, a to Path tracing [11], který je založen na sledování cest, které jsou náhodně voleny a sledovány ve scéně, přičemž s počtem cest se konverguje k výsledku. Další takovou metodou je Photon mapping [10], kde jsou do scény vysílány fotony, přičemž je jejich poloha uložena do fotonové mapy, následně proběhne zobrazení scény. Vliv na kvalitu výsledku má potom množství vyslaných fotonů. Vylepšením předchozích metod potom vznikl Vertex connection and merge [7], který nejprve sleduje cesty ze zdroje světla a ukládá si pozice kolizí se scénou, potom rekonstruuje možné cesty do zdroje světla pro každý pixel. Přínosem této metody je, že odstraňuje problémy předchozích metod a zároveň dokáže vytvořit kvalitnější výsledek v nižším čase. Obecně lze říct, že tyto algoritmy dokáží produkovat výsledky, které se dokáží blížit realitě, avšak za cenu vysoké výpočetní náročnosti.

2.3 Fyzikální reprezentace paprsku

Z fyzikálního hlediska je paprsek [9] chápán jako křivka určující směr vlnění, která je zároveň vždy kolmá k vlnoploše. Takováto křivka má obvykle podobu přímky respektive polopřímky, jejíž počátek je shodný se zdrojovým bodem vlnění. Z matematického hlediska lze tedy paprsek chápat jako klasickou polopřímku, kterou lze modelovat za pomocí dvou vektorů, a to bodem počátku a směrovým vektorem. Množinu bodů ležících na paprsku lze potom popsat rovnicí 2.1.

$$\vec{P(t)} = \vec{O} + t\vec{D} \mid t \geq 0 \tag{2.1}$$

Proměnná $\vec{P(t)}$ v rovnici tedy popisuje libovolný bod ležící na paprsku, vektor $\vec{O}$ určuje počáteční bod paprsku, vektor $\vec{D}$ definuje směr paprsku a proměnná t určuje délku paprsku. Právě hodnotu proměnné t se snažíme nalézt při hledání bodu kolize paprsku se scénou.

Při výpočtu algoritmu sledování paprsku je nejprve vrhán do scény primární paprsek, tedy bod počátku shodý s pozicí pozorovatele a směrový vektor je získán jako normalizovaný rozdíl pozice konkrétního bodu průmětny a pozice pozorovatele. Poté se hledá nejbližší bod kolize, formálně tedy bod $\vec{P(t)}$ ležící na paprsku, který koliduje s geometrií scény mající nejnižší hodnotu proměnné t. Další cesta paprsku závisí především na způsobu modelování optických vlastností předmětů ve scéně. Je možné jej pouze odrazit, ale fyzikálně správný postup je určit podle Fresnelových rovnic množství světla, jež bude odraženo a množství, které bude pohlceno v závislosti na indexu lomu prostředí a úhlu dopadu paprsku na povrch.

Jak již bylo zmíněno, klíčovou součástí výpočtu je hledání bodu kolize paprsku se scénou. Způsob tohoto výpočtu závisí na reprezentaci scény. Ta bývá v dnešní době většinou reprezentována pomocí trojúhelníků, tudíž kolize je počítána jako průsečík polopřímky a trojúhelníku. Existuje několik algoritmů pro řešení tohoto problému. Patrně nejpoužívanějším je v této oblasti algoritmus Möller-Trumbore [17]. Formálně vychází z rovnice 2.2, kde levá strana popisuje libovolný bod ležící na paprsku a pravá strana definuje libovolný bod uvnitř trojúhelníku.

$$\vec{O} + t\vec{D} = (1 - u - v)\vec{V_0} + u\vec{V_1} + v\vec{V_2} \tag{2.2}$$

Po přeuspořádání proměnných a následné aplikaci Cramerova pravidla dostaneme tvar z rovnice 2.3, kde $\vec{E_1}$ a $\vec{E_2}$ jsou hrany trojúhelníku, $\vec{T}$ je rozdíl počátku paprsku a vrcholu $\vec{V_0}$, $\vec{P}$ je vektorový součin směrového vektoru a hrany $\vec{E_2}$ a $\vec{Q}$ je vektorový součin vektoru $\vec{T}$ a hrany $\vec{E_1}$.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\vec{P} \cdot \vec{E_1}} \begin{bmatrix} \vec{Q} & \vec{E_2} \\ \vec{P} & \vec{T} \\ \vec{Q} & \vec{D} \end{bmatrix} \tag{2.3}$$

Dalším, relativně novým přístupem k této problematice může být algoritmus od Douga Baldwina a Michaela Webera [5]. Ten je založen na předpočítání matic, transformujících globální souřadnice do barycentrických, pro každý trojúhelník. Tato matice poté násobí paprsek, následně jsou získány vzdálenost t a barycentrické souřadnice u a v. Ty jsou následně otestovány, zda leží v intervalu $< 0, 1 >$ (jinými slovy leží uvnitř trojúhleníku). Tato metoda může být v určitých podmínkách rychlejší než dříve zmíněný algoritmus, avšak za cenu paměťových nároků a úvodního předpočítání matic.

Kapitola 3

Architektura GPU

Hlavním účelem grafických čipů je akcelerace počítačové grafiky vzhledem k faktu, že algoritmy z této oblasti patří k výpočetně náročnějším. Zpočátku byla velká část funkcionality hardwarově definována s minimální možností jakékoliv variability. V současné době je však velká část GPU programovatelná a je možné ji využít nejen pro počítačovou grafiku, ale i pro obecné výpočty (tzv. GPGPU). Zásadním faktorem je zde právě možnost akcelerace daných výpočtů. Nelze však říci, že každý algoritmus lze takto akcelerovat, u některých toto může být problémem, vzhledem k tomu, že GPU má v určitých ohledech jiné koncepty než klasický procesor, a proto je tedy nutné akcelerovaný algoritmus přizpůsobit této architektuře.

Cílem této kapitoly je vysvětlit základní koncepty GPU a ukázat rozdíly oproti procesorům, a tím i ukázat cestu jak správně přemýšlet při návrhu algoritmu, jenž má běžet na GPU.

3.1 Základní koncepce

Obecně je GPU [1] stavěno na konceptu SIMT (single instruction, multiple threads), tedy jedna instrukce respektive posloupnost instrukcí je vykonávána mnoha vlákny, přičemž každé vlákno vykonává tyto instrukce nad svými daty. Tento princip plyne z logiky věci, kdy je vhodné akcelerovat grafické aplikace pomocí paralelizace (paralelní zpracování více vrcholů, fragmentů apod.). Zde je možné vidět první rozdíl oproti klasickým procesorům, kde je možnost paralelismu více limitována. Je však pochopitelné, že ačkoliv GPU obsahuje velké množství výpočetních jednotek, nebudou obsahovat takové množství funkcionality, které je obsažené v dnešních procesorech. Nesetkáme se zde například s prediktory skoků, které jsou dnes samozřejmou součástí procesoru sloužící při vyhodnocování větvení, nevyskytují se zde ani trace cache pro přednačítání instrukcí nebo logika umožňující spekulativní vykonávání instrukcí. Z toho lze vyvodit, že program obsahující velké množství větvení není ideálním pro grafický čip (a to nejen z důvodu absence predikce skoků). Ideální je pro něj program, jenž provádí mnoho výpočtů s minimem větvení přičemž má data rychle k dispozici.

Srdcem grafického procesoru je tzv. Stream multiprocessor (případně také CUDA jádro) na němž jsou jednotlivá vlákna vykonávána. Takovýchto jednotek je na grafickém procesoru celá řada (v současné době několik desítek), ilustrace SM je ukázána na obrázku 3.1. Vlákna jsou na multiprocesory alokována po pracovních skupinách. Na základě požadavků vláken ve skupině na množství použitých registrů případně sdílené paměti se určí, kolik skupin bude možné na multiprocesor přidělit. Tato vlákna jsou poté rozdělena na hardwarově de-

finované bloky tzv. warpy, jenž obvykle obsahují 32 případně 64 vláken. Každé vlákno ve warpu potom vykonává vždy jednu stejnou instrukci (sdílí programový čítač). Tímto způsobem lze efektivně vykonávat množství vláken najednou, avšak problém nastává při větvení, kdy se cesty vláken mohou rozcházet, tento jev je nazýván jako divergence vláken. Dalším problémem může být nekonzistence přístupů do paměti, kdy je pro jeden warp nutné několik přístupů do globální paměti (tyto přístupy jsou z časového hlediska drahé), ideální varianta nastává pokud jsou data pro warp získána jediným přístupem. Řešení těchto problémů, a s tím související efektivní vykonávání programu, je však plně v rukou programátora.

Vzhledem k paralelnímu charakteru vykonávání aplikací na GPU je na místě otázka ohledně synchronizace. Ta je pochopitelně možná a je realizována pomocí bariér, které fungují na úrovni pracovní skupiny. Synchronizace v rámci warpu nedává smysl a synchronizace celé úlohy napříč všemi skupinami by byla obtížná a znamenala by značný dopad na výkon. Synchronizace celé úlohy je možné provádět ze strany procesoru pomocí různých bariér.

Obrázek 3.1: Schéma stream multiprocesoru. Každý multiprocesor obsahuje množství výpočetních jednotek, velké množství (řádově desetitisíce) registrů, texturovací jednotky, vyrovnávací paměť cache, sdílenou paměť a také plánovací logiku. Převzato z webu[2].

Nyní nastává otázka, jak tedy psát aplikace pro jejich efektivní běh na GPU. Velmi důležité je správným způsobem využívat dostupné paměti, dobré zásobování výpočetních jednotek daty je základem vysoké výkonnosti. V tomto ohledu jsou nejdražší přístupy do globální paměti. Jejich úplná eliminace obvykle není možná, to se však nemusí projevit na výkonnosti vzhledem k faktu, že probíhá vykrývání těchto latencí pomocí vykonávání jiného warpu. Jinými slovy, pokud dojde k přístupu do globální paměti, dojde k přepnutí na jiný warp (režie přepnutí bývá minimální), který vykonává své instrukce, zatímco probíhá čtení

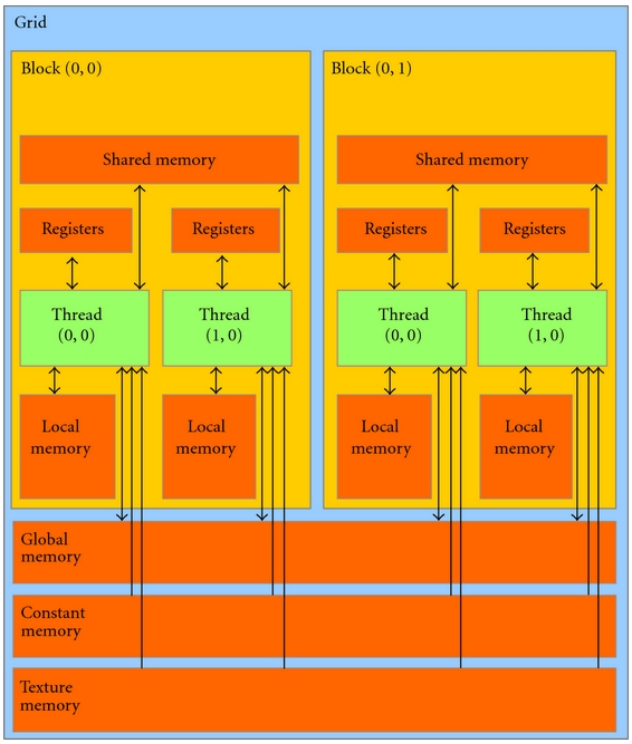
[2] https://www.researchgate.net/profile/Jason_Spencer/publication/51952258/figure/fig1/AS:305935617675265@1449952143398/Streaming-Multiprocessor-detail-NVIDIA-Corporation-Finally-accessing-global-memory.png

respektive zápis do globální paměti. Dále je potřeba brát ohled na množství registrů, jenž využívá jedno vlákno. Ačkoliv by se mohlo zdát, že jich je dostatek (narozdíl například od klasického procesoru), musí se brát v potaz, že registry multiprocesoru mohou sdílet stovky až tisíce vláken, navíc v případě nedostatku registrů dochází k jejich odkládání na paměť mimo čip.

Celkově se dá říci, že při vývoji aplikací pro GPU je nutné brát v potaz řadu aspektů, aby aplikace běžela maximálně efektivně, což je hlavní cíl využití GPU. Hlavní doménou těchto čipů je paralelní zpracování dat, tím pádem nelze říci, že lze každý algoritmus akcelerovat využitím grafického procesoru, v určitým případech tomu může být naopak a lepší výkonnost bude mít klasický procesor.

3.2 Hierarchie pamětí

U grafických procesorů se setkáváme s poměrně velkým množstvím druhů pamětí. Důvod jejich existence je ekonomický a technologický. Práce s pamětí obvykle mívá velký vliv na výkonnost aplikací běžících na GPU (ať už pozitivní či negativní). Jednotlivé typy pamětí mají různý účel a tím pádem i rozdílné parametry (kapacita, rychlost přístupu atd.). Ilustrace hierarchie pamětí je na obrázku 3.2.



Obrázek 3.2: Hierarchie pamětí na GPU. Nejrychlejším typem pamětí jsou registry a sdílená paměť, avšak nenabízejí takovou kapacitu. Zatímco globální a texturní paměť nabízejí velkou kapacitu, leží mimo samotný čip a přístup k nim je tudíž poměrně drahý. Převzato z webu[4].

Prvním typem paměti je paměť globální. Její kapacita se v dnešní době obvykle pohybuje v řádech gigabytů, zároveň se nachází mimo čip, což znamená poměrně velké latence při přístupu. Složí k ukládání velkého množství dat (např. buffery vrcholů, primitiv, částic apod.), které budou využívat vlákna napříč skupinami. Lze ji taktéž využít jako prostředek pro komunikaci mezi vlákny v jiných skupinách. Pro řešení drahých přístupů k této paměti se využívá mimo vykrytí latencí pomocí běhu jiných warpů také cachování, jež je na některých architekturách řešeno automaticky (pomocí L2 cache) nebo lze využít sdílenou paměť.

Dalším typem paměti se kterou se na těchto architekturách setkáváme je paměť texturní. Jak samotný název napovídá slouží k ukládání texturových struktur. Pro práci s texturami obsahuje GPU speciální hardware tzv. texturovací jednotky. Ty slouží pro přístup k texturám, interpolaci hodnoty a dalším operacím. GPU zároveň obsahuje texturovací cache, která zajišťuje cachování dat z textur, jelikož texturovací paměť leží stejně jako globální paměť mimo čip, což znamená velkou latenci při přístupu.

Mimo čip leží také konstantní paměť mající velikost obvykle v řádu desítek kilobytů. Je optimalizována na broadcast, tedy poskytnutí určité hodnoty všem vláknům. Její účel je tedy poskytnout malé množství dat, které je konstantní pro všechna vlákna (např. pozice světel, kamery apod.). I zde funguje cachování pomocí tzv. konstantní cache.

Poměrně důležitou roli z hlediska zvýšení výkonu aplikace hraje sdílená paměť. Její kapacita se v současnosti pohybuje v desítkách kilobytů. Slouží jako prostředek ke komunikaci mezi vlákny v rámci jedné pracovní skupiny. Dále také může sloužit jako uživatelsky řízená cache. Je organizována do 16, dnes již běžně do 32 bank obvykle po 4 bytech. Důvodem organizace do bank je možnost obsloužit více přístupů vláken do sdílené paměti najednou, tudíž se předpokládá, že každé vlákno ve warpu přistoupí do jiné banky. V případě přístupu více vláken do jedné banky mohou nastat dvě situace, je-li přistupována jedna adresa z banky, data z této adresy jsou zaslána všem vláknům, která o ně žádají najednou. Pokud jsou však přistupovány jiné adresy v rámci jedné banky, dochází k bankovému konfliktu. V tomto případě jsou přístupy serializovány, což se projeví snížením výkonnosti aplikace. Avšak vzhledem k faktu, že její obsah je řízen uživatelem, tak je i jeho úkolem uložit data do sdílené paměti tím způsobem, aby se tyto konflikty minimalizovaly respektive eliminovaly. Vzhledem k faktu, že se sdílená paměť nachází na čipu, je tudíž doba přístupu podobná jako u registrů. U některých aplikací tak může tento typ paměti pomoci značně zvýšit výkonnost, avšak v případě mnohačetných bankových konflitků naopak významně srazit celkovou výkonnost.

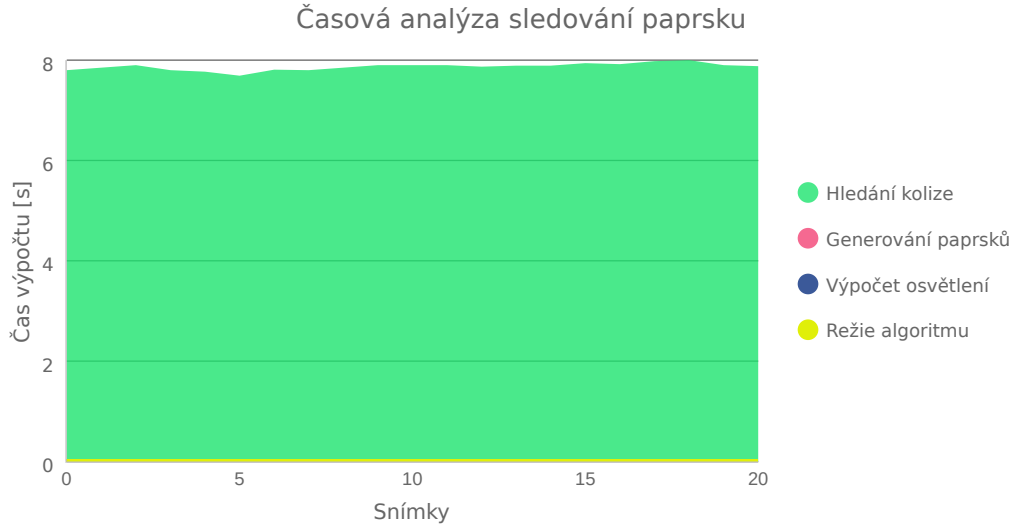
Každý multiprocesor obsahuje několik desítek tisíc registrů, o které se však musí dělit všechna vlákna v rámci pracovní skupiny. Jedná se o typ paměti s nejrychlejším přístupem. Obsah registrů je pro každé vlákno soukromý a není možné jej přistupovat z jiného vlákna (možné je to pouze v rámci jednoho warpu pomocí operace shuffle, která nemusí být vždy podporována). V případě, že na potřebná data pro jedno vlákno není dostatek registrů využije se lokální paměť (tzv. register spilling). Data, která budou umístěna do lokální paměti a která zůstanou v registrech určuje běžně kompilátor. Problémem je, že tato paměť opět leží mimo čip, avšak tento typ paměti je cachován pomocí L1 cache.

[4] https://www.hindawi.com/journals/afs/2012/698062.fig.002.jpg

Kapitola 4

Akcelerace Ray Tracingu

Akceleraci sledování paprsku je možné realizovat kupříkladu jeho paralelizací, k čemuž je vhodné využití grafického čipu. To samotné však nemusí stačit vzhledem ke komplexitě výpočtu jediného pixelu. Při menší analýze algoritmu sledování paprsku (graf na obrázku 4.1) jde celkem zřetelně vidět, kde padá výkonnost. Pochopitelně za ním stojí výpočet kolize paprsku se scénou. Samotný algoritmus výpočtu kolize trojúhelníku s polopřímkou nějakým výrazným způsobem urychlit nejde. Jediným způsobem, kde získat ztracený výkon, je snížení množství těchto výpočtů. Zde existují některé způsoby, které mají potenciál dosažení vyššího výsledného výkonu. Tento způsob bývá označován jako akcelerační datové struktury, jež mají poměrně široké využití, a to nejen v oblasti počítačové grafiky, ale například v prostorových databázích nebo při akceleraci simulací či při výpočtu kolizí.



Obrázek 4.1: Časová analýza sledování paprsků pro jednoduchou scénu nad open-source raytracerem nanoRT[2]. V grafu lze vidět jakým způsobem se jednotlivé části výpočtu podílejí na celkovém času. Zatímco výpočet osvětlení či režie algoritmů mají zanedbatelný vliv na výsledný čas, zdaleka nejvíce času stráví algoritmus při hledání kolize paprsku se scénou. Pro chod sledování paprsku v reálném čase je prakticky nezbytné zaměřit se na akceleraci této části. Zdroj: vlastní.

Následující kapitola se věnuje tématu akcelerace sledovaní paprsku a jemu podobných metod. Popíše způsoby, jakými lze snížit obecnou složitost výpočtu, a tím ho i zrychlit. Konkrétně bude soustředěno na akcelerační datové struktury, přesněji strategie jejich stavby a způsoby průchodu jimi.

4.1 Obecný přehled akceleračních struktur

Pod pojmem akcelerační struktura se dá v jednoduchosti představit struktura, která dělí danou geometrii rozloženou v prostoru. Otázkou může být potom, jakým způsobem prostor dělit a jakou strategii k tomu použít. V dnešní době se využívá několik druhů takovýchto struktur, zmínit lze hierarchii obalových těles, uniformní mřížku nebo například Octree. Využití těchto struktur není omezeno pouze na sledování paprsku (či jiné aplikace z oblasti počítačové grafiky), ale obecně na libovolné algoritmy pracující s prostorovými daty vyžadující určitou akceleraci při výpočtu.

Základní myšlenkou při akceleraci výpočtů za použití této techniky je obecně řečeno snížení časové složitosti výpočtu, kdy redukujeme počet vykonaných operací při výpočtu kolize. Jinými slovy řečeno jednotlivá primitiva ve scéně jsou obalena určitými tělesy, přičemž se provádí výpočty pouze nad těmi primitivy, jenž leží v zasaženém obalovém tělese. Pokud zvážíme konkrétní případ, kde předpokládáme, že primitiva představují trojúhelníky a jako obalová tělesa jsou použity koule, navíc při faktu, že testování průsečíku polopřímky a koule vyžaduje méně výpočtů než v případě trojúhleníku, tak vidíme, že s poměrně jednoduchými výpočty dokážeme efektivně redukovat množství primitiv, nad kterými bude probíhat výpočet. Je však potřeba zvážit, jak bude scéna rozdělena, při nevhodném rozdělení může být výkonnostní benefit malý, pokud vůbec nějaký. Každá tato struktura k tomuto dělení přistupuje jiným způsobem a nedá se jednoduše říct, že některá je vždy ideální, jelikož v různých situacích může být vhodná jiná z nich.

Nejjednodušší akcelerační strukturou může být uniformní mřížka, která jak název napovídá dělí primitiva do rovnoměrné n-rozměrné mřížky. Výhodou takového přístupu je poměrně snadné sestavení, které lze díky deterministickému rozdělení dobře paralelizovat, navíc průchod strukturou je poměrně snadnou záležitostí. Nevýhodou, která však podstatně sráží přínosy, je špatná adaptace na danou geometrii, což ve výsledku degraduje obecnou kvalitu a nemusí ani přinést výraznější urychlení výpočtu.

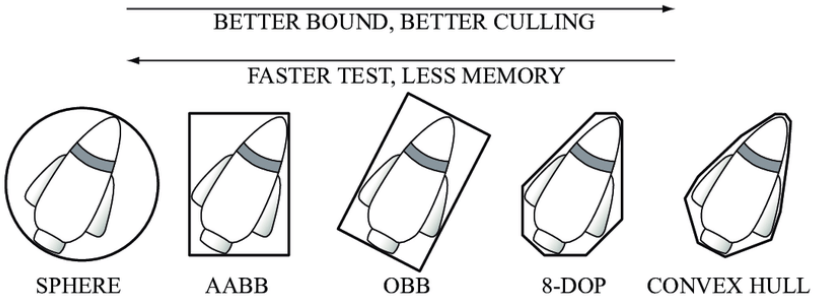
Poněkud lepší variantou je Octree, který se mnohem lépe dokáže adaptovat určené geometrii. Princip je zde prostý, pokud některý z uzlů obsahuje geometrii je rozdělen uniformně ve všech osách na 8 poduzlů, a takto se rekurzivně pokračuje do určité hloubky. Tento přístup vylepšuje uniformní mřížku, jelikož se dokáže lépe přizpůsobit scéně. Sestavení je opět relativně snadnou operací, kdy je daný uzel rozdělen vždy v polovině v každé ose. Přesto však lze říci, že adaptabilita této struktury na scénu není vždy ideální. V případech, kde je velké množství primitiv lokalizováno v určité oblasti, dochází k situaci, ve které dosahuje část stromu mnohem větší hloubky než ostatní oblasti. Toto může vést k mírné degradaci výkonu.

Za nejlepší přístup se mohou považovat hierarchie obalových těles a k-d stromy. Zde existuje řada strategií, jakým způsobem je prostor (respektive geometrie) dělený. Obě struktury se dokáží lepším způsobem přizpůsobit dané scéně než předchozí zmíněné, což je činí výhodnějšími. Detailnějšímu popisu hierarchie obalových těles se věnuje následující sekce.

[2]Repozitář projektu nanoRT: https://github.com/lighttransport/nanort.

4.2 Hierarchie obalových těles

Hierarchie obalových těles [13], jak již název napovídá, obaluje geometrii scény do obalových těles, které organizuje do stromové hierarchie. S tímto konceptem přišli Kay a Kajiya koncem 80. let, přičemž svojí prací navazovali na článek od Whitteda a Rubina z roku 1980. Ti v článku představují nejen koncept struktury, ale zároveň také obalové těleso, které obaluje geometrii pomocí několika rovnoběžných rovin. Obecně se však dá říct, že je možné použít libovolné obalové těleso (nebo dokonce i různá obalová tělesa v rámci jedné struktury), které bude v dané aplikaci nejvhodnější. Ilustrace obalových těles s jejich vlastnostmi lze vidět na obrázku 4.2.



Obrázek 4.2: Ukázka příkladů obalových těles. Vlevo jsou tělesa, jenž jsou snadno použitelná při konstrukci, mají nízké nároky na paměť a zároveň se snadno testují na průsečík s polopřímkou. Hůře však obalují danou geometrii, což není problémem u těles na pravé straně, ale právě průsečík s polopřímkou se testuje obtížněji. Obvykle se používají spíše koule nebo AABB. Převzato z webu[4].

Fundamentální myšlenkou struktury, která byla představena ve zmíněném článku, je obalování geometrických celků do jejich minimálního obalového tělesa. Blízká tělesa jsou seskupena a opět obalena do minimálního obalového tělesa, a takto se rekurzivně pokračuje, dokud není zpracovaná celá scéna. Výsledkem je potom stromová hierarchie obalových těles, přičemž listové uzly obsahují samotnou geometrii. Tato myšlenka však naráží na problém, konkrétně při seskupení blízkých těles, kdy při nevhodné volbě může dojít k zásadní degradaci celé struktury. Součástí onoho článku však je i navrženo možné řešení, konkrétně při použití top-down strategie navrhují rozdělit dané obalové těleso a vložit daný geometrický celek na základě jeho středového bodu do příslušné části, takto opět rekurzivně pokračovat. Dále se mohou objevit otázky na kolik podčástí konkrétní uzel rozdělit, případně kde jej rozdělit. Uzly bývají obvykle děleny na 2 podčásti, avšak nemusí to tak být vždy (je možné rozdělit je například na 4 podčásti, což vede k mělčí stromové struktuře), co se týče výběru místa dělení, tak zde je již řada strategií, avšak tímto a obecně stavbou hierarchie se zabývá následující sekce.

Obecně lze říci, že se hierarchie obalových těles poměrně slušně adaptuje na danou geometrii a může tak významným způsobem akcelerovat algoritmy jako například sledování paprsku. Určité nevýhody (především při procházení strukturou) mohou být překryvy jednotlivých obalových těles v rámci jedné úrovně, existují však přístupy k jejich snížení či dokonce eliminaci. Dále může být na škodu (vzhledem k binárnímu dělení) výsledná výška

[4] https://www.researchgate.net/figure/Bounding-volumes-sphere-axis-aligned-bounding-box-AABB-oriented-bounding-box_fig9_272093426

struktury, avšak při zvolení dobré strategie dělení (popřípadě vyššího dělícího faktoru) lze dosáhnout nižších hodnot.

4.3 Strategie stavby

Tématem této sekce bude představení přístupů stavby hierearchie obalových těles. Nejprve budou probrány obecné principy konstrukce hierarchie, následovat bude představení přístupů výstavby této struktury na grafických jednotkách, tedy možnosti jejich paralelizace a tím i získané akcelerace.

4.3.1 Obecné strategie stavby

Hierarchii obalových těles je možné stavit jako i jiné struktury buď strategií top-down nebo bottom up. Nedá se říct, která je vyloženě výhodnější, zde záleží na konkrétní situaci. Obecně top-down strategie výstavby sestává z následujících bodů:

1. Seřazení geometrických entit podle některé z os
2. Nalezení bodu rozdělení
3. Rozdělení geometrických entit a nalezení jejich minimálního obalového tělesa
4. Rekurzivní pokračování bodem 1, dokud není dosaženo určité výšky (případně primitiv v uzlu)

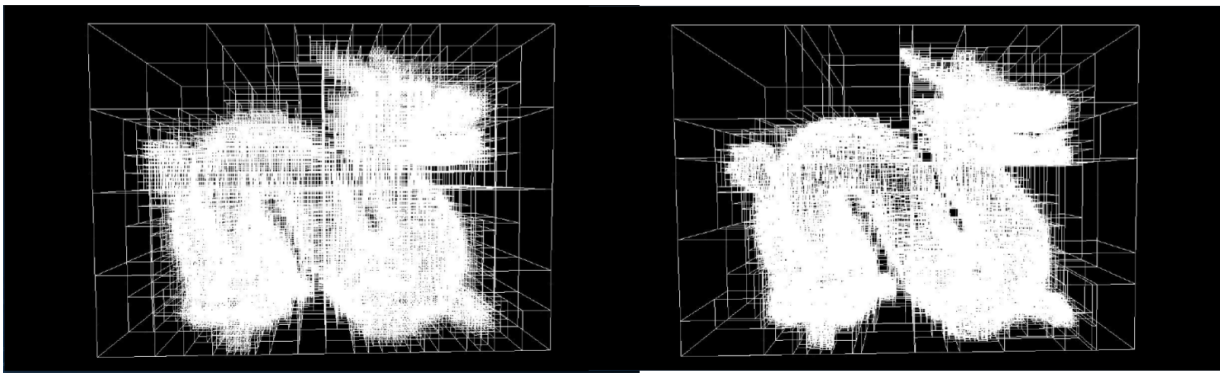
Z hlediska rychlosti stavby mohou být zajímavé první tři body, přičemž nejvýznamnější vliv hraje obvykle seřazení primitiv (respektive jejich středových bodů). Naopak z hlediska kvality výsledné struktury je důležitý bod 3 (obecně spíše volba obalového tělesa) a především bod 2. Ten se podílí rozhodujícím způsobem na tom, jak kvalitně bude geometrie obalená a jaké bude získané zrychlení. Zde existuje celá řada strategií, nejjednodušším způsobem je rozdělení tělesa v polovině, což však nemusí být vždy optimální, zvláště v případech, kdy je velká část primitiv nashromážděná v menší oblasti. Lepším řešením může být dělení tělesa pomocí mediánu, které opět nemusí být za všech okolností nejlepší volbou. V současné době poskytuje asi nejoptimálnější řešení využití metriky SAH [15] (Surface Area Heuristic) představená MacDonaldem a Boothem, která se mimo jiné používá při hledání dělící roviny v kd-stromech. Tato metrika je popsána rovnicí 4.1.

$$cost = \frac{C_i \cdot \sum_{i=1}^{N_i} SA(i) + C_l \cdot \sum_{i=1}^{N_l} SA(l) + C_o \cdot \sum_{l=1}^{N_l} SA(l) \cdot N(l)}{SA(root)} \tag{4.1}$$

Proměnná C_i značí cenu průchodu k tomuto uzlu, C_l cenu průchodu listovým uzlem, C_o cena výpočtu průsečíku paprsku a primitiva, $SA(i)$, $SA(l)$ a $SA(root)$ jsou povrchy uzlu, listového uzlu a kořenového uzlu. N_i, N_l určuje počet uzlů respektive listových uzlů a $N(l)$ počet primitiv v listovém uzlu.

Při pohledu na rovnici je možné si všimnout, že nalezení optimálního bodu dělení může být poměrně složité vzhledem k počtu potenciálních bodů rozdělení. To je možné řešit tím, že se dělené těleso rozdělí na několik diskrétních, stejně velkých částí, nad kterými poté probíhá vyhodnocování metriky SAH a výběr nejlepšího bodu rozdělení. Porovnání rozdělení scény prostřednictvím mediánu a metriky SAH je možné vidět na obrázku 4.3.

[6] http://www.cs.uu.nl/docs/vakken/magr/2016-2017/slides/lecture%2003%20-%20the%20perfect%20BVH.pdf



Obrázek 4.3: Srovnání výsledného BVH nad danou geometrií při dělení s využití mediánu (vlevo) a SAH (vpravo). Lze vidět, že SAH se lépe adaptuje na scénu. Tento fakt se nejvíce projeví u paprsků, jenž těsně minou geometrii. Ty budou složitě procházet hustou strukturou, přičemž průsečík nenaleznou. Převzato z webu[6].

Problémem, nikoli metriky samotné, ale obecně hierarchie obalových těles je, že se jednotlivá tělesa mohou překrývat, a to v extrémních případech zcela. To se ukazuje jako problém při procházení, kdy je potřeba mnohem více testování průsečíků, jak s uzly samotnými, tak i s primitivy. Tento problém do jisté míry řešení tzv. SBVH [19], který rovněž využívá metriku SAH, avšak trochu mění koncept samotné struktury. U BVH platí, že každé primitivum náleží vždy jednomu uzlu, to však nemusí platit u SBVH, který rozšiřuje základní koncept BVH využívající metriku SAH o možnost prostorového dělení. To umožňuje primitivum vložit do obou obalových těles, díky čemuž poté nevzikají jejich překryvy. Obecně tento algoritmus nejprve vyhodnotí klasický SAH s dělením podle objektů, poté vyhodnotí SAH s prostorovým dělením (přičemž bere v úvahu jaká část primitiva zasahuje do jednoho z těles), výsledky obou porovná a vybere výhodnějšího kandidáta. Celkově se jedná o způsob kvalitnější výstavby BVH, avšak daní je vyšší časová náročnost a do určité míry i zvýšení paměťové náročnosti.

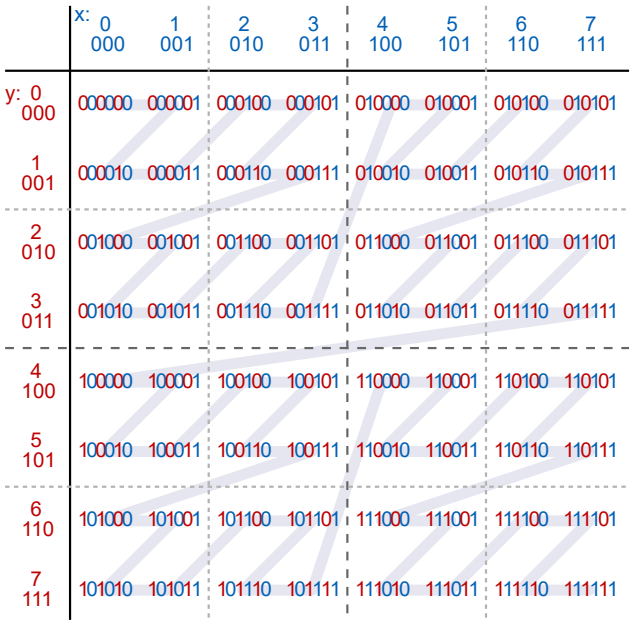
4.3.2 Strategie stavby na GPU

Výše zmíněné přístupy již umožňují výstavbu kvalitních struktur zajišťujících dobré zrychlení aplikace. Slabinou těchto přístupů je však jejich obtížná paralelizace. Akcelerace výpočtů pomocí GPU je v současné době využívaným trendem a může dávat smysl i v této oblasti, kdy pokud chceme pomocí sledování paprsku zobrazovat nějakou dynamickou scénu je téměř klíčové postavit akcelerační strukturu co možná nejrychleji.

Právě problém špatné paralelizace se zde ukazuje jako limit, jelikož pro akceleraci na GPU je nutné vykonávat práci v co možná nejvyšším počtu vláken. V opačném případě se jedná o mrhání výkonu (nedostatečné využití výpočetních jednotek grafického čipu) a výsledný čas stavby by byl ještě vyšší. Předpokládejme, že používáme strategii top-down pro stavbu, vždy začínáme s jedním uzlem, tedy s jedním vláknem. Vzhledem k dosud neznámému rozdělení nemůžeme počítat další uzly, ale počkat, až bude toto rozděleno. V další úrovni mohou být spuštěna dvě vlákna atd. Rozumný počet vláken bude tedy dosažen

až někde okolo 10. úrovně (paralelizace na GPU dává smysl až při užití přibližně alespoň tisíců vláken), což je velmi nevýhodné. Pro lepší způsob paralelizace je nutné změnit způsob myšlení, který dokáže lépe využít výkon grafického čipu.

Jedním z prvních přístupů byl představen v článku [14] Christiana Lauterbacha. Fundamentální myšlenkou tohoto přístupu je seřadit primitiva (respektive jejich centra) do jednoho seznamu a ten rekurzivně dělit. Po dosažení určité hloubky jsou jednotlivé uzly děleny podle již známé metriky SAH. Důvodem, proč není použitá metrika SAH od kořenového uzlu, je právě nízký stupeň paralelizace, který se ukazuje jako výhodnější až v nižších úrovních. Naopak důvod, proč není použit lineární přístup, je vcelku logický, a tím je nižší kvalita výsledné struktury. Tento hybridní přístup tedy nabízí dobrý poměr mezi rychlostí stavby a zároveň kvalitou struktury. Nyní však detailněji k tomuto algoritmu. Prvním krokem je tedy seřazení primitiv. Zde je však otázkou podle kterého klíče bude řazení probíhat, jelikož v počítačové grafice jsou body obvykle popsány trojicí. V tomto případě se obykle používá tzv. Mortonův kód [18]. Ten byl původně využit pro popis zeměpisných lokací. Každý bod v trojrozměrném (obecně v n-rozměrném) prostoru tedy lze popsat konkrétním kódem. Ukázka principu mortonova kódu i s popisem je na obrázku 4.4.



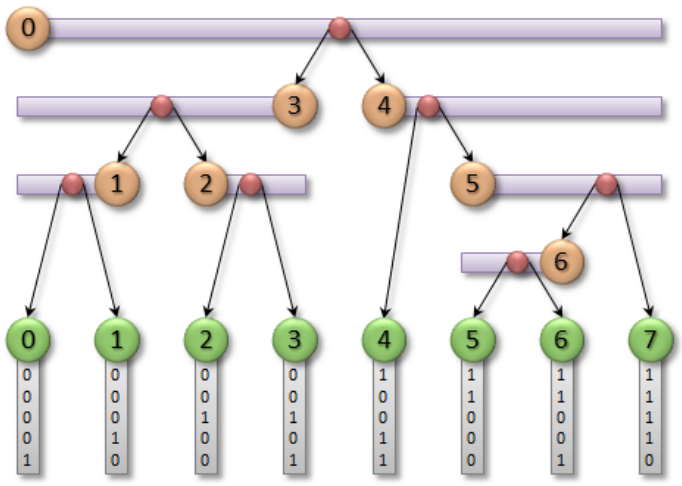
Obrázek 4.4: Ukázka principu Mortonova kódu v dvojrozměrném prostoru. Jak jde vidět, nejvýznamnější bit kódu je také nejvýznamější bit souřadnice v ose y, druhý nejvýznamnější bit kódu je zase nejvýznamnější bit souřadnice v ose x. Takto se pokračuje až po nejméně významné bity. Tento kód bývá zároveň označován jako Z order, jelikož, jak je možné si všimnout, vyplňuje prostor pomocí křivek ve tvaru Z. Tento princip tedy lze využít i do vícerozmněrných prostorů. Pořadí jednotlivých bitů podle os je ilustrační, obecně je možné použít libovolně zvolené pořadí. Převzato z webu[8].

Získané kódy tedy můžeme bez větších problémů seřadit. V tomto případě je opět vhodné použít nějaký dobře paralelizovatelný algoritmus. Využít zde například klasický quick sort by nemuselo být výhodné, lepší alternativou je využití paralelizované verze radix

[8] https://en.wikipedia.org/wiki/Z-order_curve#/media/File:Z-curve.svg

sortu. Princip tohoto algoritmu je takovýto, nejprve je každá hodnota vydělená zvoleným základem (vhodné je využití hodnot se základem 2). Podle výsledku inkrementujeme patřičnou hodnotu v pomocném poli. V dalším průchodu na základě hodnot v tomto poli jednotlivé prvky přeorganizujeme. Následně se základ zvýší o jeden řád a takto se pokračuje až k nejvyššímu řádu. Zpět ale k algoritmu stavby BVH, nyní je již vyřešená část reprezentace a seřazení primitiv v prostoru, dělení v nejvyšších úrovních BVH není problémem, zajímavější je však poté dělení metrikou SAH. K tomu je použita metoda breadth first, tedy nejprve jsou zpracovávány všechny uzly v jedné úrovni, poté v následující a tak dále. Vzhledem ke skutečnosti, že každý uzel je možné zpracovat nezávisle na druhém, je možná snadná paralelizace.

Lepší metodu [12], která je lépe paralelizovatelná, představil Tero Karras z nVidie. Základní myšlenka zůstává, nejprve se vypočítají mortonovy kódy pro jednotlivá primitiva (jejich centra) a ty jsou poté seřazeny (opět paralelním radix sortem). Strategie dělení je už však jiná a právě zde se projevuje vyšší paralelizační potenciál. Výhodnou vlastností je, že známe počet uzlů hierarchie, který je vždy $N - 1$, kdy N je počet primitiv (jedná se o vlastnost redix tree). To je právě zásadní výhoda oproti předchozí metodě, kdy nebyl známý počet uzlů a zároveň byly uzly závislé na výsledku svých předků. Primitiva jsou dělena do tzv. radix tree (ilustrován na obrázku 4.5). Uzly jsou děleny na základě rozílu v bitech Mortonova kódu. V nejvyšších úrovních hierarchie se dělí v místech, kde je rozdíl v nejvýznamnějších bitech, kdežto u nejnižších úrovní se dělí na pozicích, kde jsou rozdíly až u nejméně významných bitů. Zásadní výhodou tohoto přístupu však je, že je možné zpracovávat všechny uzly současně, nezávisle na sobě. Tato metoda do určité míry přináší změny struktury, konkrétně listové uzly nemusí nutně obsahovat pouze odkazy na primitiva, ale také mohou odkazovat mimo primitiv i na uzel potomka. Obecně tento přístup přináší velký výkonnostní potenciál, kdy je možné každou fázi algoritmu (výpočet mortonových kódů, jejich seřazení a sestavení radix tree) poměrně dobře paralelizovat.



Obrázek 4.5: Ilustrace radix tree. Jednotlivé body rozdělení uzlů jsou určeny na základě rozdílů bitů v Mortonově kódu primitiv. Nejvyšší úrovně jsou děleny v místech rozdílů nejvýznamnějších bitů (v tomto případě je například uzel 0 rozdělem v místě diference nejvýznamnějších bitů, tedy primitiva 0-3 budou v jedné části a primitiva 4-7 zase ve druhé). Získaný strom poté odpovídá výsledné BVH struktuře. Převzato z webu[10].

Dalším možným přístupem ke stavbě hierarchie obalových těles na GPU může být využití shlukování (nebo-li Clustering). Takový způsob implementuje metoda PLOC (Parallel Locally-Ordered Clustering) [16]. První kroky jsou podobné jako u předchozí metody, tedy získání center trojúhelníků, spočítání jejich Mortonových kódů a poté seřazení těchto kódů. Stavba struktury probíhá postupným shlukováním trojúhelníků do skupin. Při shlukování do skupin se hledá nejbližší soused (tzv. nearest neighbour). Vzhledem k faktu, že je celá sekvence Mortonových kódů seřazena, se může zdát, že nejbližší soused je vždy trojúhelník sousedního kódu. To však nemusí být nutně pravda, nejbližší soused může být v sekvenci od současného vzdálenější. Z tohoto důvodu probíhá hledání nejbližšího souseda v určitém, předem stanoveném okolí. Čím větší je zvoleno okolí tím je poté pravděpodobnější, že bude nalezen skutečný nejbližší soused, přičemž tento fakt se může pozitivně projevit na kvalitě struktury. V opačném případě nemusí být nalezen nejbližší soused, ale pouze jeho aproximace, avšak sníží se nároky na výkon při stavbě. Shlukování probíhá nejprve na úrovni trojúhelníků, a poté se postupuje sluhováním uzlů až ke kořenovému.

4.4 Algoritmy průchodu

Pokud je již nad danou scénou sestavená hierarchie obalových těles, přichází na řadu její využití, tedy její procházení a hledání primitiva se kterým by určený paprsek mohl kolidovat. Jedná se svým způsobem o kritickou operaci, která určuje výslednou výkonnost apikace, která hierarchii využívá (samozřejmě za předpokladu dobře postavené struktury). Při binárním dělení může struktura připomínat binární vyhledávací strom, který je možné rekurzivně procházet různými strategiemi. Svým způsobem je toto možné, princip průchodu a hledání kolize tkví v průchodu všech uzlů hierarchie, přičemž jsou navštíveny uzly kterými prochází, v případě sledování paprsku, určený paprsek, jinak je uzel zamítnut a není navštíven on ani jeho podstromy. V průběhu prohledávání je logicky možné najít více primitiv, s nimiž může paprsek kolidovat, avšak důležité je pouze to nejbližší. Kolizi je možné najít v jednom uzlu, avšak v jiném uzlu, který je procházen později je nalezena bližší kolize, to je důvod nutnosti průchodu celé struktury. V některých případech stačí pouze informace, zda došlo ke kolizi (například při výpočtu stínů), zde není nutné hledat nejbližší kolizi, ale pouze otestovat přítomnost jakékoliv kolize. Jindy je však zapotřebí právě primitivum s kterým paprsek koliduje nejblíže svému počátku (například při řešení viditelnosti). Obecný algoritmus průchodu pro N-nární hierarchii sestává z následujících bodů:

1. Vezmi aktuální uzel na vrcholu zásobníku
2. V případě listového uzlu testuj případnou kolizy s primitivy v daném uzlu
3. Najdi průsečíky s obalovými tělesy všech N potomků
4. Seřaď uzly potomků podle vzdálenosti jejich průsečíků
5. Vlož uzly na zásobník podle vzdálenosti průsečíku (nejbližší je vložen na vrchol), přičemž uzly které průsečík nenalezly nejsou na zásobník vloženy
6. Pokračuj bodem 1

Využití rekurzivního procházení je tedy vpodstatě možné, avšak přináší několik problémů. Prvním problémem je, že tento způsob procházení nemusí být efektivní, další vadou tohoto

[10] https://devblogs.nvidia.com/wp-content/uploads/2012/11/fig06-numbering.png

přístupu je nemožnost využití rekurze na grafickém čipu. Je samozřejmě možné rekurzi simulovat pomocí iterační verze a udržovat si zásobník ve vlastní režii, avšak ani to není optimální cesta. Na toto téma tedy vznikla řada článků s cílem nalézt lepší řešení této problematiky.

Otázkou je tedy, jaké je možné východisko. Při průchodu hierarchie obalových těles na GPU se ukazuje jako nejlepší varianta iterační bezzásobníkové algoritmy. V případě nemožnosti využití rekurze na grafickém čipu je iterační verze logickou volbou, avšak i v případě, kdy by bylo možné rekurzi využít by patrně docházelo k velké divergenci vláken, a tudíž by se i tak ukázala iterační verze jako výhodnější. Nevyužití zásobníku sice nemusí na první pohled být logické, jenže na GPU smysl bezpochyby má. Pokud na GPU provádíme nějakou úlohu, snažíme se pokud možno minimalizovat počet využitých registrů na vlákno, pokud vláknu nestačí zdroje na čipu, jsou hodnoty registrů odkládány do lokální paměti mimo čip, což může být překážka výkonu. Pokud zvážíme množství registrů, které by mohl takový zásobník spotřebovat, můžeme vidět potenciální problém. Dnes již však existují přístupy umožňující obejít nutnost použití zásobníku při průchodu BVH. Jeden takový možný přístup je algoritmus [8] využívající jednoduchou stavovou logiku. Celá filozofie tohoto algoritmu je poměrně jednoduchá, hierarchie je procházena iteračně, přičemž jediné, co je potřeba si pamatovat, je informace o stavu. Celý zásobník je tedy nahrazen jedinou proměnnou, která může nabývat tří hodnot. Tato proměnná popisuje odkud se do současného uzlu přišlo (od rodiče, potomka nebo uzlu ze stejné úrovně). Pomocí stavové logiky pak jdou popsat všechny případy, které mohou při průchodu strukturou nastat. Jediné, co je ještě potřeba, je dodat každému uzlu informaci o předkovi, ta slouží při zpětném průchodu do vyšších úrovní hierarchie. S určitým problémem se však tento přístup potýká. Vzhledem ke stavovému řízení obsahuje algoritmus mnoho větví, což může opět vést na divergenci vláken. Toho si jsou však autoři vědomi, a taktéž představují variantu téhož algoritmu, jenž obsahuje méně větvení a může se tedy jevit jako výhodnější.

Dalším možným přístupem může být algoritmus [2] využívající bitový zásobník. Článek, jenž popisuje tento algoritmus, mimo jiné poukazuje i na jistou slabinu předchozího přístupu, a to, že může docházet k vícenásobnému testování kolize paprsku a obalového tělesa pro jeden uzel. Tento přístup supluje celý zásobník jednou 32 bitovou popřípadě 64 bitovou proměnnou. Každá úroveň potom zabírá $N-1$ bitů na zásobníku, přičemž N značí stupeň dělení hierarchie. Sémantika bitů je potom následující, pokud je bit nulový, znamená to, že nemá vstupovat do sourozeneckého uzlu (byl již navštíven nebo nedošlo k průsečíku jeho obalového tělesa a testovaného paprsku), ale jít do rodičovského uzlu. V případě, kdy má bit hodnotu jedna, znamená to, že se bude pokračovat uzlem sourozence. Algoritmus končí, pokud je aktualní uzel kořenový a hodnota bitu na vrcholu zásobníku je nula. Aby mohla být tato metoda použita, je však zapotřebí několik úprav ve struktuře BVH. Každý uzel musí mít odkaz na svého předka a zároveň musí obsahovat odkaz na svého sourozence (respektive své sourozence). Celkově vzato má algoritmus podobné paměťové nároky jako předchozí metoda, zároveň článek uvádí implemntaci této metody vhodnou pro GPU architektury s menším množstvím větvení.

Kapitola 5

Návrh aplikace

Cílem této kapitoly bude představit vlastní návrh implementační části práce, a to konkrétně zvolené postupy, přístup k jejich implementaci a technologie, které budou při vývoji použity. První částí bude popis knihovní části pro GPUEngine, tedy návrh implementace akceleračních struktur, dále bude následovat popis návrhu ray tracing aplikace, která bude primárně sloužit pro zhodnocení kvality implementace akceleračních struktur.

5.1 Použité technologie

V této sekci budou popsány technologie využité při implementaci aplikace, jež mají nejdůležitější vliv na návrh implementovaných částí.

5.1.1 OpenGL

OpenGL je knihovna sloužící pro akceleraci vykreslování na GPU. Obecně bývá využívána pro práci s 3D grafikou, avšak je možné její využití i v případě 2D grafiky. Jedná se o platformně nezávislou knihovnu. V současnosti je spravována konsorciem Khronos. Bývá využívána například v CAD aplikacích či v počítačových hrách.

Její koncepce je založena na vykreslovacím řetězci podobně jako je tomu i u jiných API pro 3D grafiku (Vulkan, Direct3D apod.). Na jeho vstupu jsou data, která mají být vykreslena (souřadnice vrcholů geometrie) a na výstupu vyrenderovaný snímek. Řetězec je možné rozdělit na část vektorovou a rastrovou, přičemž pomyslný hraniční bod tvoří rasterizace. Řetězec se skládá z fixních a programovatelných částí (tzv. shadery), jež je možné programovat ve speciálním programovacím jazyce GLSL. Těchto shaderů se v řetězci nachází pět typů, přičemž první čtyři pracují s geometrií a poslední již pracuje se samotnými fragmenty.

Mimo vykreslovcí řetězec obsahuje OpenGL také část pro obecné výpočty na grafické jednotce (tzv. GPGPU). K tomuto účelu je zde speciální typ shaderů tzv. compute shader. Ten je možné programovat podobně jako ostatní shadery v jazyce GLSL. Rozdílem však je, že nemá pevně definované vstupy a výstupy jako tomu bylo u shaderů ve vykreslovací části. Vstupem může být určitý druh bufferu nebo například textura, výstupem může být rovněž buffer nebo image objekt (v podstatě se jedná o texturu, jenž má pouze 1 úroveň MIP mapy). Rovněž je zapotřebí definovat, v jakém formátu jsou uložena vstupní respektive výstupní data. Shaderu je dále potřeba sdělit velikost lokální pracovní skupiny (jejíž význam byl vysvětlen v kapitole o GPU). Veškerou synchronizaci je potřeba zajišťovat explicitně a to buď pomocí bariér (bariéry v rámci shaderu slouží pouze k synchronizaci v rámci pracovní

skupiny) nebo k tomu využít atomických operací, případně použít bariéru ze strany CPU pro synchronizaci celé úlohy. Využití výpočetní části může být různé, lze zde například předpočítat stíny statických objektů, simulovat pohyb částic nebo počítat globální osvětlení pomocí různých metod jakými jsou ray tracing nebo radiozita.

5.1.2 GPUEngine

Jedná se o knihovnu[1] implementovanou v jazyce C++, jejíž vývoj probíhá na FIT VUT. Jejím primárním účelem je poskytnout prostředky pro zpracování a vykreslování 3D grafiky prostřednictvím GPU. Obsahuje objektové rozhraní nad OpenGL, dále obsahuje třídy pro práci s grafem scény, implementaci kamery ad. Dále implementuje třídu pro načtení scény založenou na knihovně Assimp, případně také třídy pro práci s oknem aplikace, správou vstupů a výstupů, jenž je založeno na knihovně SDL. Knihovna je nadále ve vývoji, avšak pro určité aplikace je již možné její plnohodnotné využití.

5.1.3 GLM

GLM[2] (neboli OpenGL Mathematics) je knihovna pro podporu matematických struktur a operací běžně využívaných v počítačové grafice. Obsahuje datové typy jako n-rozměrný vektor (o maximální velikosti 4) nebo n-rozměrné matice. Zároveň podporuje operace nad těmito typy, jež jsou běžně užívány v počítačové grafice, jako je skalární součin, vektorový součin, transpozice matice nebo inverze matice a řada dalších. Dále obsahuje také podporu pro práci s kvaterniony. K akceleraci těchto operací využívá knihovna techniku vektorizace (pomocí instrukčních souborů AVX, SSE). Datové typy definované v této knihovně jsou kompatibilní s datovými typy používanými v GLSL, navíc používá stejné konvence názvů.

5.2 Návrh knihovní části

Tato sekce popisuje způsob návrhu implementace akceleračních struktur, jenž bude přidána do knihovny GPUEngine. Nejprve bude popsán přístup implementovaný na CPU, poté způsob návrhu GPU implementace.

5.2.1 Implementace na CPU

V rámci teoretické části byly představeny různé možné přístupy ke stavbě hierarchie obalových těles. Tento fakt jako takový je tedy potřeba zohlednit v rámci návrhu knihovní části. Je tedy nutné brát zřetel na fakta, že BVH struktura může být stavěna různými strategiemi s použitím různých druhů obalových těles. Z tohoto plyne, že je nutností, aby návrh poskytoval rozšiřitelnost o další možné strategie. V rámci této práce bude implementována stavba BVH s využitím heuristiky SAH a jako obalové těleso zvoleno AABB. Co se týče výběru tělesa, tak AABB poskytuje rozumný kompromis mezi obalením přidělené geometrie a složitostí sestavení respektive testování na průsečík s paprskem. Heuritika SAH je zvolena jakožto způsob, který se snaží dělit scénu tak, aby byla pravděpodobnost zásahu primitiva v obou částech podobná, tedy měla by poskytovat kvalitní způsob dělení scény a tím pádem i dobrou výslednou akceleraci.

[1] Oficiální repozitář projektu: https://github.com/Rendering-FIT/GPUEngine
[2] Oficiální webové stránky knihovny: https://glm.g-truc.net

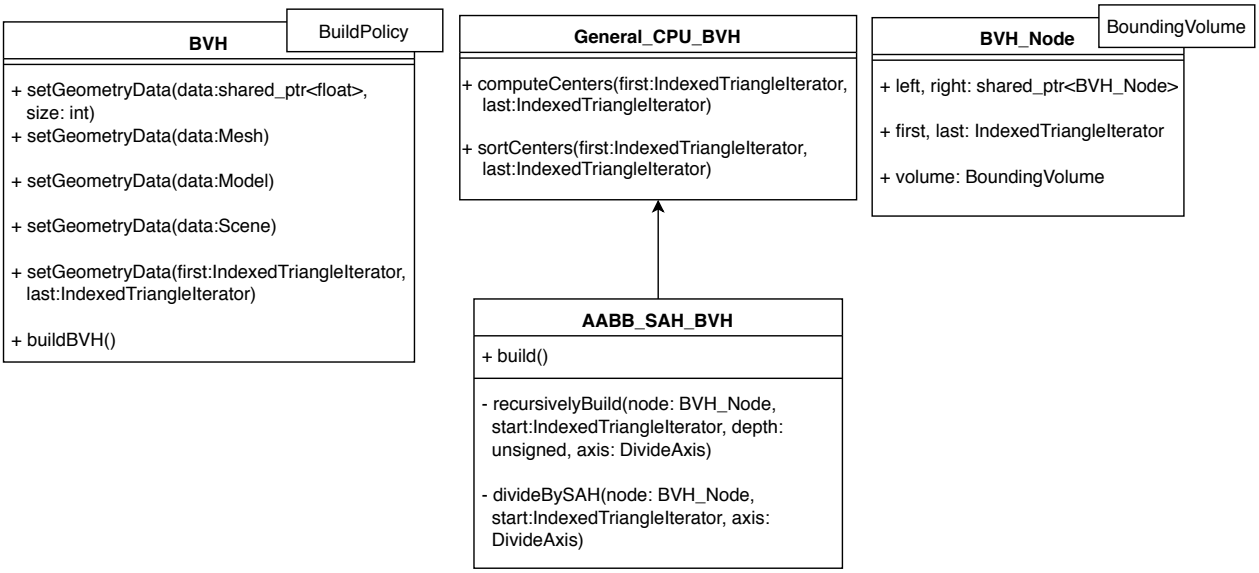
Vstupem pro sestavení akcelerační struktury obvykle bývají údaje o geometrii scény. V případě, kdy nebudou brány v potaz struktury přímo z GPUEngine, lze chápat geometrii scény jako vektor float hodnot, které reprezentují souřadnice bodů. Tento základní popis scény však obvykle nedostačuje vzhledem k faktu, že je scéna logicky členěna na geometrické entity, přičemž u každé je možné uvažovat řadu atributů. Kromě již zmíněných souřadnic také například normálové vektory nebo texturovací souřadnice. Z těchto důvodu dává smysl uvažovat jako vstup mimo vektor float hodnot také struktury z knihovny GPUEngine a to konkrétně třídu `ge::sg::Mesh`, což je abstrakce geometrické entity (útvaru) v rámci modelu, pak právě samotného modelu (tedy třída `ge::sg::Model`) a kompletní scény (třída `ge::sg::Scene`). Pro samotnou práci s primitivy v rámci těchto struktur poslouží třída `ge::sg::IndexedTriangleIterator`, která umožňuje práci jak s indexovanou geometrií, tak i s neindexovanou (případ vektoru float hodnot).

Nyní již k samotnému návrhu implementace. Vzhledem k povaze této části je možné využít tzv. přístup Policy-based design [3], který byl představen v knize Andreie Alexandrescua Modern C++ design. Tento přístup využívá principu šablon v C++ (tzv. templates). Vystupuje zde ustřední třída zvaná host class, která slouží jako určité rozhraní. Ta je zároveň parametrizovatelná, přičemž jako parametr je nastavena určitá politika (policy). Host class tedy zapouzdřuje implementaci konkrétní politiky tím, že obstarává volání funkcí a předávání parametrů konkrétní nastavené politiky. V tomto případě si lze představit, že host class bude představovat obecnou třídu pro práci s BVH, přičemž jednotlivé politiky budou implementovat různé strategie stavby této struktury. Níže na obrázku 5.1 je zobrazen a detailněji popsán diagram tříd této části.

5.2.2 Implementace na GPU

Podobná fakta jako platila u předchozí části platí i zde, tedy existuje řada různých technik, jakým způsobem je možné implementovat stavbu hierarchie obalových těles na GPU. Opět je tedy nutné navrhnout jednotlivé třídy tím způsobem, aby bylo možné jejich využití pro jiné přístupy. Do určité míry se lze inspirovat z předchozí části, jelikož se dá očekávat, že vstupní data budou stejného charakteru. I zde může být vhodné využití Policy-based designu zmíněného dříve. Opět poslouží třída `BVH`, která nebude sloužit jen pro práci se strukturami BVH vytvořenými na CPU, ale bude poskytovat univerzální rozhraní pro jakoukoliv práci s BVH, přičemž bude parametrizována konkrétní třídou, která bude zajišťovat implementaci stavby hierarchie. V rámci této práce bude konkrétně implementována technika od Tero Karrase (viz [12]) založená na struktuře radix tree využívající mortonovy kódy center trojúhelníků. Nespornou výhodou tohoto přístupu je, jak již bylo zmíněno, možnost vysoké úrovně paralelizace procesu stavby struktury.

V této části je rovněž možné využít podobný přístup jako u té předchozí, tedy vytovřit třídu, jež bude implementovat určitou funkcionalitu, kterou bude možné využít v různých implementacích stavby BVH na GPU. V tomto případě se jedná především o výpočet mortonových kódů a jejich řazení, což jsou operace často využívané v oblasti stavby těchto struktur na GPU. Konkrétní metoda stavby pak může tuto funkcionalitu jednoduše oddědit. Je tedy možné vidět zde podobnost s návrhem z předchozí části. Objevují se zde však určité odlišnosti, ty přináší výpočetní část na straně GPU. Právě API pro práci s GPU neumožňují práci s objektově orientovaným paradigmatem a jiná další omezení, není například možné využít klasických ukazatelů (to umožňuje pouze CUDA API, avšak při použití tohoto API by byla knihovní část využitelná pouze na hardwaru od společnosti nVidia). Proto je tedy nutné udržovat celou strukturu jako buffer v grafické paměti. Stejným způsobem jsou na



Obrázek 5.1: Diagram tříd knihovní části aplikace. Třída `BVH` má za úkol poskytovat určité společné rozhraní pro práci s touto akcelerační strukturou (z pohledu policy-based design se jedná o host class). Jak je naznačeno, třída je parametrizovaná a to konkrétně typem strategie stavby. Dále zde figuruje třída `General_CPU_BVH`, jež implementuje společné operace užívané při stavbě BVH. Třída `AABB_SAH_BVH`, která dědí od `General_CPU_BVH` implementuje již konkrétní strategii výstavby BVH struktury, přičemž využívá operace definované v její nadtřídě. Poslední je zde třída `BVH_Node`, která zapouzdřuje data uzlu, přičemž je parametrizovaná typem obalového tělesa uzlu. Zdroj: vlastní.

grafickou kartu předány i údaje o geometrii (souřadnice vrcholů). Tento přístup nemusí být na škodu při použití této akcelerační struktury přímo na GPU, kdy je možné přímé využití bez nějakých nutných transformací. Problém může být při použití na straně CPU, v tomto případě je zapotřebí implementovat konverzi bufferu z GPU na nelineární BVH strukturu, která bude uložena do operační paměti. Na závěr je taky nutné poznačit, že se jedná o jiný typ hierarchie než ve standardním případě, kdy jsou primitiva ukládána pouze do listových uzlů, což v tomto případě nemusí platit, jelikož v rámci radix tree struktury mohou uzly obsahovat jak odkaz na potomka, tak odkazovat i na některé primitivum.

5.3 Návrh Ray Tracing aplikace

Koncepce této části bude odlišnější narozdíl od předchozích částí, jež jsou navrženy jako součásti knihovny. Hlavním cílem aplikace je vykreslování zvolených scén pomocí techniky distribuovaného sledování paprsku. Výpočet algoritmu sledování paprsku a zobrazování vykerslených snímků bude akcelerováno pomocí OpenGL (s využitím compute shaderů respektive vertex a fragment shaderů). Předpokládaným vstupem aplikace je tedy soubor, přesněji řečeno soubory popisující geometrii a materiály scény. Samotné načtení scény bude obstarávat součást knihovny GPUEngine `AssimpModelLoader`, jež je založená na knihovně Assimp. Získaná data jsou předána příslušné třídě a zde jsou zpracována do podoby, aby bylo možné je přenést a zpracovávat na grafické jednotce. Data důležitá pro chod aplikace

bude udržovat třída `App`, jež bude zároveň řídit chod celé aplikace (udržování GUI, vykreslování scény, načtení scény, sestavení akcelerační struktury ad.). Tyto jednotlivé podúlohy budou zajišťovat opět příslušné třídy, třída `RayTracing` tedy bude zajišťovat řízení výpočtu snímku, jeho vykreslení, předání dat na GPU, dále třída `UserInterface` bude vykreslovat uživatelské rozhraní a udržovat si všechny potřebné proměnné.

Samostatně pak vystupují části výpočtu a zobrazení snímku, jež budou ze strany CPU prostřednictvím třídy `RayTracing` pouze načteny a volány. Jejich samotný běh bude probíhat výlučně na GPU. Konkrétní implementační detaily a použité postupy jsou blíže popsány v následující kapitole.

Kapitola 6

Implementace

Nasledující kapitola je zaměřená na popis implementačních detailů a využitých postupů při implementaci význačných částí této práce jako je stavba BVH struktury a to jak na CPU, tak i varianta akcelerovaná na GPU, dále také průchod těmito strukturami a nakonec i implementační detaily sledování paprsku.

6.1 Implementace BVH na CPU

Při výstavbě BVH struktury je využita rekurzivní strategie. Jejím zálkladem je funkce, jež se stará o sestavení uzlu a rekurzivně volá tutéž funkci pro potomky daného uzlu. Struktura této funkce je představena v algoritmu 1.

Algorithm 1 Stavba BVH

```
1:  procedure recursiveBuild(první primitivum, poslední primitivum, hloubka,
    dělící osa)
2:      nalezni minimální objemové těleso
3:      if hloubka = 0 then
4:          return
5:      if (poslední primitivum - první primitivum) < minimum v uzlu then
6:          return
7:      seřaď trojúhelníky uzlu podle dělící osy
8:      dělící pozice ← urči dělící pozici pomocí metriky SAH
9:      dělící osa ← urči novou dělící osu
10:     recursiveBuild(první primitivum, dělící pozice, hloubka-1, dělící osa)
11:     recursiveBuild(dělící pozice, poslední primitivum, hloubka-1, dělící osa)
```

Prvním krokem je naleznout minimální obalové těleso pro daný výčet trojúhelníků. V případě klasického Axis aligned bounding boxu jsou nalezeny extrémní hodnoty (tedy minimální i maximální) v rámci všech os. Následně jsou otestovány podmínky zastavující rekurzivní volání, tedy konkrétně test na dosažení maximální hloubky a minimální množství primitiv v uzlu, kdy již neprobíhá další dělení. Následně se provádí seřazení trojúhelníků v konkrétním uzlu. Dělení probíhá podle osy, jež je aktuálně zvolena jako dělící. V této implementaci probíhá řazení podle centra trojúhelníku (je možné také řadit podle barycentra případně některého z vrcholů). Důvod proč je prováděno samotné dělení je především ten, že je možné striktně oddělit jednotlivé trojúhelníky, přičemž potomkům stačí následně předat indexy případně iterátory na první a poslední trojúhelníky v uzlu. Dalším důvodem

řazení je i určitá akcelerace dělící politiky, kdy např. v případě mediánu je vybrán jako dělící prostřední prvek, v případě SAHu se zamezí zbytečnému testování některých trojúhelníků. Dále už následuje samotná dělící politika, tedy v tomto případě rozdělení pomocí heuristiky SAH. Samotné vyhodnocení této heuristiky by bylo však poměrně náročné a jak již bylo zmíněno, bývá výhodnější SAH vyhodnotit na několika situacích a následně vyhodnotit, který případ je pro dělení nejvýhodnější. Dále následuje výběr osy podle níž se bude dělit v další úrovni. Zde je možné cyklicky volit jednotlivé osy, případně vybrat osu, kde je obalové těleso nejdelší (což však může být v některých případech neefektivní). Na závěr je rekurzivně zavolána tato funkce s nově získanými parametry.

6.2 Implementace BVH na GPU

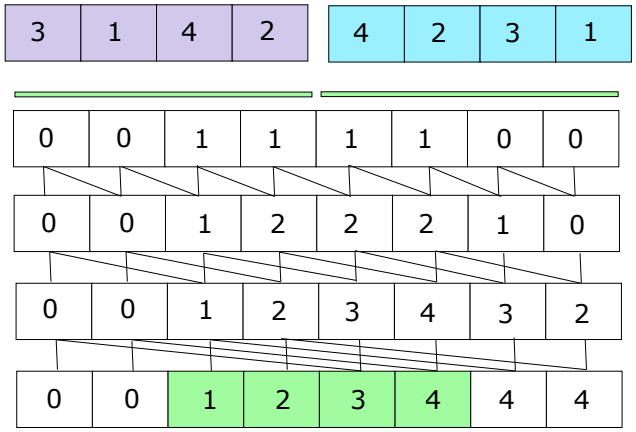
Sestavení BVH struktury na GPU bude probíhat poměrně odlišně než v předchozím případě. Samotná konstrukce je složena ze čtyř fází. První fází je výpočet mortonových kódů, ty jsou dále seřazeny a na jejich základě je postaven radixový strom, jenž poslouží jako základ BVH a nakonec jsou nad jednotlivými uzly spočítány minimální obalová tělesa.

Nejprve se tedy začne výpočtem mortonových kódů trojúhelníků respektive jejich center (popřípadě barycenter). Před samotným výpočtem probíhá normalizace souřadnic do jednotkové krychle, až následně proběhne spočítání centra a určení mortonova kódu. Tento úkon je možné provést velmi snadno v rámci jednoho kernelu a paralelně napříč všemi trojúhelníky, bez nutnosti jakýchkoliv synchronizací, jelikož každá invokace pracuje pouze s vlastními daty.

Dalším krokem je vlastní seřazení získaných dat. Zde bude využita paralelní verze klasického radix sortu. Základní idea algoritmu je podobná jako u standardní verze tohoto algoritmu, tedy nejprve se na základě konkrétního bitu (muže být brána v potaz i cifra) inkrementuje čítač pro ten daný bit, ve chvíli, kdy jsou zpracovány všechny hodnoty, jsou tyto hodnoty na základě hodnot čítačů umístěny na nové pozice. Tento postup se opakuje pro všechny bity (od nejméně významného po ten nejvíce významný). Tato implementace používá 30-ti bitové mortonovy kódy, tudíž proběhne 30 iterací algoritmu.

V rámci každé iterace, tak proběhnou 2 respektive 3 kroky, nejprve každé vlákno provede inkrementaci na příslušné pozici pole čítačů (přičemž počáteční hodnota je vždy 0). Tato pozice je určena na základě hodnoty aktuálně tesotvaného bitu a na základě pozice konkrétního kódu v rámci celého bufferu. Než přijde na řadu další krok je nutné dokončit výpočet všech vláken, z tohoto důvodu je zapotřebí využít bariéru pro všechna vlákna na straně CPU. Dalším krokem je určení nových pozic jednotlivých prvků. Tato pozice je určena jako součet všech předchozích položek v poli čítačů získaných v rámci předchozího kroku. V podstatě se jedná o typ operace známý jako paralelní redukce, kdy je na počátku aktivní maximální počet vláken, jenž je postupně redukován. V tomto případě jde konkrétně o sumu prefixů. Tato operace se provádí v několika krocích. Počet kroků je určen jako dvojkový logaritmus počtu sčítaných prvků. V každém kroku je nutná synchronizace všech vláken pomocí bariéry na straně CPU. Tuto operaci lze mírně zefektivnit tím, že budou v průběhu výpočtu postupně odpojovány celé warpy, avšak v několika posledních krocích je nutné odpojovat vlákna v rámci jediného běžícího warpu. Po dokončení výpočtu prefixové sumy je pro každý prvek určeno, kam má být vložen. Při přemisťování mortonových kódů (a s nimi také indexů trojúhelníků k nimž jsou asociovány) jsou využity dvě kopie bufferů, přičemž jeden slouží jako vstupní a druhý jako výstupní. Tyto buffery jsou po každé iteraci přehozeny, tedy vstupní za výstupní a naopak. Po provedení patřičného počtu iterací (v tomto případě tedy 30) je pole s mortonovými kódy seřazeno a s ním jsou

seřazeny i indexy patřičných trojúhelníků. Celý tento proces je detailněji zobrazen a popsán na obrázku 6.1.



Obrázek 6.1: Ilustrace principu paralelního radix sortu. Ve fialovém poli jsou vstupní hodnoty, v modrém jsou hodnoty po provedení jedné iterace algoritmu. V první iteraci se zjišťuje hodnota nejméně významného bitu. Na základě hodnoty tohoto bitu dojde k inkrementaci příslušné hodnoty v poli čítačů, jenž je zobrazen pod vlastními daty. Pole čítačů má dvojnásobnou velikost jako vstupní data. První polovina je určena pro bit s hodnotou 0, druhá pro hodnotu 1. Poté co všechny prvky provedou inkrementaci následuje suma prefixů, jejichž počet kroků je udán jako dvojkový logaritmus délky vstupu. V tomto případě je délka vstupu 8, tedy počet iterací je 3. Po výpočtu sumy prefixů mohou jednotlivé prvky naleznout pozici, kam mají být vloženy. Tato pozice je dostupná na místě v poli čítačů, kterou ten samý prvek na počátku inkrementoval (zeleně označené prvky). Poté proběhne samotné přeuspořádání do podoby zobrazené v modrém poli. Zdroj: vlastní.

Na základě získaných dat, tedy seřazených mortonových kódů, lze již stavět strukturu radix tree, jež bude sloužit jako struktura pro výsledné BVH. Jeho stavbu je možné maximálně paralelizovat, kdy je možné každý uzel provádět nezávisle na ostatních. Samotná stavba má několik základních kroků, nejprve je určen směr sekvence, kterou bude uzel pokrývat. Tento směr se určí na základě rovnice 6.1.

$$dir = sgn(\delta(n, n+1) - \delta(n, n-1)) \tag{6.1}$$

Symbol sgn značí funkci signum, δ znační funkci výpočtu délky společného prefixu, jenž se vypočítá jako exluzivní součet dvou prvků a následně se zjistí počet prefixových nul. Symboly n, $n+1$ a $n-1$ značí mortonovy kódy na n-tém respektive $n+1$ a $n-1$ indexu. Dále se určí délka dané sekvence v získaném směru. Tato délka se určí tím, že se v daném směru nalezne prvek, jehož délka společného prefixu s aktuálním prvkem je menší než délka společného prefixu s prvkem před začátkem této sekvence (jinými slovy s prvním prvkem v opačném směru). V dalším kroku se určí místo, kde proběhne rozdělení. To se nalezne prohledáváním kódu v získaném směru, přičemž daný kód musí splňovat podmínku, že délka jeho společného prefixu s prvkem ze začátku sekvence je větší než délka jeho společného

prefixu s koncem sekvence. Na základě získaných hodnot jsou na závěr určeni potomci současného uzlu.

Poslední fází stavby BVH je výpočet minimálních obalových těles pro každý uzel. Teoreticky je možné pro všechny uzly spočítat minimální obalová tělesa naráz, jelikož každý uzel ví, jaký rozsah trojúhelníků pokrývá. Tento přístup by byl však značně neefektivní, protože uzly v nejvyšších úrovních BVH pokrývají velké množství trojúhelníků. Efektivnější varianta je počítat objemová tělesa postupně od listových uzlů až ke kořenovému. Výpočet tedy začne u listových uzlů, následně jdou tato vlákna k rodičovským uzlům. Ke každému uzlu potom však přistupují dvě vlákna, což znamená, že je nutné zaručit výlučný přístup k danému uzlu. V tomto případě je využito atomických instrukcí. Po výpočtu minimálního objemového tělesa současného uzlu je zastaven výpočet prvního příchozího vlákna a k rodičovskému uzlu postupuje pouze druhé přistupující vlákno. Takto se postupuje směrem ke kořenovému uzlu, přičemž je postupně redukován počet aktivních vláken. Celý výpočet končí ve chvíli, kdy poslední běžící vlákno dopočítá minimální objemové těleso pro kořenový uzel. Po dokončení této fáze je sestavena již kompletní struktura BVH.

6.3 Implementace sledování paprsku

Sledování paprsku je z principu rekurzivní algoritmus. Jako takový je tento algoritmus velmi dobře paralelizovatelný, vzhledem k faktu, že nejsou zapotřebí různé mezivláknové synchronizace. Určitou nevýhodou však může být to, že se může jednat o úlohu, kde dochází k divergenci vláken. Hlavní však je, že obvykle API pro GPU nepodporují rekurzi (s výjimkou API CUDA), tudíž je potřeba klasický algoritmus přestavět na iterativní verzi a zásobník simulovat ve vlastní režii. Přepis iterativní verze sledování paprsku je ukázán v algoritmu 2.

Algorithm 2 Algoritmus sledování paprsku.

```
1:  procedure rayTrace(ray)
2:      pixel ← (0,0,0)
3:      while hloubka < MAX_HLOUBKA do
4:          if paprsek koliduje se scénou then
5:              spočítej osvětlení v bodě
6:              pixel ← pixel + získané osvětlení
7:          else
8:              return pixel
9:          ray.origin ← aktuální bod kolize
10:         urči nový směr paprsku podle optického prostředí
11:         ray.direction ← nový směr
12:         hloubka ← hloubka + 1
13:     return pixel
```

Vstupem algoritmu je primární paprsek sestávající z bodu počátku a směrového vektoru. Dále algoritmus operuje také s geometrií scény, která má být zobrazována. Scéna bývá obvykle reprezentována jako seznam trojúhelníků, a tento přístup je využit i zde. Hlavní část výpočtu figuruje uvnitř smyčky, jejíž terminální podmínka je určena jako dosažení maximální hloubky sledování.

Nejprve proběhne hledání bodu kolize paprsku a scény. Tento bod bude nalezen během průchodu vytvořenou BVH strukturou, jenž zajistí významnou akceleraci. Na základě výsledku testu kolize se určí následující postup. V případě nenalezení kolize je výpočet okamžitě ukončen, pokud je nalezen vypočítá se osvětlení v daném bodě s tím, že je podle prostředí s nímž paprsek kolidoval určen nový směr. Tento směr lze získat buď jako směr odrazu současného poprsku podle normály povrchu, kde kolize nastala nebo lomením paprsku podle normály povrchu.

V rámci výpočtu osvětlení se počítají i stíny a další efekty. Právě stíny se s pomocí metody sledování paprsku implementují poměrně snadno, přičemž je jejich kvalita vysoká. Vzhledem k tomu, že se jedná o distributivní sledování paprsku, je možné počítat i měkké stíny. Implementace stínů je založená na testování přítomnosti kolize mezi aktuálním bodem a zdrojem osvětlení, jinými slovy je vyslán paprsek ze současného místa směřující ke zdroji světla a následně probíhá testování, zda došlo ke kolizi. Měkké stíny se počítají také tímto způsobem, jen s malým rozdílem, že je z aktuálního bodu vysláno několik paprsků s náhodně zvoleným směrem (vybere se směr v rámci jednotkové polokoule okolo povrchové normály). Ostatní efekty jako například Depth of field nebo Ambient occlusion jsou implementovány vpodstatě stejným způsobem. Jediným problémem však zůstává, že při implementaci bude zapotřebí generátor náhodných čísel. Ten jako takový není v rámci compute shaderů dostupný, a tak je nutné si jej ve vlastní režii doimplementovat.

Výsledný snímek je ukládán do struktury Image (v podstatě se jedná o texturu s jedinou úrovní MIP-mapy). Zobrazení výsledku poté probíhá za pomoci raterizační pipeline, tedy kombinace vertex a fragment shaderu. Obrázek 6.2 zobrazuje scénou vykreslenou implementovaným ray tracerem.

Obrázek 6.2: Snímek z implementované aplikace, zobrazující lom světla a měkké stíny. Zdroj: vlastní.

6.4 Implementace průchodu BVH

Podobně jako u sledování paprsku by bylo možné provést procházení hierarchie obalových těles s využitím rekurze. I v tomto případě je nutné použít iterační algoritmus a zásobník udržovat ve vlastní režii. Jak bylo ale zmíněno, udržovat zásobník v tomto případě by mohlo být z hlediska prostorové složitosti nepříjemné. Z tohoto důvodu je v tomto místě využit bitstack algoritmus představený dříve.

Základním principem algoritmu je procházet v rámci BVH struktury ty uzly, s nimiž může testovaný paprsek kolidovat. Je nutné procházet všechny takové uzly, jelikož není známo ve kterém může docházet k nejbližší kolizi s některým primitivem. Algoritmus tedy nekončí v okamžik, kdy nalezne první kolizi s určitým primitivem, ale ve chvíli kdy projde všechny uzly jimiž prochází paprsek a až se nakonec vrátí zpět do kořenového uzlu. Jak již samotný název algoritmu napovídá, v každé úrovni struktury si pomocí bitů ukládá, zda má navštívit i druhého potomka či pokračovat v návratu směrem ke kořenovému uzlu. Pseudokód algortimu průchodu BVH je popsán v algoritmu 3.

Algorithm 3 Bitstack algortimus pro průchod BVH

```
1:  procedure bvhTraversal(ray)
2:      node ← kořenový uzel
3:      bitstack ← 0
4:      while node ≠ -1 do
5:          backtrack ← false
6:          if node je listový uzel then
7:              for každé primitivum v uzlu do
8:                  test kolize primitiva a paprsku
9:              backtrack ← true
10:         else
11:             test kolize s oběma potomky
12:             if některý potomek zasažen then
13:                 bitstack ← bitstack ≪ 1
14:                 if zasažen jeden potomek then
15:                     node ← zasažený potomek
16:                 else
17:                     node ← bližší potomek
18:                     bitstack ← bitstack ∨ 1
19:             else
20:                 backtrack ← true
21:         if backtrack = true then
22:             while bitstack ∧ 1 = 0 do
23:                 if bitstack = 0 then return
24:                 node ← rodičovský uzel aktuálního uzlu
25:                 bitstack ← bitstack ≫ 1
26:             node ← druhý nenavšívený uzel
27:             bitstack ← bitstack ⊕ 1
```

Jádrem algoritmu je cyklus, v němž je zpracováván aktuální uzel. Na zásobník není nutné ukládat celý kontext, ale pouze jediný bit. Začne se logicky kořenovým uzlem a vynulovaným zásobníkem. Následuje test na typ uzlu, jestli se jedná o listový či standardní

uzel. V případě listového jsou testována příslušná primitiva na test kolize s paprskem, dále se nastaví příznak backtrack, jenž značí, že se bude vracet zpět ke kořenu.

V případě neterminálního uzlu proběhne test, zda paprsek protíná některého z potomků současného uzlu. V případě kolize jedním se pokračuje právě s tímto. Pokud paprsek prochází oběma potomky, pokračuje se s nejbližším a na zásobník se uloží skutečnost, že je potřeba projít i druhý uzel (na zásobník je na nejméně významný bit uložena hodnota 1). Pokud nedojde k zásahu ani jednoho z potomků, nemá smysl touto větví pokračovat a je nastaven příznak backtrack.

Na závěr se zde nachází samotná část starající se o backtracking v rámci struktury. Jedná se o cyklus, který v případě, že se na vrcholu zásobníku nachází hodnota 0, nastaví aktuální uzel na předka toho současného (předpokládá se, že informace o předkovi uzlu je buď uložena v rámci struktury nebo je jiným způsobem dostupná). Je-li na vrcholu zásobníku hodnota 1, je jako aktivní uzel nastaven dosud nezkoumaný uzel. Tímto způsobem tak dojde k navštívení všech uzlů v rámci BVH, jimž paprsek prochází a v nichž se potenciálně může nacházet primitivum se kterým koliduje.

Kapitola 7

Testování

Účelem této kapitoly je vyhodnocení vytvořené implemetace, a to konkrétně dosažená akcelerace (teda výsledná výkonnost aplikace) a zkonzumovaný čas při tvorbě akceleračních struktur pro testovanou scénu. Pro účel měření budou využity referenční scény o složitosti statisíců až milionů trojúhelníků.

7.1 Způsob a cíle měření

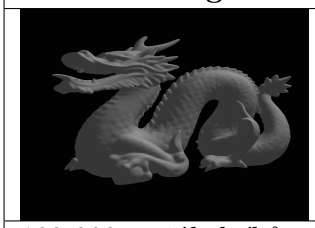
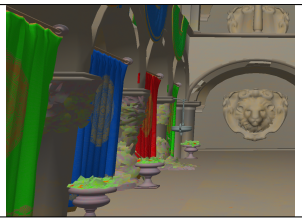
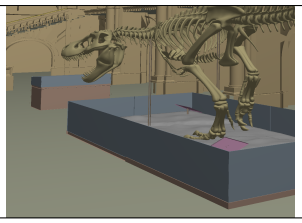
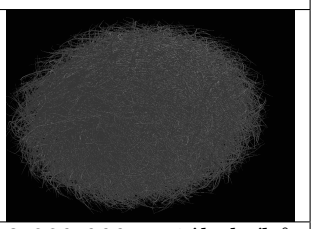
Hlavním cílem měření je porovnat obě implementované verze akceleračních struktur. Při měření bude vyhodnocena konzumace času při výstavbě struktury respektive využití času každé součásti stavby. Dále bude vyhodnoceno množství potřebné paměti pro udržování akcelerační struktury a dalších dat. Dále je důležitou součástí i vyhodnocení výsledné akcelerace, což je klíčová vlastnost, jež určuje výslednou kvalitu akcelerační struktury. Pro určení celkové výkonnosti ray traceru bude použita metrika počtu vyhodnocených paprsků za sekundu (tzv. rays per second). Meření bude probíhat pro obě realizace akceleračních struktur nad stejnými scénami. Jednotlivé scény obsahují různě složitou geometrii s různým rozložením primitiv v prostoru. Scény využité při měření jsou uvedeny v tabulce 7.2. Testování bude probíhat na několika strojích na různém hardwaru, podrobnější seznam je uveden v tabulce 7.1.

	Stroj 1	**Stroj 2**	**Stroj 3**
CPU	Intel Core i5 6402P	AMD Ryzen Threadripper 1920X	AMD Ryzen Threadripper 1920X
GPU	nVidia Geforce GTX 950	nVidia GeForce RTX 2080 Ti	AMD Radeon RX Vega 64
RAM	8 GB DDR4	16 GB DDR4	16 GB DDR4
OS	Windows 10 Pro 64-bit	Windows 10 Pro 64-bit	Windows 10 Pro 64-bit

Tabulka 7.1: Přehled strojů na nichž proběhlo testování.

7.2 Zhodnocení výsledků

Při získání výsledků jednotlivých scén bylo provedeno několikanásobné měření a jako výsledek byl použit průměr ze získaných hodnot. Získané hodnoty jsou vkládány do tabulek

Stanford dragon	**Crytek sponza**	**Stanford lucy**
		
100 000 trojúhelníků	262 205 trojúhelníků	499 518 trojúhelníků
Airbus airplane	**Nature museum**	**Hairball**
		
1 099 995 trojúhelníků	1 468 284 trojúhelníků	2 880 000 trojúhelníků

Tabulka 7.2: Tabulka scén využitých pro potřeby měření.

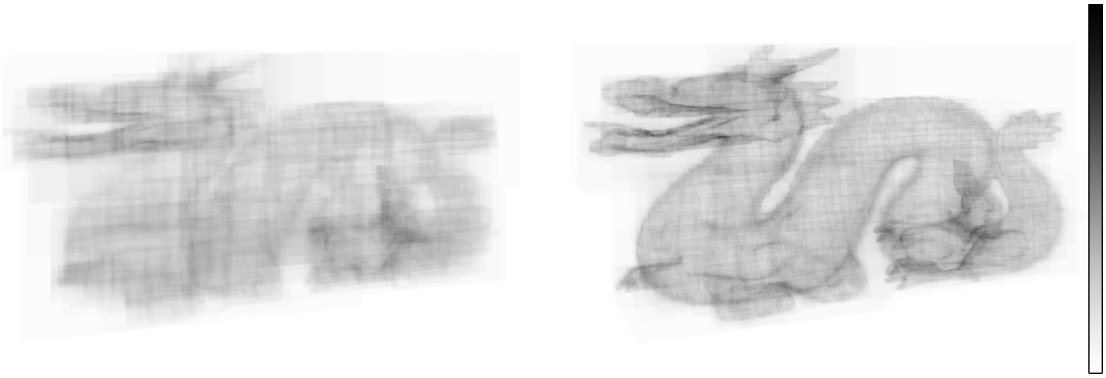
(každá obsahuje hodnoty pro jednu konkrétní scénu). Hodnoty získané v rámci první scény (Stanford dragon) jsou uvedeny v tabulce 7.3.

CPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	218.87 ms	62.52 ms	15.36 MRay/s	59 MB
Stroj 2	167.81 ms	7.49 ms	128.17 MRay/s	59 MB
Stroj 3	167.85 ms	9.11 ms	105.38 MRay/s	59 MB
GPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	153.02 ms	34.82 ms	27.57 MRay/s	66 MB
Stroj 2	18.04 ms	4.89 ms	196.32 MRay/s	66 MB
Stroj 3	19.21 ms	5.73 ms	167.54 MRay/s	66 MB

Tabulka 7.3: Naměřené hodnoty nad první scénou. Obsahuje rychlost stavby BVH, rychlost vykreslování jednoho snímku, výslednou výkonnost ray traceru a paměť využitá pro uložení akcelerační struktury.

Z uvedených hodnot je možné vidět, že i na staší grafické kartě je možné vykreslování přibližně 16 snímků za vteřinu u CPU varianty, druhá varianta vykazuje dokonce lepší výsledky (určité odůvodnění je možné vidět na heatmapě, jež je ukázána na obrázku 7.1). Co se týká stavby výsledky jsou víceméně očekávatelné, kdy GPU varianta je výrazně rychlejší (vzhledem k vysokému stupni paralelizace). U modernějších karet je vidět poměrně vysoký výkon, kdy je u obou variant dosaženo víde než 100 vykreslených snímků za sekundu.

Následující scéna je mnohem komplexnejší než předchozí, naměřené výsledky (tabulka 7.4) ukazují poměrně znatelný pokles výkonnosti, který je neúměrný počtu primitiv, pokud provedeme porovnání s předchozí scénou (zdůvodnění této skutečnosti obsahuje obrázek



Obrázek 7.1: Heatmapy první scény zobrazující počet navšívených uzlů BVH struktury v daném pixelu. U CPU varianty (vlevo) jde vidět, že struktura má v rámci celé scény přibližně stejnou hloubku, zatímco GPU varianta se lépe adaptuje na model, což je způsobeno tím, že uzly obsahují 1-2 trojúhelníky. Dále je vidět, že GPU varianta vytváří v některých místech poněkud hlubokou strukturu, kdy tato vlastnost může mít u některých scén negativní dopad. V tomto případě se to však ukazuje jako výhodnější právě GPU varianta.

7.2). Z hlediska stavby struktury je pozorovatelný nárůst času výstavby především u CPU varianty (konkrétní příčinou je výpočet minimálních bounding boxů uzlů, který u GPU varianty probíhá až po sestavení struktury pomocí strategie zdola nahoru, kdy uzel získá minimální bounding box ze svých potomků, zatímco u CPU varinaty je nutné spočítat minimální bounding volume v okamžiku zpracování konkrétního uzlu).

CPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	582.61 ms	231.91 ms	4.14 MRay/s	94 MB
Stroj 2	477.59 ms	21.91 ms	43.82 Mray/s	94 MB
Stroj 3	476.81 ms	25.18 ms	38.13 MRay/s	94 MB
GPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	186.15 ms	117.22 ms	8.19 MRay/s	125 MB
Stroj 2	35.13 ms	17.35 ms	55.33 MRay/s	125 MB
Stroj 3	37.77 ms	21.26 ms	45.16 MRay/s	125 MB

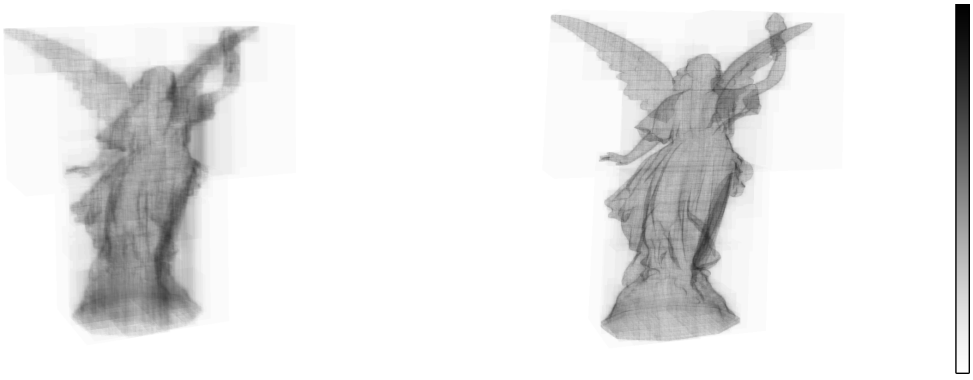
Tabulka 7.4: Hodnoty získané při měření druhé testovací scény.

Následující scéna je, co se týče počtu primitiv, přibližně dvojnásobně velká. Při pohledu na získané hodnoty (tabulka 7.5) je možné pozorovat téměř lineární růst času stavby oproti předchozím (logicky se zde projevuje právě zmíněné hledání minimálních bounding boxů a taktéž nutnost řazení primitiv). Určitý nárust lze zaznamenat i u GPU varianty, přičemž tento nárust je důsledkem řazení mortonových kódů. Naopak co se týče výkonnosti, zde dochází navzdory většímu množství primitiv k rychlejšímu výpočtu. Při pohledu na heatmapy scény (obrázek 7.3) je poměrně jasně vidět důvod, proč se takto děje. Jednoduše řečeno trojúhelníky jsou ve scéně poměrně rovnoměrně rozložené, a tím pádem nevzniká tak hluboká struktura.

Obrázek 7.2: Heatmapy druhé testované scény. Lze vidět, že hloubka struktur je značně větší než u předchozí scény. Rovněž se projevuje i fakt, že geometrie není rozložená rovnoměrně jako u předchozí scény, ale lze vidět, že se zde objevují lokace s větší složitostí, kde je hloubka značně větší. Opět se zde projevují povahy obou struktur, jak bylo zmíněno u předchozí scény.

CPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	1 271.21 ms	155.82 ms	6.16 MRay/s	106 MB
Stroj 2	1 033.91 ms	14.38 ms	66.76 MRay/s	106 MB
Stroj 3	1 033.98 ms	17.63 ms	54.45 MRay/s	106 MB
GPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	197.33 ms	117.28 ms	8.19 MRay/s	162 MB
Stroj 2	77.11 ms	7.74 ms	124.03 MRay/s	162 MB
Stroj 3	81.56 ms	10.94 ms	87.75 MRay/s	162 MB

Tabulka 7.5: Naměřené hodnoty nad třetí testovací scénou.

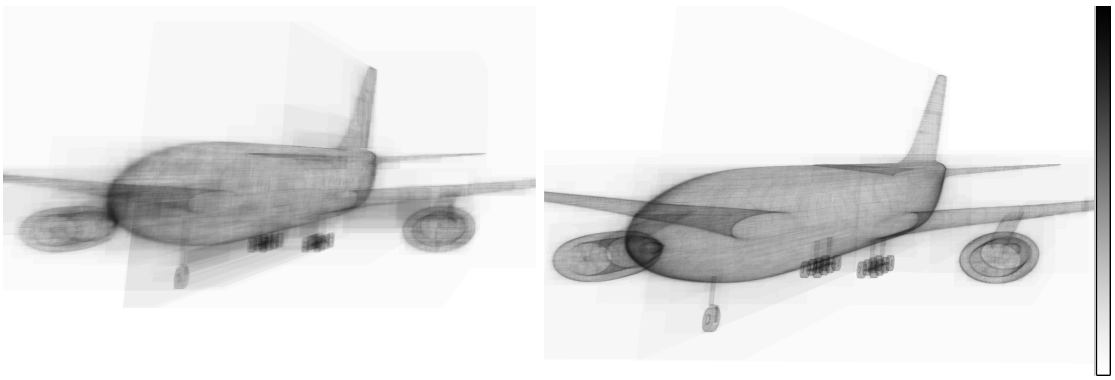


Obrázek 7.3: Heatmapy třetí testované scény. Opět se ukazují vlastnosti obou struktur, kdy GPU varianta disponuje lepší adaptací na scénu, zatímco CPU varianta obsahuje nedisponuje ideální adpatací vzhledem k faktu, že v rámci uzlů udržuje vícero primitiv.

Čtvrtá testovací scéna již obsahuje více než milion trojúhelníků. Z logiky věci se dá předpokládat, že čas stavby BVH u CPU varianty vzroste opět přibližně lineárním způsobem. Určitý nárůst se dá předpokládat i v případě GPU varianty. Při pohledu na naměřené hodnoty (tabulka 7.6) je vidět, že tomu tak přibližně je. Poněkud větší nárůst lze vidět u GPU varianty, kde je úzkým hrdlem již zmíněné řazení. V otázce výkonnosti je ovšem vidět poměrně značné zvýšení výkonnosti. Poměrně dobře tuto skutečnost ilustruje obrázek 7.4, který ukazuje, proč dochází k tomuto zrychlení.

CPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	2 745.89 ms	95.72 ms	10.03 MRay/s	213 MB
Stroj 2	2 241.39 ms	8.63 ms	111.24 MRay/s	213 MB
Stroj 3	2 241.89 ms	11.58 ms	82.90 MRay/s	213 MB
GPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	466.33 ms	62.13 ms	15.45 MRay/s	302 MB
Stroj 2	222.04 ms	5.82 ms	164.95 MRay/s	302 MB
Stroj 3	231.86 ms	8.21 ms	116.93 MRay/s	302 MB

Tabulka 7.6: Hodnoty naměřené na čtvrté testovací scéně.



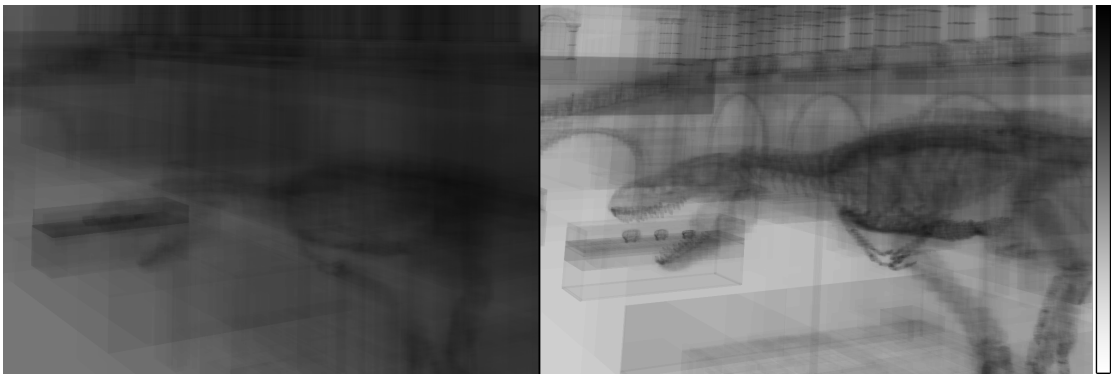
Obrázek 7.4: Heatmapy čtvrté scény. Na obou variantách lze vidět, že se poměrně dobře adaptují na určenou geometrii. I vzhledem k faktu, že scéna obsahuje přes milion trojúhelníků, výsledná struktura není příliš hluboká (navíc se dobře adaptuje), což stojí za poměrně vysokou výkonností nad touto scénou.

V případě páté scény se již projevuje zřetelné snížení výkonnosti (tabulka 7.7). To je dáno především složitostí scény (nikoliv počtem trojúhelníků, který není výrazně vyšší oproti předchozí scéně). Kvůli již zmíněné složitosti geometrie scény vzniká struktura, jež je v určitých lokacích značně hluboká a je tedy nutné procházet velké množství uzlů (viz obrázek 7.5). Samotná doba stavby opět vzrostla víceméně očekávaným způsobem.

U poslední testovací scény je situace poněkud odlišná. Nijak zvlášťní nejsou doby stavby obou variant, ty dopadly poměrně očekávaně vzhledem k předchozím scénám. Významný pokles lze zaznamenat u dob výpočtu, a to u obou variant (viz tabulka 7.8). Za tímto významným poklesem stojí především charakter geometrie scény, ačkoliv se struktury po-

CPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	3 774.12 ms	272.22 ms	3.53 MRay/s	251 MB
Stroj 2	3 063.49 ms	44.55 ms	21.55 MRay/s	251 MB
Stroj 3	3 063.46 ms	49.74 ms	19.30 MRay/s	251 MB
GPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	539.11 ms	134.82 ms	7.12 MRay/s	347 MB
Stroj 2	181.62 ms	32.31 ms	29.71 MRay/s	347 MB
Stroj 3	192.71 ms	36.19 ms	26.53 MRay/s	347 MB

Tabulka 7.7: Hodnoty naměřené nad pátou testovací scénou.

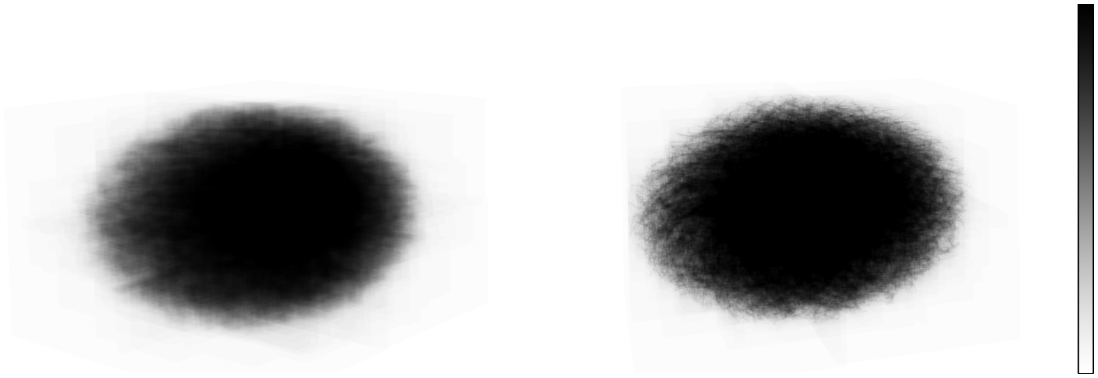


Obrázek 7.5: Heatmapy páté testovací scény. U CPU varianty je vidět, že má u této scény poměrně problémy a nevytváří tak kvalitní a dobře adaptovanou strukturu jako v případě některých předchozích scén. Tento fakt se jednoznačně podepisuje na celkové výkonnosti. V tomto případě celkem výrazněji vítězí GPU varianta, která však také v některých lokalitách tvoří poměrně hlubokou strukturu.

měrně dobře adaptují na danou scénu, výsledné výkonnosti to téměř nepomůže. Obrázek 7.6 ilustruje příčinu nízké výkonnosti nad touto scénou.

CPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	9 359.67 ms	896.72 ms	1.07 MRay/s	372 MB
Stroj 2	7 263.72 ms	319.68 ms	3.01 MRay/s	372 MB
Stroj 3	7 264.15 ms	347.55 ms	2.76 MRay/s	372 MB
GPU varianta				
	Stavba BVH	**Čas vykreslování**	**Výkonnost**	**Spotřeba paměti**
Stroj 1	792.15 ms	891.92 ms	1.08 MRay/s	446 MB
Stroj 2	371.08 ms	368.45 ms	2.61 MRay/s	446 MB
Stroj 3	384.16 ms	386.65 ms	2.48 MRay/s	446 MB

Tabulka 7.8: Naměřené hodnoty u poslední testovací scény.



Obrázek 7.6: Heatmapy poslední scény. Ačkoliv se obě varianty celkem obstojně adaptují na danou geometrii, paprsky, které musí procházet prostřední částí, musí procházet poměrně složitou strukturou geometrie, tím pádem i velkým množstvím uzlů BVH. Na příkladu této scény se ukazuje, že nemusí být vždy vhodné použití BVH z důvodu nutného testování všech uzlů, jimiž testovaný paprsek může procházet, což má zde fatální vliv na celkový výkon.

Obecně se dá říci, že díky akceleračním strukturám je možné zobrazovat i poměrně komplexní scény v reálném čase, avšak nemusí to tak vždy platit, jak ukazuje příklad poslední scény, kde by bylo pravděpodobně vhodné využít jinou datovou strukturu. Z měření lépe vycházela GPU varianta, která se obecně lépe zvládala vhodněji adaptovat na scénu, u CPU varianty, ačkoliv byla použita metrika SAH, záleží na zvolených cenových funkcích, které slouží jako vstup pro vlastní výpočet, tedy v případě nalezení vhodnějších kandidátů by bylo možné potenciálně vytvořit i lepší strukturu a tedy dosáhnout vyšší akcelerace.

Kapitola 8

Závěr

Cílem této práce bylo nalézt možné způsoby akcelerace algoritmu sledování paprsku. Nejprve bylo zapotřebí kromě studia principu této metody také nalezení tzv. úzkého hrdla. Na základě tohoto bylo zvoleno využití akceleračních struktur majících za cíl urychlit část hledání kolize paprsku se scénou. Jako akcelerační struktura byla vybrána konkrétně hierarchie obalových těles (BVH). V rámci práce byly implementovány dva způsoby stavby této struktury, přičemž jeden, který je založený na metrice SAH, je stavěn standardně na CPU, zatímco stavba druhé varianta je akcelerována na grafickém procesoru.

Dále byla pro účely testování vytovřena aplikace, jež zobrazuje scénu pomocí techniky sledování paprsku, přičemž využívá výše zmíněné implementované datové struktury. Obě struktury byly otestovány na scénách jejichž složitost se pohybuje od statisíců až do řádu milionů trojúhelníků, přičemž bylo měřeno na jedné standardní grafické kartě a dvou vysoce výkonných kartách. Výsledky měření ukázaly, že i zmíněná méně výkonná karta zvládala výpočet snímku a jeho následné zobrazení u jednotlivých scén poměrně obstojně (s výjimkou poslední scény). Trochu překvapivým úkazem může být zároveň i lepší výkonnost GPU varianty akcelerační struktury oproti struktuře založené na metrice SAH. Odůvodněním může být volba horších cenových funkcí, jež slouží jako vstup pro výpočet této heuristiky. Dalším kritériem je také strategie volby dělící osy, jež není snadné vždy vhodně určit. Tato část může být tak předmětem dalšího experimentování a hledání optimálního řešení.

Využití akceleračních struktur je však možné i mimo aplikace sledování paprsku, je možné jej využít například u technik jako jsou Frustum culling, picking, případně pro akceleraci výpočtů ve fyzikálních enginech. Z tohoto důvodu je část akceleračních struktur navržená takovým způsobem, aby byla použitelná a do budoucna rozšiřitelná o další strategie stavby heirerchie obalových těles. Zdrojové kódy jsou k nalezení na webu[1] [2].

[1] Knihovní část: https://github.com/davidnov12/BoundingVolumeHierarchy
[2] Ray Tracing aplikace: https://github.com/davidnov12/RayTracingMT

Literatura

[1] NVIDIA Fermi Compute Architecture Whitepaper. 2009.
URL https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[2] Áfra, A. T.; Szirmay-Kalos, L.: Stackless Multi-BVH Traversal for CPU, MIC and GPU Ray Tracing. *Comput. Graph. Forum*, ročník 33, č. 1, Únor 2014: s. 129–140, ISSN 0167-7055, doi:10.1111/cgf.12259.
URL http://dx.doi.org/10.1111/cgf.12259

[3] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN 0-201-70431-5.

[4] 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

[5] Baldwin, D.; Weber, M.: Fast Ray-Triangle Intersections by Coordinate Transformation. *Journal of Computer Graphics Techniques (JCGT)*, ročník 5, č. 3, September 2016: s. 39–49, ISSN 2331-7418.
URL http://jcgt.org/published/0005/03/03/

[6] Cook, R. L.; Porter, T.; Carpenter, L.: Distributed Ray Tracing. *SIGGRAPH Comput. Graph.*, ročník 18, č. 3, Leden 1984: s. 137–145, ISSN 0097-8930, doi:10.1145/964965.808590.
URL http://doi.acm.org/10.1145/964965.808590

[7] Georgiev, I.; Křivánek, J.; Davidovič, T.; aj.: Light transport simulation with vertex connection and merging. *ACM Trans. Graph.*, ročník 31, č. 6, Listopad 2012: s. 192:1–192:10, ISSN 0730-0301, doi:10.1145/2366145.2366211.
URL http://doi.acm.org/10.1145/2366145.2366211

[8] Hapala, M.; Davidovič, T.; Wald, I.; aj.: Efficient Stack-less BVH Traversal for Ray Tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics*, SCCG '11, New York, NY, USA: ACM, 2013, ISBN 978-1-4503-1978-2, s. 7–12, doi:10.1145/2461217.2461219.
URL http://doi.acm.org/10.1145/2461217.2461219

[9] Hecht, E.: *Optics*, kapitola The Propagation of Light. Addison-Wesley, 2002, ISBN 9780321188786, str. 107.

[10] Jensen, H. W.: Global Illumination Using Photon Maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, London, UK, UK: Springer-Verlag, 1996, ISBN 3-211-82883-4, s. 21–30.
URL http://dl.acm.org/citation.cfm?id=275458.275461

[11] Kajiya, J. T.: The Rendering Equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, New York, NY, USA: ACM, 1986, ISBN 0-89791-196-2, s. 143–150, doi:10.1145/15922.15902.
URL http://doi.acm.org/10.1145/15922.15902

[12] Karras, T.: Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. In *High Performance Graphics 2012*, Červen 2012.
URL https://devblogs.nvidia.com/wp-content/uploads/2012/11/karras2012hpg_paper.pdf

[13] Kay, T. L.; Kajiya, J. T.: Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph.*, ročník 20, č. 4, Srpen 1986: s. 269–278, ISSN 0097-8930, doi:10.1145/15886.15916.
URL http://doi.acm.org/10.1145/15886.15916

[14] Lauterbach, C.; Garland, M.; Sengupta, S.; aj.: Fast BVH construction on GPUs. In *Computer Graphics Forum*, ročník 28, Wiley Online Library, 2009, s. 375–384.

[15] MacDonald, D. J.; Booth, K. S.: Heuristics for Ray Tracing Using Space Subdivision. *Vis. Comput.*, ročník 6, č. 3, Květen 1990: s. 153–166, ISSN 0178-2789, doi:10.1007/BF01911006.
URL http://dx.doi.org/10.1007/BF01911006

[16] Meister, D.; Bittner, J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE transactions on visualization and computer graphics*, ročník 24, č. 3, 2018: s. 1345–1353.
URL https://dcgi.felk.cvut.cz/projects/ploc/ploc-tvcg.pdf

[17] Möller, T.; Trumbore, B.: Fast, Minimum Storage Ray-triangle Intersection. *J. Graph. Tools*, ročník 2, č. 1, Říjen 1997: s. 21–28, ISSN 1086-7651, doi:10.1080/10867651.1997.10487468.
URL http://dx.doi.org/10.1080/10867651.1997.10487468

[18] Morton, G. M.: A computer oriented geodetic data base and a new technique in file sequencing. Březen 1966.
URL https://domino.research.ibm.com/library/cyberdig.nsf/papers/0DABF9473B9C86D48525779800566A39/\protect\T1\textdollarFile/Morton1966.pdf

[19] Stich, M.; Friedrich, H.; Dietrich, A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 2009, s. 7–13.

[20] 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