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

BUDOVÁNÍ AKCELERAČNÍCH STRUKTUR NA GPU

ACCELERATION STRUCTURE BUILDING ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **JIŘÍ HÁBA**
AUTHOR

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

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Hába Jiří**
Program: Informační technologie
Název: **Budování akceleračních struktur na GPU**
Acceleration Structure Building on GPU
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte problematiku akceleračních struktur, jejich použití a vytváření. Nastudujte potřebné knihovny a nástroje.
2. Navrhněte knihovnu pro tvorbu vybrané akcelerační struktury.
3. Implementujte knihovnu. K implementaci použijte nástroje CMake a Git.
4. Implementujte demonstrační aplikaci.
5. Proveďte měření a vyhodnocení.

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a část bodu 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Starka Tomáš, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 13. listopadu 2019

Abstrakt

Tato práce se zaměřuje na výstavbu oktalového stromu pro trojrozměrné trojúhelníkové modely na grafickém hardware. Tato akcelerační struktura byla zvolena na základě porovnání nejčastěji užívaných akceleračních struktur. Přístup ke stavbě této struktury vychází ze způsobu výstavby z ní odvozeného řídkého voxelového oktalového stromu. Při procesu je využívána voxelizace urychlovaná vykreslovacím řetězcem grafické karty. K usnadnění práce a umožnění začlenění navrhovaného postupu do v budoucnu vznikajících aplikací je součástí této práce i návrh knihovny jazyka C++.

Abstract

This thesis focuses on the construction of octree for three-dimensional triangular models on graphics hardware. This acceleration structure was chosen based on a comparison of the most commonly used acceleration structures. The approach to the construction of this structure is based on the method of construction of a sparse voxel octree which is derived from the former. The process uses voxelization accelerated by the graphics card's rendering pipeline. To facilitate this process and enable the integration of the proposed procedure into future applications, this thesis also includes design of a C++ language library.

Klíčová slova

akcelerační struktura, oktalový strom, řídký voxelový oktalový strom, voxelizace, GPU, OpenGL, knihovna

Keywords

acceleration structure, octree, sparse voxel octree, voxelization, GPU, OpenGL, library

Citace

HÁBA, Jiří. *Budování akceleračních struktur na GPU*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Starka

Budování akceleračních struktur na GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Starky. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
Jiří Hába
28. května 2020

Poděkování

Děkuji vedoucímu práce, panu Ing. Tomáši Starkovy, za odborné vedení a užitečné rady.

Obsah

1 Úvod **2**

2 Teorie **4**
2.1 Akcelerační struktury 4
2.1.1 Objektové hierarchie 5
2.1.2 Metody dělení prostoru 6
2.1.3 Hybridní metody 8
2.1.4 Metriky hladových algoritmů 9
2.2 Oktalový strom 10
2.2.1 Budování stromu 11
2.2.2 Průchod stromem 11
2.3 Sparse Voxel Octree (SVO) 12
2.3.1 Budování stromu 12
2.3.2 Průchod stromem 13
2.4 Voxelizace vykreslovacím řetězcem OpenGL 13
2.4.1 Aplikace rasterizace při voxelizaci 13

3 Návrh **16**
3.1 Knihovna pro práci s oktalovým stromem 16
3.1.1 Inicializace 17
3.1.2 Stavba akcelerační struktury 18
3.1.3 Průchod strukturou 18

4 Implementace **21**
4.1 Třída OB::Octree 21
4.2 Příklad použití knihovny 22
4.3 Voxelizace scény 23
4.4 Tvorba struktury stromu 26
4.5 Ukázková aplikace 28

5 Měření **29**
5.1 Testovací scény 29
5.2 Postup a výsledky měření 30

6 Závěr **32**

Literatura **33**

A Obsah přiloženého paměťového média **35**

Kapitola 1

Úvod

Jedním z hlavních cílů počítačem generované grafiky je snaha ošálit lidské oko, aby nebylo schopné rozlišit, zda mu byl předložen uměle vytvořený obraz, či fotografie reálného prostředí. Dosažení takové míry detailu však vyžaduje velmi podrobně definovanou 3D scénu. Scény tak zcela běžně obsahují nemalé množství trojúhelníkových síťových modelů, které jsou tvořeny tisícovkami trojúhelníků.

To ale není jediná věc přispívající k věrohodnosti výsledku. Důležité je také použití věrné zobrazovací metody. Pro svou kvalitu výsledného obrazu a univerzální definici povrchových vlastností materiálů (umožňuje popsat materiály lesklé, matné, broušené, průsvitné, průhledné, světlo vyzařující atd.) bývá často volena metoda sledování paprsku. Ta je ovšem velmi výpočetně náročná, jelikož vyšetřuje jednotlivé paprsky světla a způsob jakým se odrážejí od těles ve scéně. Nejkritičtějším místem výpočtu je tedy značně vysoký počet testů kolizí paprsků s trojúhelníky definujícími jednotlivé objekty.

Jednou ze skupin metod, které hrají důležitou roli při zajištění urychlení vykreslování 3D grafiky, jsou akcelerační struktury. Ty obecně umožňují eliminaci výpočtů, které nijak nepřispívají k vytvoření výsledného obrazu (snižují celkový počet testů kolizí paprsku s trojúhelníkem). Jejich hlavní nevýhodou je však čas, který je potřebný k vybudování dané akcelerační struktury. U větších scén, které obsahují více než statisíce, někdy až desítky milionů grafických primitiv, se doba budování takovýchto struktur často pohybuje v řádu desítek až stovek vteřin. Pokud je kritériem zobrazovat 3D scény v reálném čase, musíme být schopni vykreslovat snímky v intervalech nanejvýš 33,3 ms (pro snímkovou frekvenci 30 snímků za vteřinu). Pro statické scény není žádným problémem provést sestavení akcelerační struktury předem. Problém nastává v dynamických scénách s nezanedbatelným množstvím animovaných a pohyblivých prvků, kde je nutné akcelerační struktury velmi často aktualizovat, či rovnou budovat znovu.

V úvodní části práce jsou srovnány současně nejpoužívanější akcelerační struktury. Jsou rozebrány způsoby jejich výstavby a způsob použití v koncových případech. Kapitola se dále podrobněji zabývá oktalovými stromy a jejich řídkou variantou.

Kapitola 3 popisuje zhodnocení současně užívaných metod a návrh řešení jejich restrikcí. Součástí návrhu je i definice knihovny usnadňující použití představované metody v budoucích aplikacích.

V kapitole 4 je popsán způsob jakým je implementována knihovna pro práci s oktalovým stromem. Je zde představena ukázková aplikace jednoduchého nástroje pro zobrazování 3D modelů, který demonstruje použití vytvořené knihovny.

V kapitole 5 jsou představeny výsledky měření efektivity představovaného řešení na testovacích 3D scénách. Rovněž jsou popsány pravděpodobné důvody vznikajících časových prodlev.

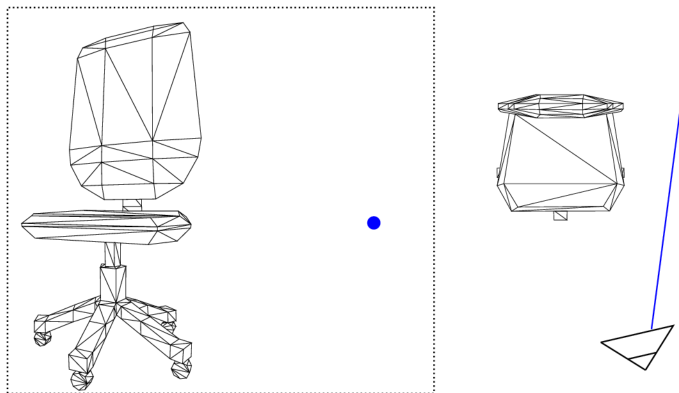
Kapitola 2

Teorie

Tato kapitola poskytuje čtenáři základní přehled běžně užívaných akceleračních struktur pro urychlování paprskových vykreslovacích metod. Více do hloubky jsou zde popsány dvě varianty oktalových stromů. Na konci této kapitoly je popsán způsob, kterým je možné nahradit test příslušnosti trojúhelníků v prostoru k jednotlivým voxelům scény pomocí voxelizace při využití rasterizačního řetězce grafické karty (GPU).

2.1 Akcelerační struktury

Ke značnému urychlení vykreslování paprskovými zobrazovacími metodami kromě optimalizací samotných algoritmů (např. využitím datového/vláknového paralelismu [27][20]), či optimalizací na úrovni hardware (např. návrhem specializovaných obvodů [23]) přispívají právě akcelerační datové struktury. Ty efektivně snižují množství výpočtů spojených s určováním průniků vržených paprsků s prvky scény. Dosahuje se toho tak, že se pro velké skupiny grafických primitiv provede nejprve jeden test, který zjišťuje, zdali vyšetřovaný paprsek prochází jejich sdíleným prostorem. Pouze v kladném případě se přistoupí k určování průniku paprsku s jednotlivými prvky scény. Tímto způsobem se pak při vykreslování zanedbá významné množství geometrie (při vhodném rozčlenění scény) pro každý vržený paprsek, protože ji ani nemůže protnout a není potřeba průnik s takovými prvky uvažovat. Taková situace je znázorněna na obrázku 2.1.



Obrázek 2.1: Paprsek (modrý) neprotíná žádný z trojúhelníků modelu kancelářské židle, ale i tak by bez použití akcelerační struktury bylo nutné ověřit kolizi s každým z nich

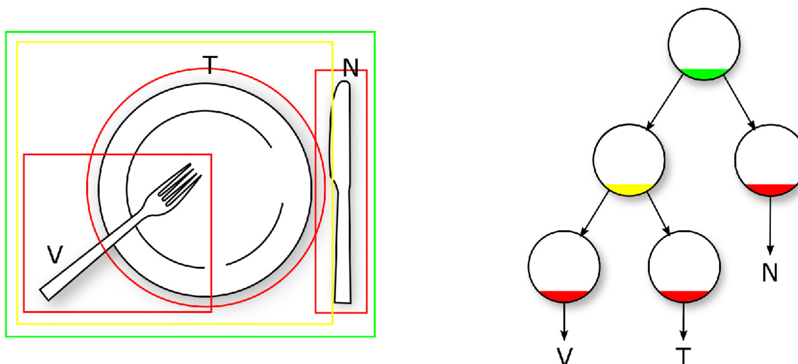
Valná většina akceleračních struktur je hierarchicky členěna. Je proto možné je v takových případech chápat jako stromové datové struktury. Atraktivita hierarchických struktur spočívá v časové složitosti průchodu strukturou, která je logaritmická.

Obecně jsou pak akcelerační struktury rozdělovány do dvou skupin. První jsou založeny na shlukování prvků scény a následném zapouzdření nalezených shluků obalovacími tělesy – objektové hierarchie. Ty druhé naopak scénu dělí množstvím rovinných řezů na menší podprostory, kterým se přisuzuje náležitost obsažených grafických primitiv – metody dělení prostoru. Můžeme se ale také setkat s kombinovanými, hybridními přístupy [13].

2.1.1 Objektové hierarchie

Do této skupiny metod se řadí především hierarchie obalovacích těles (angl. bounding volume hierarchy, zkráceně BVH) a metody z ní odvozené. Hlavním společným rysem objektových hierarchií je to, že oblasti odpovídající jejich uzlům na dané úrovni se mohou překrývat, ale jednotlivé prvky scény jsou vždy přiřazeny jen jedné z těchto oblastí. Největší výhodou je bezpochyby to, že objektové hierarchie popisují pouze oblasti, které jsou neprázdné. Tím totiž předcházejí zbytečnému vyšetřování oblastí, kde žádná kolize paprsku se scénou nastat ani nemůže.

BVH. Objekty ve 3D scénách se v závislosti na jejich poloze dají seskupovat do logických celků, které je možné dále sdružovat ve vzrůstající míře abstrakce, až do té doby než dojde k vytvoření posloupnosti shluků, která obsahuje celou scénu (např. knihy na polici → skříňka → kancelářský kout → celý byt). Z tohoto pozorování vykládají Weghorst a kol. [28] význam rozdělení scény na hierarchii shluků prvků jejího prostředí. Jednotlivé objekty i shluky jsou pak popisovány obalovacími tělesy, které je zcela obklopují (viz obrázek 2.2). Hierarchie složená z obalovacích těles jednotlivých shluků a objektů se reprezentuje jako binární strom (existují ale i varianty s vyššími aritami [7]), který je při vykreslování procházen od nejvyšší úrovně níž podle toho, které shluky jsou na dané úrovni paprskem zasaženy.



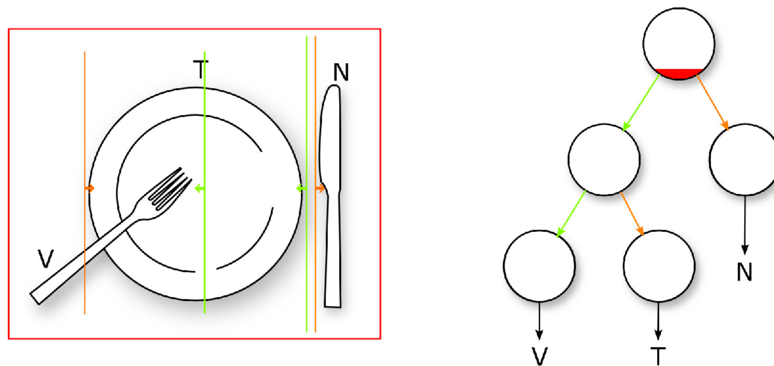
Obrázek 2.2: Reprezentace rozmístění objektů (V: vidlička; T: talíř; N: nůž) pomocí BVH

BIH. Hierarchie obalovacích intervalů (angl. bounding interval hierarchy) je obdobou BVH s tím rozdílem, že shluky na jednotlivých úrovních jsou zapouzdřeny pomocí dvou obalovacích intervalů – jedná se tedy opět o binární strom (ukázáno na obrázku 2.3). Obalovací interval je chápán jako poloprostor, jehož mezní rovina je kolmá na některou ze souřadných os scény. Obalovací intervaly jednoho rodiče se mohou překrývat, ale mohou být vůči sobě i disjunktní (včetně případu kdy je mezi nimi prázdný prostor). Podobně

jako u BVH je tedy možné se vyhnout nutnosti popisovat rozsáhlé prázdné oblasti prostoru scény. Oproti BVH má však velké výhody při budování a při průchodu.

K vybudování struktury je možné použít nehladového algoritmu, což značně urychluje tento proces. Oblasti uzlů (mající tvar kvádru) jsou rozděleny na poloviny dle jejich nejdelší strany. Objekty které těmto oblastem náleží jsou přiděleny vzniklým polovinám podle toho, do které zasahují větším dílem. Synovské obalovací intervaly se pro daný uzel vytvoří tak, že se stanoví maximální případně minimální koordináta vrcholů objektů těchto shluků v dané ose [26].

Výhodou je i to, že průchod strukturou se velmi podobá průchodu k-d stromu (popsán v následující podkapitole), který byl Havranem statisticky vyhodnocen za nejefektivnější akcelerační strukturu z řady nejběžněji užívaných [12].

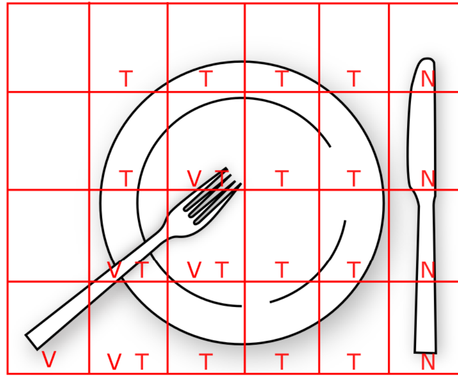


Obrázek 2.3: Reprezentace rozmístění objektů (V: vidlička; T: talíř; N: nůž) pomocí BIH

2.1.2 Metody dělení prostoru

Na rozdíl od objektových hierarchií metody dělení prostoru organizují prvky scény do disjunktních oblastí. Prvky, které zasahují do více než jedné oblasti, musí být tedy odkazovány ze všech listových uzlů odpovídajících náležitým oblastem.

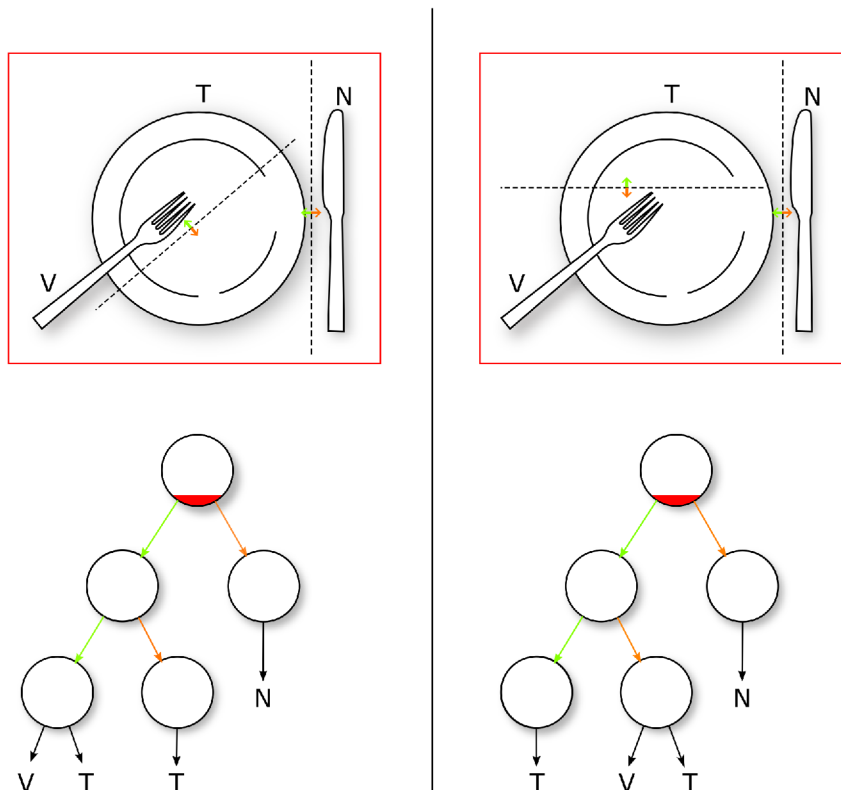
Uniformní mřížka. Uniformní mřížka rozděluje oblast, která obsahuje všechna primitiva scény, na pravidelně velké oddíly, které jsou zarovnané s osami, v předem stanoveném rozlišení (viz. obrázek 2.4). Průchod strukturou je realizován pomocí trojrozměrné obdoby digitálního diferenciálního analyzátoru (DDA) pro rasterizaci přímky [2]. Nejefektivnější je užití uniformní mřížky ve scénách, kde jsou jednotlivá primitiva v prostoru distribuována pravidelně. To proto že v tomto případě skončí průchod paprsku strukturou již po pár krocích. Navíc pro takový typ scén rychlost průchodu předčí mnohé hierarchické struktury [10]. Značným zpomalením však uniformní mřížka trpí ve scénách, které obsahují prvky nerovnoměrně uspořádané v nepravidelných shlucích. Řadí se sem scény s nerovnouměrnou úrovní detailu v různých oblastech scény nebo scény s nepravidelným využitím prostoru. S tímto problémem se snaží vypořádat různé hierarchické varianty uniformní mřížky jako rekurzivní mřížky, hierarchie uniformních mřížek a adaptivní mřížky [12].



Obrázek 2.4: Obsah jednotlivých buněk uniformní mřížky

BSP a k-d strom. Binární stromy pro dělení prostoru (angl. binary space partition trees) rekurzivně dělí prostor na dvě části řezovou rovinou. Je možné je použít k definování tvaru pevného objektu, ale i jako binární vyhledávací stromy pro grafická primitiva umístěná v prostoru. Při stavbě stromu je nejnáročnější operací nalezení optimální polohy a orientace řezových rovin. Běžně se tedy volí užití hladové heuristiky pro nalezení dostačujícího suboptimálního řešení v nepoměrně kratším čase. Postupným dělením prostoru se vždy vytvářejí konvexní oblasti. Pokud se řezová rovina v listovém uzlu shoduje s rovinou jediného trojúhelníka v tomto uzlu, je možné provést test kolize paprsku již při běžném průchodu [14].

Další obdobnou strukturou jsou k-d stromy, které je možné považovat za specializaci obecného BSP stromu (porovnání těchto dvou struktur je vyobrazeno na obrázku 2.5). Od obecného BSP stromu se liší pouze tím, že dělící roviny jednotlivých uzlů stromu jsou vždy zarovnány dle souřadných os scény. Výpočet bodu průniku paprsku s oblastí uzlu je tak značně zjednodušen. Pro určení orientace a pozice řezových rovin může být zvoleno mnoho postupů. Základním způsobem je cyklické určení pořadí hlavních os pro jednotlivé úrovně stromu s dělením ve středu oblasti. Ke zvýšení kvality struktury ale nejvíce přispívá určení řezových rovin podle metriky předpokládané ceny průchodu strukturou (popsáno v sekci 2.1.4). Ukončovací podmínka rekurzivního dělení scény může být maximální hloubka stromu, maximální počet odkazovaných primitiv z listového uzlu, či maximální hodnota metriky ceny [12].



Obrázek 2.5: Srovnání BSP (vlevo) a k-d stromu (vpravo)

Oktalový strom. Touto akcelerační strukturou se tato práce zabývá podrobněji v podkapitole 2.2 na straně 10 a její řídkou variantou v podkapitole 2.3 na straně 12.

2.1.3 Hybridní metody

Samozřejmě existují i kombinace předešlých dvou skupin akceleračních struktur. Nejjednodušším příkladem by mohl být BVH strom, který v listových uzlech odkazuje uniformní mřížky vytvořené z odpovídajících obalových těles. Taková struktura je ovšem předpokládána jako ne příliš efektivně průchodná v paralelizované podobě oproti klasickému BVH [25]. Existují však i hybridní přístupy, které jsou pro paralelizaci vhodnější.

SBVH. Dělená hierarchie obalovacích těles (angl. split bounding volume hierarchy) je metodou, která vznikla v reakci na nedostatky běžného BVH a to především na negativní vliv případu, kdy geometrie scény není rovnoměrně rozmístěna. To způsobuje nevyhnutelné překrývání jednotlivých obalovacích těles, které vede ke snížení efektivity užití akcelerační struktury. Naivním řešením by mohlo být vybrat obalovací tělesa tak, aby se nepřekrývala, a veškerou geometrii, která přesahuje do druhé oblasti rozdělit jejich hraniční rovinou. To však může vést k vytvoření nezanedbatelného počtu trojúhelníků, což by znamenalo nutnost prohloubení stromu a navýšení času průchodu strukturou. Namísto toho se u SBVH v podobných případech nedělí geometrie ale prostor uzlu, je-li to přínosnější než rozdělení na synovské shluky (určuje se metrikou ceny průchodu vyhodnocené pro obě možnosti). Rozhoduje se tedy o tom, jestli bude daný uzel při stavbě stromu rozdělen jako BVH, nebo jako k-d strom [24].

2.1.4 Metriky hladových algoritmů

Obecně pro to aby byla daná akcelerační struktura užitečná pro konkrétní aplikaci, musí být sestavena v dostatečné kvalitě. Tím je myšleno to, jak dobře popisuje rozmístění objektů a jakou má vnitřní topologii. Samotná kvalita se stanoví metrikou ceny průchodu strukturou. Důležitým aspektem budování hierarchických akceleračních struktur je tedy právě zvolení vhodné metriky která se použije pro určení, zda je vhodné strukturu při stavbě nadále dělit.

SAH. Jednou z běžných metrik je i heuristika povrchové plochy (angl. surface area heuristic), která se navíc využívá pro stanovení odhadu ceny užití daného stromu (jeho celková cena). Je definována jako očekávaná cena testu kolize paprsku vedeného náhodným směrem z náhodného počátku s prvky scény [16]:

$$SAH = \frac{C_i * \sum_{i=1}^{N_i} SA(i) + C_l * \sum_{l=1}^{N_l} SA(l) + C_o * \sum_{l=1}^{N_l} SA(l) * N(l)}{SA(root)} \tag{2.1}$$

Kde:

C_i je cena průchodu vnitřním uzlem stromu

C_l je cena průchodu listovým uzlem stromu

C_o je cena testu kolize paprsku s trojúhelníkem

N_i je celkový počet vnitřních uzlů

N_l je celkový počet lisových uzlů

$N(l)$ je počet trojúhelníků odkazovaných z listového uzlu l

$SA(x)$ je povrchová plocha oblasti uzlu x

Zde je možné pozorovat, že na určení této ceny má největší vliv stanovení pravděpodobnosti, zdali paprsek zasáhne oblast daného uzlu. Poměr povrchové plochy oblasti daného uzlu a kořenového uzlu se pak chápe jako odhad této pravděpodobnosti.

U akceleračních struktur, u kterých se cena průchodu vnitřního a listového uzlu neliší, je možné výraz zjednodušit:

$$SAH = \frac{C_n * \sum_{n=1}^{N_n} SA(n) + C_o * \sum_{l=1}^{N_l} SA(l) * N(l)}{SA(root)} \tag{2.2}$$

Kde:

C_n je cena průchodu uzlem stromu

N_n je celkový počet uzlů stromu

Lokální SAH. Nalezení stromu, který by měl SAH rovnu globálnímu minimu pro konkrétní scénu, není v komplexních scénách považováno za uskutečnitelné v rozumném čase. Z tohoto důvodu se přistupuje k aproximaci hladovým algoritmem, který používá lokální odhad ceny při rozdělování uzlu. Předpokládá, že vzniklé synovské uzly mohou být pouze listy a strom jimi dále nepokračuje. Tím efektivně nadhodnocuje cenu průchodu uzlů, jejichž potomci budou v budoucnu dále děleni. V praxi se ukazuje, že to však není vážný nedostatek a že stromy vytvořené s užitím této heuristiky (byť jsou suboptimální) jsou stále velmi kvalitní [27].

RDH. K přesnějšímu vyjádření vlivu prostorového uspořádání jednotlivých paprsků na pravděpodobnost zásahu některé z oblastí stromu je možné užít heuristiku distribuce paprsků (angl. ray distribution heuristic). Ta bere v potaz distribuci paprsků v prostoru, čímž se zabraňuje dělení stromu v oblastech scény zastíněných ostatními objekty. V takových případech je totiž vysoká míra jemného dělení scény zbytečná, vezme-li se v úvahu nepravděpodobnost podobných situací. K výpočtu ceny RDH se nejprve musí vytvořit sada reprezentativních paprsků (obsahující i odražené paprsky), která je výrazně menší než množina paprsků vržených při samotném vykreslování. Pravděpodobnost zda paprsek bude při použití akcelerační struktury procházet danou oblastí uzlu se stanoví jako poměr počtu paprsků reprezentativní sady, které tímto uzlem procházejí a těmi, které procházejí jeho rodičem. Výsledná cena se stanoví pomocí interpolace pravděpodobnosti stanovené pomocí RDH a SAH v závislosti na počtu paprsků z reprezentativní sady, které daný uzel protnuly [3].

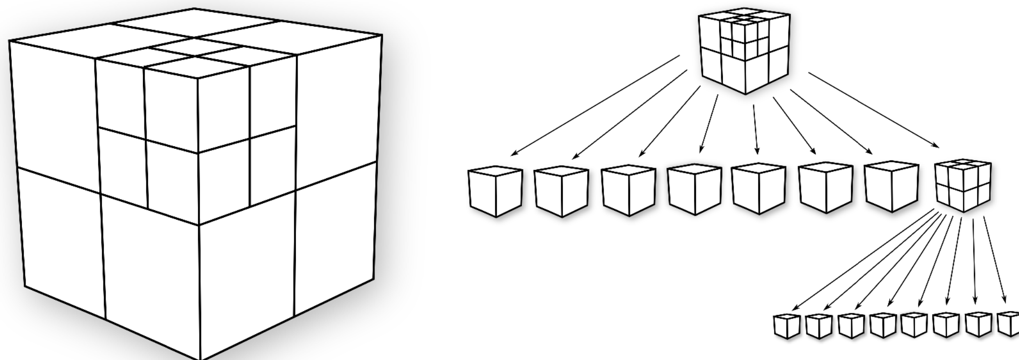
2.2 Oktalový strom

Oktalový strom (angl. octree) je stromovou datovou strukturou patřící do skupiny technik dělení prostoru (angl. spatial subdivision). Jednotlivé uzly stromu se běžně nazývají buňky, či oktanty, je ale možné se setkat i s označením voxely (ty jsou ovšem snadno zaměnitelné s pojmenováním jednotky voxelové mřížky). Odpovídají jim oblasti prostoru scény, které jsou ohraničeny šesti ořezovými rovinami. Kořenový uzel stromu popisuje ohraničení celého prostoru scény. Pro N-dimenzionální prostor může být každý uzel stromu rozdělen na 2^N stejných podčástí N řezy (zarovnanými podle souřadných os a procházejících těžištěm prostoru znázorněným daným uzlem). Ve 3D prostoru tak vzniká 8 synovských uzlů (viz. obrázek 2.6) a odtud také pochází název oktalový strom. Jednotlivé úrovně oktalového stromu se ve výsledku nápadně podobají rozdělení prostoru uniformní mřížkou (je však porušena přímá sousednost osmic buněk).

Původní definice Donalda Meaghera [17] popisuje oktalový strom jako reprezentaci tvaru jediného N-dimenzionálního objektu pomocí 2^N-árního stromu. Zde vnitřní uzly stromu popisují oblasti, do kterých objekt pouze zasahuje, a listové uzly pak popisují oblasti, které jsou buď objektem vyplněny zcela, a nebo naopak do nich objekt nezasahuje vůbec.

V současnosti je oktalový strom spíše než jako popis tvaru pevného objektu chápán ve smyslu struktury sloužící k reprezentaci volumetrických dat. Jednotlivé uzly stromu tak popisují množinu vlastností dané podčásti prostoru, které svou pozicí ve stromu odpovídají. Tyto vlastnosti se mohou týkat barvy, hustoty, teploty nebo i toho, jaká grafická primitiva do jejich oblasti spadají. Poslední případ odpovídá použití oktalového stromu jako akcelerační struktury, kdy se při vykreslování paprskovými zobrazovacími metodami umožňuje testovat kolize paprsku procházejícího scénou s podoblastmi tvořícími hierarchii scény ještě

před tím, než se provádí kolize paprsku se všemi objekty scény. Díky tomu je možné stanovit kandidátní objekty, které se nacházejí pouze v oblastech scény, kterými paprsek skutečně prochází [9][16]. Tím se vynechá nezanedbatelné množství testů kolizí velkých skupin trojúhelníků.



Obrázek 2.6: Ukázka hierarchického členění scény pomocí oktalového stromu

2.2.1 Budování stromu

Při stavbě oktalového stromu se postupuje sestupně od kořenového uzlu (jedná se tedy o budování struktury shora dolů, což je jediný způsob stavby oktalového stromu). Počáteční uzel stromu se vytvoří z osově zarovnaného obalovacího tělesa scény a přidělí se mu vlastnictví všech trojúhelníků ve scéně. Tento uzel je dále rekurzivně dělen v závislosti na stanoveném limitu trojúhelníků, které může každý uzel nanejvýš obsahovat. Běžně se jako součást ukončovací podmínky využívá i limit maximální hloubky stromu.

V závislosti na potřebě vytvoření dostatečně kvalitní akcelerační struktury se běžně přistupuje k užití pokročilejších metrik než pouhého limitu geometrie na listový uzel. Mezi nejpoužívanější se řadí SAH [30], ale pro určité scény může být výhodnější užití RDH [3], která zohledňuje distribuci paprsků v prostoru.

2.2.2 Průchod stromem

Algoritmy průchodu oktalového stromu mají úlohu nalezení množiny trojúhelníků (případně jiných grafických primitiv), kterou protíná vržený paprsek. Tyto algoritmy se obecně dělí na dva možné přístupy – průchod zdola nahoru a shora dolů. Algoritmy průchodu zdola nahoru se zaměřují pouze na listové uzly stromu, které prohledávají. Do této skupiny patří například 3D DDA použitý obdobně jako v případě uniformních mřížek (vhodný pouze pro oktalové stromy s konstantní hloubkou), či alogoritmus nalezení souseda [22] (vhodný i pro stromy nekonstantní hloubky). Oba uvedené algoritmy spočívají v nalezení prvního listového uzlu, který je vrženým paprskem protnut, a pokud v něm není nalezen trojúhelník, který by paprsek též protnul, stanoví se sousední buňka, kterou paprsek dále prochází. Tak se pokračuje do první kolize s některým z trojúhelníků odkazovaných ze zkoumané buňky.

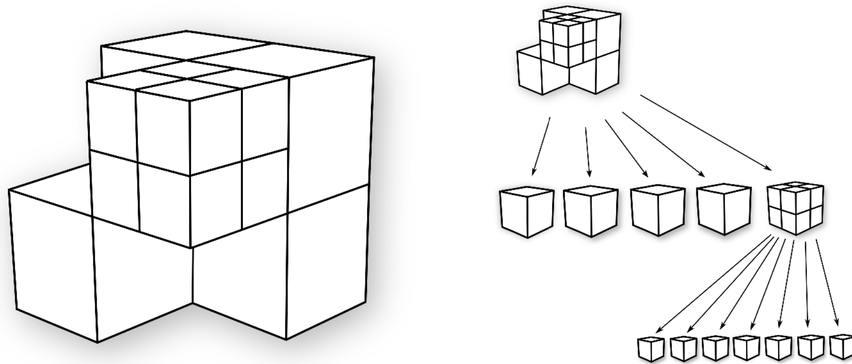
Druhým přístupem je tedy průchod stromu shora dolů. V tomto případě je strom procházen rekurzivně od kořenového uzlu. Nejvhodnější je přístup prohledávání do hloubky, protože úkolem je nalézt první kolizi paprsku s trojúhelníkem scény (a tedy další průchod stromem neprovádět). Kvůli tomu je ale nutné procházet synovskými uzly v pořadí, ve

kterém jsou paprskem zasaženy. Nejprve se tedy určí, který synovský oktant je paprskem protnut jako první, a v závislosti na směrovém vektoru paprsku se stanoví uspořádaná množina následně zasažených oktantů. Příkladem těchto algoritmů mohou být algoritmus HERO [1] a algoritmus průchodu oktalovým stromem (Revelles a spol.) [21].

2.3 Sparse Voxel Octree (SVO)

Laine a Karras [15] definují řídký voxelový oktalový strom jako analogii řídkého pole pro oktalové stromy, které tvoří prostorovou hierarchii voxelechových scén, jež obsahují pouze povrchy objektů, a nikoliv jejich celé objemy (podobně jak je tomu u trojúhelníkových scén). Řídkého charakteru je dosaženo úpravou definice oktalového stromu a to tak, že každý vnitřní uzel stromu nyní může být dělen na 1 až 8 potomků (namísto striktního dělení vždy na 8 potomků), podle toho zda oblast náležící potomkovy obsahuje některý z voxelů v konečném rozlišení (viz. obrázek 2.7). Tím se efektivně snižují paměťové nároky datové struktury a urychluje se i její průchod při prohledávání.

V SVO navíc mohou být data uchovávána i v interních uzlech struktury (vzniklá interpolací dat potomků uzlu). Tím je možné realizovat úroveň detailu popisované scény nižší než v listových uzlech. Např. při větší vzdálenosti od bodu pohledu potom není potřeba procházet strom do celé jeho hloubky, ale je možné se zastavit na nejnižší přípustné úrovni, pro danou vzdálenost. Avšak při použití SVO jako akcelerační struktury tato vlastnost pozbývá důležitosti. Pro dosažení stejného efektu by musely být v interních uzlech odkazovány modely objektů s nižší úrovní detailu.



Obrázek 2.7: U SVO, oproti oktalovému stromu, mají interní uzly 1 *až* 8 potomků

Jednotlivé vnitřní uzly stromu jsou popsány pomocí záznamů v poli deskriptorů a obsahují ukazatel na seznam potomků, masku platných potomků (ta určuje, které z osmi podčástí prostoru uzlu jsou neprázdné a strom jimi dále pokračuje) a masku listových potomků (ta indikuje, které synovské uzly jsou zároveň uzly listovými).

2.3.1 Budování stromu

Na rozdíl od obecného oktalového stromu je SVO tvořeno pro voxelové scény, a pokud mají rozlišení rovné mocnině dvou, je tedy i dopředně možné odvodit maximální hloubku požadovaného SVO a je daná i podoba jeho listových dat. Tím je umožněno přistupovat k budování struktury i způsobem zdola nahoru. V takovém případě se jednotlivé voxely

scény vkládají do struktury a průběžně se slučují do rodičovských uzlů [6]. Ovšem ani přístup budování shora dolů nelze vyloučit.

2.3.2 Průchod stromem

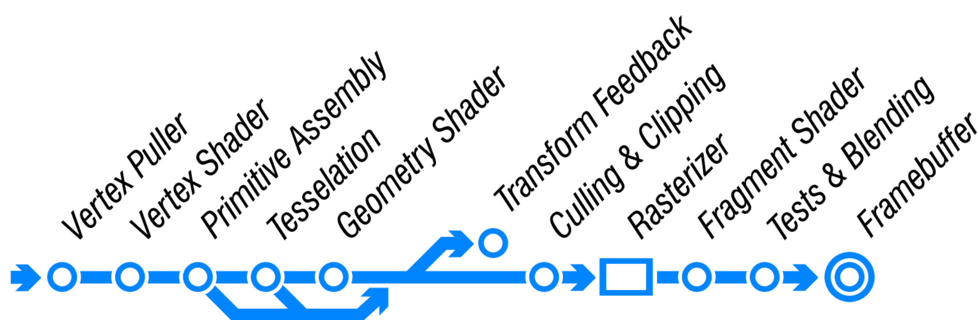
Pro průchod strukturou je možné zvolit v podstatě kterýkoliv z algoritmů pro průchod obecného oktalového stromu [8]. Značnou výhodou je však využití přístupu prohledávání shora dolů, jelikož kromě toho, že se stanovuje uspořádaná množina paprskem zasažených oktantů, je pro každý uzel SVO uchována maska znázorňující množinu neprázdných částí jeho oblasti ve scéně. Průnikem těchto dvou množin se tak ještě více redukuje počet oktantů, které musí být při prohledávání otestovány na kolizi s paprskem.

2.4 Voxelizace vykreslovacím řetězcem OpenGL

Proto aby bylo možné použít SVO jako akcelerační strukturu nejen pro voxelové scény, ale i například pro scény trojúhelníkové, je v takových případech nutné scénu voxelizovat. Každý voxel by tedy odkazoval na množinu trojúhelníků, které jej protínají. Voxelizace může být provedena jako pouze povrchová (není nutné voxelizovat celý objem trojúhelníkového modelu, ale pouze trojúhelníky tvořící jeho povrch). Naivním řešením by mohlo být rozdělení celé scény podle uniformní mřížky (v rozlišení mocniny 2) a následné vkládání vzniklých voxelů do SVO. Hned z několika hledisek je toto řešení zatíženo četnými nedostatky. Především tím, že testování průniku voxelů mřížky s jednotlivými trojúhelníky je nutné provádět i pro prázdné oblasti scény.

Možným východiskem je technika zvaná *bucketing*. Při ní se provádí testy průniku trojúhelníku pouze s oblastmi uniformní mřížky, do kterých zasahuje obalovací těleso vytvořené pro daný trojúhelník [19]. V takto vytyčených oblastech se za použití scan-line algoritmu stanoví množina voxelů, které kolidují s trojúhelníkem. Tento přístup se na první pohled velmi podobá rasterizaci trojúhelníků pro vytváření 2D obrazu.

Crassin a Green [5] uvedený přístup voxelizace dále rozvíjejí užitím vykreslovacího řetězce grafických akcelerátorů GPU (znázorněný na obrázku 2.8). Argumentují tím, že tento druh hardware je pro podobné operace vysoce optimalizovaný.



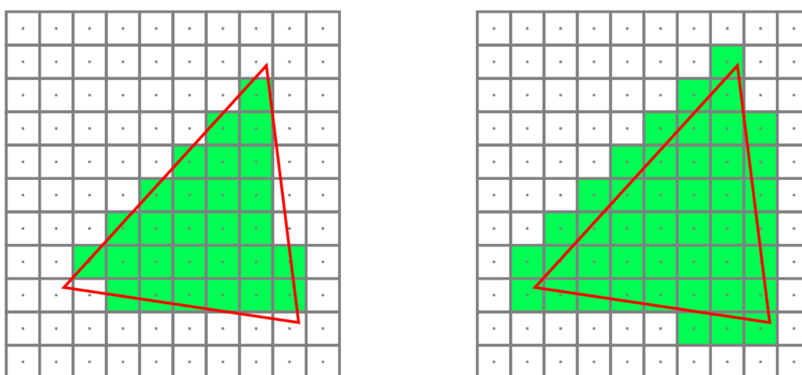
Obrázek 2.8: Schéma vykreslovacího řetězce OpenGL

2.4.1 Aplikace rasterizace při voxelizaci

Crassin a Green [5] popisují způsob, jak využít jednotlivé kroky vykreslovacího řetězce pro efektivní realizaci povrchové voxelizace trojúhelníkových modelů, a to následovně. Vertex shader nijak vstupní vrcholy netransformuje. Veškeré afinní transformace se odehrávají

v geometry shaderu, kde je nejprve proveden výběr dominantní osy normálového vektoru vstupního trojúhelníka. Poté se provede paralelní zobrazení vstupního trojúhelníka do prostoru frusta tvořeného osově zarovnaným obalovacím tělesem scény v pohledu protisměru vybrané dominantní osy normály tohoto trojúhelníka.

Takto zobrazený trojúhelník se dále rasterizuje v rozlišení shodném s požadovaným rozlišením výsledné voxelové reprezentace scény. K tomu aby při generování nebyly vynechány žádné fragmenty (především oklolo hran trojúhelníka, nebo v případě, že trojúhelník má subpixelovou velikost), musí daný systém podporovat (případně emulovat) tzv. konzervativní rasterizaci. Porovnání se standardní rasterizací je ukázáno na obrázku 2.9. V závislosti na implementaci v hardware může docházet ke generování falešně pozitivních případů, což není významný problém, ale dochází tak k mírnému snižení kvality výsledné voxelizace. Konzervativní rasterizace je dostupná na současném hardware, který podporuje DirectX 11.3, nebo DirectX 12[1], rozšíření OpenGL 4.3 NV_conservative_raster pro GPU výrobce NVIDIA[2], či rozšíření OpenGL 4.5 INTEL_conservative_rasterization pro GPU výrobce Intel[3]. Není-li však dostupná, musí být emulována expanzí hran trojúhelníka směrem ven (o vzdálenost odpovídající půl pixelu) v geometry shaderu a následným ořezáním redundantních fragmentů (vznikajících především v okolí vrcholů, jejichž ramena svírají ostrý úhel) ve fragment shaderu [11].



Obrázek 2.9: Standardní rasterizace (vlevo) generuje fragmenty jen pro pixely, jejichž střed leží uvnitř trojúhelníka. Konzervativní rasterizace (vpravo) generuje fragmenty pro všechny pixely, které jsou trojúhelníkem protnuty.

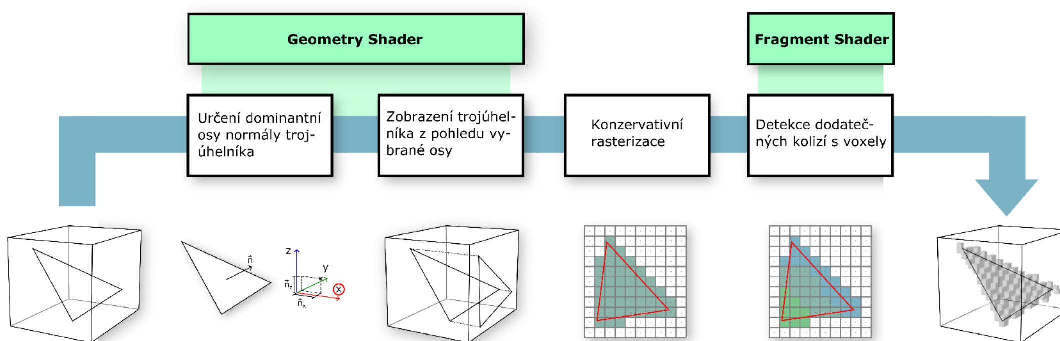
Alternativou ke konzervativní rasterizaci může být například užití rasterizace s multisamplingem, který se běžně používá pro eliminaci aliasingu (MSAA). Tento přístup se uplatňuje například při voxelizaci s ohledem na průhlednost objektů ve vykreslovacím systému GigaVoxels [4] pro výpočet globálního osvětlení scény pomocí její voxelové reprezentace. Samotná voxelizace tak probíhá promítnutím trojúhelníka do tří rovin kolmých na osy souřadného systému a odvozením protnutých voxelů. Multisampling zde pouze snižuje množství případů, kdy při rasterizaci dochází k nepřesnému generování fragmentů. Tento problém však zcela neřeší a nadále tak dochází k falešně negativním případům, které jsou pro další využití voxelové reprezentace scény pro budování akceleračních struktur nežádoucí.

Ve fragment shaderu se musí určit, zda trojúhelník neprotíná více voxelů se shodnou dvojicí souřadnic x a y (v koordinátech rasterizovaného obrazu). K tomu se využije faktu,

[1] https://docs.microsoft.com/cs-cz/windows/win32/direct3d12/conservative-rasterization
[2] https://www.khronos.org/registry/OpenGL/extensions/NV/NV_conservative_raster.txt
[3] https://www.khronos.org/registry/OpenGL/extensions/INTEL/INTEL_conservative_rasterization.txt

že každý fragment má uchovánu svou hloubkovou souřadnici z i přes to, že rasterizace je definována jako proces projekce trojrozměrného objektu do roviny (v praxi se uchování hloubky jeví jako vhodné pro realizaci složitějších obrazových efektů). Navíc je možné získat parciální derivace plochy trojúhelníka (opět v prostoru obrazu), které se společně se souřadnicí z fragmentu použijí k výpočtu kolize plochy trojúhelníka s dalšími voxely právě na ose z. Následně se všechny vzniklé voxely promítnou do prostoru voxelové mřížky a s jejich souřadnicemi a informacemi o tom, který trojúhelník posloužil k jejich vygenerování, se zapíší do bufferu typu SSBO (angl. shader storage buffer object). K zabránění souběhu se použije atomický čítač sloužící jako alokátor pro záznamy v tomto bufferu. Zápis fragmentů do framebufferu neprobíhá, není totiž potřebný. Celý proces je znázorněný na obrázku 2.10.

Výstupem této operace je tak pole tzv. voxelových fragmentů. Jeden voxel mřížky v něm může mít více záznamů odpovídajících trojúhelníkům, které jej protínají. Touto metodou vzniklé záznamy mají ale velmi špatnou prostorovou lokalitu a nejsou nijak seřazeny.



Obrázek 2.10: Vizuální znázornění jednotlivých kroků voxelizace

Kapitola 3

Návrh

Existuje široké spektrum možných řešení urychlování vykreslování zohledněním prostorového rozmístění jednotlivých prvků 3D scény, které je reprezentované akcelerační strukturou. Většina současných metod uplatňuje přístup, kdy je akcelerační struktura stavěna pouze pro statické scény, v předem neurčeném čase, s kritériem splnění požadavků předem určené úrovně kvality (např. pomocí určení odhadu ceny průchodu strukturou).

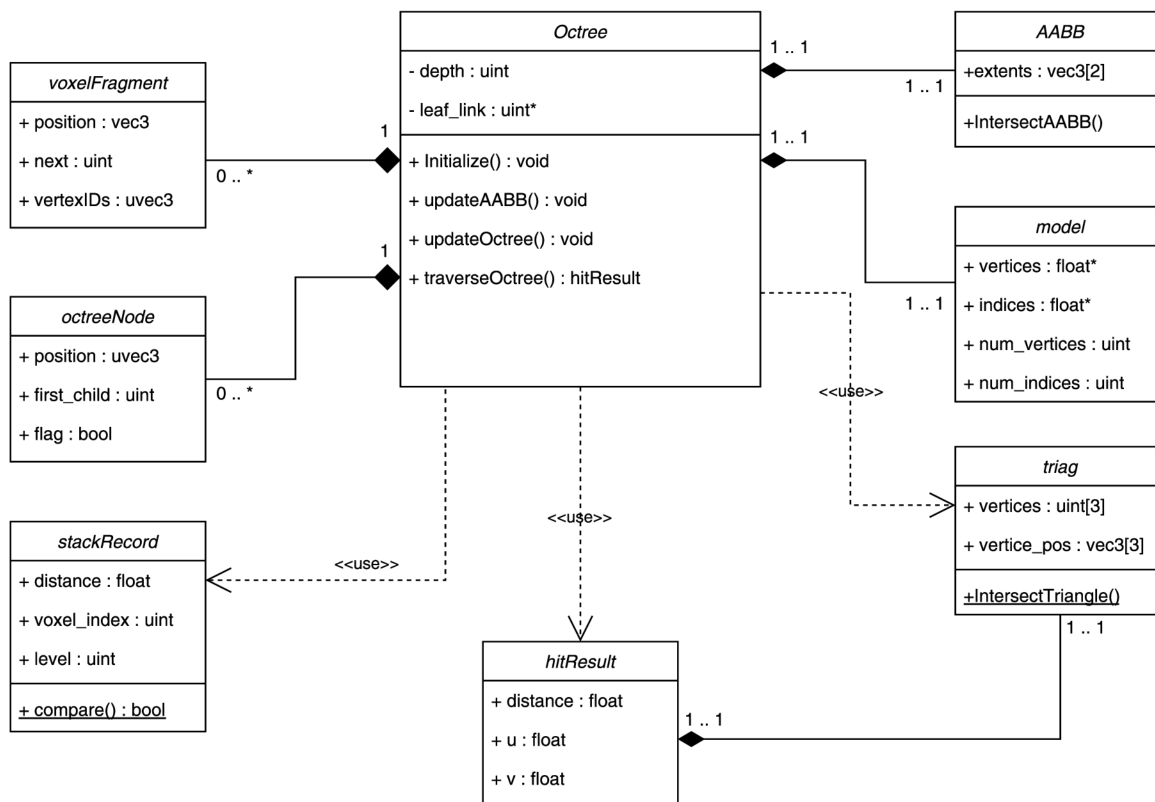
Pro účely této práce byl jako akcelerační struktura zvolen oktalový strom, protože v nedávné době zaznamenal zajímavý vývoj z hlediska jeho řídké varianty – SVO. Tyto nové poznatky je zároveň možné použít pro optimalizaci stavby původního typu struktury. Při stavbě oktalového stromu je za nejužší místo výpočtu tradičně považováno zjištění kolizí trojúhelníků scény s osově zarovnanými kvádry znázorňujícími buňky uzlů stromu v prostoru. Pokud se ale upustí od zaběhlého přístupu ke stavbě oktalového stromu rekurzivním dělením uzlů, který je ukončen až po dosažení zvoleného kritéria, a namísto toho se struktura vybuduje s přesně definovanou hloubkou, nemusí nutně dojít ke vzniku výrazně méně kvalitní struktury. Pokud je hloubka stromu definována předem, je možné nahradit výpočetně náročný proces testování kolizí za proces konzervativní povrchové voxelizace. Ten je možné vykonat na grafickém hardware určeném pro rasterizaci, který je na ji podobnou voxelizaci možné přizpůsobit. Současný grafický hardware je schopný vykreslovat miliony trojúhelníků za vteřinu. Jeho využitím by tak mělo dojít k radikálnímu zrychlení procesu stavby stromu, jelikož proces by byl implicitně paralelizován.

Jako součást této práce vzniká knihovna pro stavbu a použití oktalového stromu. Jejím cílem je poskytnout jednoduché rozhraní pro specifikaci vstupní scény a požadovaných parametrů vytvářené struktury ale také poskytnutí funkcionality průchodu stromu.

3.1 Knihovna pro práci s oktalovým stromem

Rozhraní knihovny sestává pouze z jedné třídy, kterou je třída *Octree*. Ta slouží jako abstrakce oktalového stromu a zastřešuje jeho vnitřní reprezentaci, stejně tak i základní operace s touto akcelerační strukturou. Těmito operacemi jsou: konfigurace, budování struktury a průchod strukturou. Objekt této třídy je inicializovaný uživatelským vstupem dodáním dat trojúhelníkové reprezentace scény a nastavením požadovaného parametru hloubky výstupní akcelerační struktury. Třída kromě metod pro sestavení oktalového stromu zahrnuje metodu prohledávání struktury do hloubky pro detekci kolize s paprskem.

Na obrázku 3.1 je znázorněn UML diagram tříd, který popisuje podobu a vztah třídy oktalového stromu ku ostatním pomocným třídám. V diagramu nejsou uvedeny privátní metody ani vstupní parametry metod pro zvýšení čitelnosti.



Obrázek 3.1: Diagram tříd knihovny. Jádrem knihovny je třída Octree, která je vázána na řadu podpůrných tříd.

Třída pro uchování a sestavení oktalového stromu je přístupná přes metody svého veřejného rozhraní. Tyto metody umožňují uživateli definovat model scény, pro kterou má být oktalový strom sestaven, nastavit požadovanou hloubku vygenerované akcelerační struktury, stanovit ,či automaticky vypočítat tvar osově zarovnaného obalovacího tělesa a následně spustit algoritmus stavby oktalového stromu. Vygenerovaný oktalový strom je možné procházet metodou vyhledávání paprskem zasaženého trojúhelníku vstupního modelu.

3.1.1 Inicializace

Během inicializace je potřeba definovat parametr hloubky pro generování oktalového stromu. Dále je potřeba uložit odkaz na vrcholy a indexy modelu uživatelem poskytnuté scény. Pro potřeby algoritmu stavby stromu je také nutné znát tvar osově zarovnaného obalovacího tělesa dodaného modelu scény. Uživatel jej může definovat ručně, nebo jej nechat vygenerovat ze vstupního modelu. Výpočet probíhá na principu nalezení minimálních a maximálních souřadnic x, y a z vrcholů vstupního trojúhelníkového modelu. Ty totiž mají význam vrcholů osově zarovnaného tělesa, které leží na protilehlých koncích jeho tělesové úhlopříčky. Kvůli omezení, které vyplývá z použití vykreslovacího řetězce při pozdějším budování akcelerační struktury, je nutné takto vytvořené obalovací těleso zvětšit ve všech rozměrech přičtením malé hodnoty.

3.1.2 Stavba akcelerační struktury

Navrhovaný postup popisuje proces, kdy je oktalový strom budován ve třech krocích. Prvním je vytvoření voxelové reprezentace scény, kdy jednotlivé voxely popisují v nich obsažené trojúhelníky. Dále se z těchto voxelových dat sestaví struktura prázdného stromu, tak aby odpovídala výsledné podobě akcelerační struktury. Nakonec jsou voxelová data provázána s listy stromu.

Jak již bylo předneseno, nejprve je nutné scénu voxelizovat, jak je popsáno v podkapitole 2.4. Pro jednotlivé touto metodou vytvořené voxelové fragmenty je následně potřeba sestavit samotnou strukturu oktalového stromu. I přes to, že strukturu je možné stavět zdola nahoru (vzhledem k tomu, že je sestavována pro předem danou množinu voxelů), data by musela být nejprve seřazena podle jejich sousednosti v prostoru. Nutnost data řadit by ovšem mohla vést k nechtěným časovým prolukám, a proto je namísto toho zvolen postup budování struktury shora dolů, kdy se sestupuje po jednotlivých úrovních stromu od jeho kořene, podle popisu Crassina a Greena [5]. Algoritmus postupuje následovně:

Algorithm 1 Stavba struktury oktalového stromu

Vstup: Množina voxelových fragmentů V, hloubka stromu h **Výstup:** Množina uzlů stromu O, množina voxelových fragmentů V, seznam listových dat L

1: Vlož 8 prázdných záznamů uzlů první úrovně stromu
2: **for** i=1 **to** H-2 **step** 1 **do**
3: Označ uzly z O na úrovni i, jejichž následovníkem je některý prvek z V
4: Vlož 8 prázdných potomků označených uzlů do O
5: **end for**
6: Inicializuj seznam hlaviček lineárních seznamů listových dat
7: Propoj prvky z množiny V do lineárních seznamů listových dat

Prvním krokem algoritmu 1 je vytvoření prvních 8 záznamů pro uzly stromu na první úrovni. Kořen na nulté úrovni existuje vždy a pokud strom obsahuje alespoň jeden trojúhelník existuje při hloubce stromu 2 i všech 8 těchto uzlů, což vyplývá ze samotné definice oktalového stromu. Pro všechny existující záznamy uzlů stromu na poslední vložené úrovni je potřeba zjistit, zda mají nějaké následovníky a zda budou muset být tyto uzly dále děleny. Pro uzly, pro které je rozhodnuto, že se dělit mají, pak musí vzniknout 8 nových záznamů pro jejich potomky. Do struktury se tak vkládají záznamy uzlů další úrovně. Proces určování uzlů k dělení a samotné dělení uzlů je opakován, dokud není dosaženo listové úrovně stromu.

Strukturu stromu je nakonec nutné provázat s listovími daty. Nejprve jsou tak voxelové fragmenty, které mají shodný údaj o jejich pozici, provázány do jednosměrně vázaného seznamu a odkaz na ně je možné vyvodit z pozice odpovídajícího uzlu oktalového stromu na listové úrovni.

3.1.3 Průchod strukturou

Stromovou strukturou se prochází v závislosti na parametrech vrženého paprsku. Úkolem této metody je totiž vyhledat z akcelerační struktury odkazovaný trojúhelník, který byl uživatelem zadaným paprskem jako první zasažen. Strukturou je tedy nutné procházet ne popořadě podle toho, jak jsou potomci uzlů číslováni, ale podle toho v jakém pořadí jsou sami paprskem zasaženi.

I přes to, že by bylo možné v akcelerační struktuře vyhledávat podle vzoru algoritmu 3D DDA, nevyužil by se plný potenciál hierarchického charakteru oktalového stromu a efektivita jeho průchodu by byla prakticky degradována na tu náležící uniformní mřížce. Zvolený postup je tedy odlišný.

Navrhovaná metoda vychází z dříve uvedeného Revellesova algoritmu a algoritmu HERO (podkapitola 2.2.2). Jedná se tedy o prohledávání do hloubky shora dolů. Rekurze je zde však nahrazena iterací bez předem známého počtu opakování s využitím zásobníku. Algoritmus vypadá následovně:

Algorithm 2 Prohledávání oktalového stromu

Vstup: Prázdný zásobník Z, množina uzlů stromu O, paprsek p
Výstup: Nalezený trojúhelník t

```
 1: Přidej do Z záznam kořenového uzlu stromu
 2: while Z není prázdný do
 3:     Vyjmi poslední přidaný záznam ze Z a ulož ho do n
 4:     if n je listovým uzlem then
 5:         Otestuj trojúhelníky z n na kolizi s p a ulož ten nejbližší do t
 6:         if t je validní then
 7:             return t
 8:         else
 9:             Pokračuj následující iterací
10:         end if
11:     end if
12:     Získej seznam potomků uzlu n z O v pořadí podle zásahu a ulož jej do C
13:     Vkládej prvky z C v opačném pořadí na vrchol zásobníku
14: end while
15: return neplatný
```

Tím, že při průchodu stromu se potomci od kořenového uzlu dále procházejí v pořadí ve kterém byli zasaženi, je možné vyhledávání ukončit u prvního koncového uzlu stromu, který obsahuje alespoň jeden protnutý trojúhelník. Jako výsledek je pak navrácen ten, který byl protnutý v nejbližší vzdálenosti k počátečnímu bodu paprsku. Jsou-li navíc trojúhelníky modelu rozmístěny v obalovacím tělese rovnoměrně, existuje vysoká pravděpodobnost, že některý z trojúhelníků scény bude zasažen v jednom z prvních zkoumaných listových uzlů.

Během každého sestupu po jednotlivých úrovních stromu podle algoritmu 2 se musí nejprve určit, které synovské uzly jsou paprskem zasaženy a v jakém pořadí.

Výpočet probíhá pro všech osm potomků daného uzlu a ty, které jsou paprskem protnuty, následně tvoří seznam zasažených potomků. Ten je nakonec seřazen podle toho, v jaké vzdálenosti od počátečního bodu paprsku byli zasaženi. Následujícím zkoumaným uzlem tak bude ten, jehož vzdálenost je nejmenší.

Narazí-li algoritmus při průchodu stromem na listový uzel, musí pak otestovat všechny trojúhelníky, které daný listový uzel obsahuje. Nejprve proto musí vypočítat mortonovský kód pozice zkoumaného uzlu, který je následně použit jako index do seznamu hlaviček jednosměrně vázaných seznamů listových dat. Pokud záznam seznamu na konkrétním indexu obsahuje nevalidní hodnotu, daný uzel neodkazuje na žádné trojúhelníky scény a je tak nutné pokračovat v průchodu strukturou stromu dalším uzlem v pořadí. V opačném případě je nutné prohledat celý odkazovaný seznam a jednotlivé v něm obsažené trojúhelníky podrobit testu průniku s vrženým paprskem.

Pokud nebyl žádný z trojúhelníků zasažen, opět se pokračuje vyšetřováním následujícího uzlu stromu v pořadí. Pokud však byly některé z nich paprskem protnuty, vybere se pouze ten nejblíže zasažený. Tento trojúhelník je navrácen jako výsledek a algoritmus tak úspěšně končí.

Kapitola 4

Implementace

V této kapitole jsou popsány implementační detaily výstavby oktalového stromu na grafickém hardware. Operace pro vytvoření a práci s ním byla implementována knihovny jazyka C++, což umožní snadné použití představené metody v budoucnu vznikajících aplikacích. Funkcionalita knihovny je předvedena na ukázkové aplikaci popsané v podkapitole 4.5.

Použité technologie. Implementace aplikační části práce je realizována v programovacím jazyce C++. K programování činnosti grafického procesoru je využito aplikační rozhraní OpenGL, které je zpřístupněné obalovací knihovnou GeGL v rámci knihovny pro vykreslování 3D scén GPUEngine. Ta totiž poskytuje uživatelsky přívětivější způsob přístupu k rozhraní grafické knihovny jejím zapouzdřením do tříd objektově orientovaného programovacího přístupu. Rovněž je využita knihovna GLM, která implementuje datové typy jazyka GLSL pro použití v jazyce C++. Dále je použit systém CMake pro automatizaci překladu výsledné knihovny na různých platformách. Ke správě verzí zdrojových souborů je využíván distribuovaný verzovací systém Git. Knihovna je zaměřena pouze na grafické karty výrobce NVIDIA z důvodu dostupnosti nezbytně nutné funkcionality konzervativní rasterizace potřebné pro zásadní součást implementovaného algoritmu.

4.1 Třída OB::Octree

Knihovna pro budování oktalového stromu *octreeBuilder* poskytuje třídu *OB::Octree*, která je navržena tak, aby sloužila jako abstrakce nad konkrétním oktalovým stromem vstupní scény. Rozhraní třídy slouží ke konfiguraci a spuštění algoritmu stavby oktalového stromu.

Největším problémem implementace knihovny s použitím OpenGL je to, že není možné spouštět příkazy OpenGL bez přiděleného grafického kontextu a tedy i okna grafického uživatelského prostředí daného operačního systému. Možným řešením by mohlo být použití knihovny EGL (vyvíjí ji Khronos Group, která rovněž spravuje OpenGL), který je univerzálním rozhraním tvorby grafických kontextů napříč různými platformami. Problémem je ale jeho nedostatečná podpora pro desktopové systémy, což jeho použití v této práci brání. Bohužel z tohoto pro uživatele knihovny plyne nepříjemná povinnost spravovat přepínání kontextu před použitím některých metod třídy *OB::Octree* (konkrétně se jedná o metody `initWithGL()` a `updateOctree()`).

Navíc to, že kontext pro tuto knihovnu může uživatel sdílet i pro potřeby vlastního vykreslování, může vést ke ztrátě připojení (angl. binding) uživatelských bufferů s připojovacími místy OpenGL daného kontextu (pokud se shodují jejich indexy s těmi, které používá

knihovna). Tento problém neplatí pro buffery připojené jako SSBO, jelikož jejich identifikátory je možné zpětně zjistit podle toho, na který index připojovacího bodu jsou vázány. To se ovšem netýká bufferů atomických čítačů. Atomické čítače na indexech 0 a 1 tak budou při volání některých metod třídy odpojeny.

Inicializace OpenGL. Při inicializaci se vytvářejí objekty jednotlivých bufferů. Jmenovitě se jedná o buffer voxelových fragmentů, buffer uzlů oktalového stromu a buffer pro provázání listových dat stromu. Vytváří se zde i atomické čítače, které slouží pro alokaci prostoru ve dříve zmíněných bufferech. Dále se sestavují shader programy a vytváří se objekt pole vrcholů (angl. vertex array object, zkráceně VAO). Jako vedlejší efekt se odpojují buffery atomických čítačů vázaných na indexech 0 a 1, s čímž musí uživatel knihovny počítat.

Budování. Třída *OB::Octree* také provádí režii algoritmu stavby oktalového stromu, který je vykonáván na grafickém procesoru. Skládá se ze tří částí: voxelizace scény, stavby struktury stromu z vytvořené voxelové reprezentace a napojení listových dat. Podrobně je tento postup popsán v podkapitolách 4.3 a 4.4. Konfiguruje se chování a funkcionalitu grafického procesoru pomocí rozhraní OpenGL a spouští příkazy provedení programů popsaných příslušnými shadery.

Průchod. Stromovou strukturou se prochází v závislosti na atributech paprsku vrženém skrze scénu. Při průchodu je totiž nutné nalézt zasažený trojúhelník modelu spravovaného daným objektem oktalového stromu. Pro účely zjištění zasažených potomků uzlů při průchodu stromem je použito modifikovaného výpočtu podle Williamsové [29] pro určení průniku polopřímky s osově zarovnaným kvádrem, kterým je možné reprezentovat zkoumaný oktant. K určení kolize paprsku s trojúhelníkem během vyšetřování listových dat je použit Möllerův algoritmus [18].

4.2 Příklad použití knihovny

Celá funkcionalita knihovny je dostupná z rozhraní třídy *OB::Octree*. Jak je ukázáno v úryvku kódu 4.1 uživatel nejprve vytvoří její instanci. Následně musí přepnout grafický kontext, který vytvořil ve frameworku pro grafická uživatelská rozhraní, který v aplikaci používá, na ten, který chce, aby knihovna *octreeBuilder* využívala. Teprve po té může uživatel zahájit inicializaci objektů OpenGL nutných pro potřeby výpočtu. Poté je nutné dodat model trojúhelníkové reprezentace scény a specifikovat její obalovací těleso. Zároveň je potřeba nastavit požadovanou hloubku výsledného stromu.

Předchozí metody je možné volat v libovolném pořadí. Jsou-li všechny úspěšně dokončeny, je pak možné zahájit samotný proces výstavby oktalového stromu. Hotový strom je možné procházet dotazem na nalezení kolize paprsku s trojúhelníkem z akcelerační struktury nejbližšího ku počátečnímu bodu daného paprsku.

```
1   OB::Octree myOctree; //tvorba instance tridy
2   OB::hitResult triangle;
3   Context->makeCurrent(); //prepnuti aktivniho kontextu
4   myOctree.initWithGL(); //inicializace OpenGL buffer objektu
5
6   //dodani modelu, pro ktery ma byt oktalovy strom vystaven
7   myOctree.setModel(vertices, indices, num_vertices, num_indices);
8
9   myOctree.setDepth(5); //nastaveni pozadovane hloubky stromu
10  myOctree.updateAABB(); //vypocteni tvaru AABB
11  myOctree.updateOctree(); //vygenerovani akceleracni struktury
12
13  //vrzeni paprsku ze zadaneho bodu a smeru; ulozeni nalezeneho trojuhelniku
14  triangle = myOctree.traverseOctree({0.f, 0.f, 0.f}, {1.f, 0.f, 0.f});
```

Výpis 4.1: Ukázka použití knihovny octreeBuilder v jazyce C++

4.3 Voxelizace scény

Tento proces se skládá z kroků, které jsou přímo prováděny vykreslovacím řetězcem OpenGL. Jednotlivé úkony jsou tak implicitně paralelizovány. To přináší nutnost věnovat zvýšenou pozornost datovým konfliktům. Voxelizaci scény v této implementaci je možné provést 1 voláním vykreslovacího příkazu (za předpokladu, že je celá scéna obsažena v jednom objektu pole vrcholů).

Jednou z prerekvizit tohoto procesu je nutnost vypočítat tvar osově zarovnaného tělesa (AABB) scény – ten bude předán shader programu jako uniformní proměnná. Je také nutné konfigurovat rasterizér pro vykreslování konzervativní rasterizací povolením funkcionality `CONSERVATIVE_RASTERIZATION_NV` příkazem `glEnable()`. Zároveň je potřeba nastavit rozlišení rasterizace příkazem `glViewport()` na rozlišení odpovídající požadovanému rozlišení výsledné scény. V případě voxelizace bude toto rozlišení rovno 2^{h-1}, kde h představuje maximální hloubku požadovaného oktalového stromu. Z důvodu toho, aby bylo možné vygenerované voxelové fragmenty adresovat pomocí 32-bitového lineárního indexu (mortonovsky kódovaným), je maximální hloubka stromu omezena implementačním limitem na hodnotu 10.

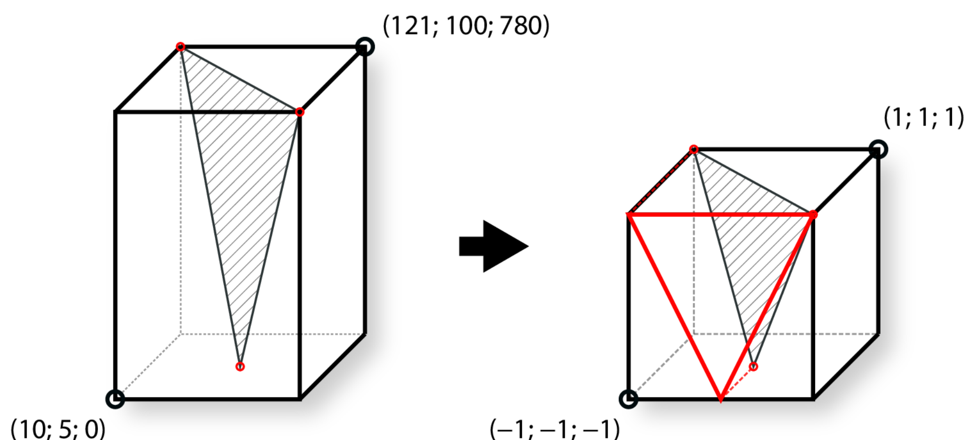
V poslední řadě je nutné zabránit zápisu fragmentů do framebufferu. K tomu slouží příkaz `glColorMask()`. Shader program vykonávající voxelizaci je sestaven z programovatelných částí vykreslovacího řetězce OpenGL. Konkrétně se jedná o vertex shader, geometry shader a fragment shader s níže popsaným chováním.

Vertex shader. Pro jednotlivé vrcholy není v kontextu popisovaného procesu nutné vykonávat jakékoliv operace. Vertex shader zde tedy slouží pouze k získání atributů pozice vrcholů, které jsou zapisovány do výstupní proměnné `gl_Position`.

Geometry shader. Po sestavení primitiv jsou již dostupné informace o celých trojúhelnících. Trojúhelníky musí být připraveny pro následnou rasterizaci. Jednotlivé fragmenty vzniklé rasterizací budou odpovídat v mřížce uspořádaným voxelům v prostoru vytyčeném AABB scény. Z tohoto důvodu je nutné trojúhelník transformovat do lokálního prostoru daného AABB. Vhodné je zvolit rozsah tohoto lokálního prostoru tak, aby $x, y, z \in (-1; 1)$.

Tento rozsah je volen z toho důvodu, že odpovídá rozsahu ořezového prostoru (angl. clip-space), ve kterém se musejí nacházet výstupní souřadnice bodů trojúhelníka před rasterizací.

Výše zmíněná transformace se skládá z operací translace, škálování a promítnutí do bází v závislosti na zvoleném pohledu. Nejprve je tedy trojúhelník posunut podle vektoru s počátkem ve středu AABB a koncem v počátku souřadného systému scény. Tím střed AABB bude odpovídat středu ořezového prostoru. Následně je trojúhelník škálován polovinou velikosti AABB v jeho jednotlivých rozměrech. Takto je oblast AABB celá obsažena v rozsahu $(-1; 1)$.



Obrázek 4.1: Transformace trojúhelníka do prostoru obalovacího tělesa a jeho promítnutí

V dalším kroku je nutné vybrat paralelní pohled, ze kterého má být trojúhelník rasterizován. Je záhodné, aby byl zvolen pohled, ze kterého má promítnutý trojúhelník největší plochu. Tím se totiž vygeneruje největší množství fragmentů při rasterizaci. Potřebný pohled se určí jako ten, který je protisměrný ku kladné poloose osy souřadného systému, která odpovídá souřadnici normálového vektoru trojúhelníka s nejvyšší absolutní hodnotou. Zvolený pohled je poté reprezentovám transformační maticí, která je předána ve výstupní proměnné, aby v pozdějších fázích procesu voxelizace mohlo dojít ke zpětnému zobrazení do jednotného pohledu pro všechny trojúhelníky.

Aspekt, který není vhodné opomenout, je fakt, že rozsah ořezového prostoru je otevřeným intervalem. Trojúhelník, který leží na stěně obalovacího tělesa scény, by v takovém případě nebyl vůbec rasterizován. Aby k tomuto nedocházelo, je AABB mírně zvětšeno o malý odstup.

Fragment shader. Rasterizací vytvořené fragmenty odpovídají voxelům, které byly protnuty některým z trojúhelníků scény. Avšak užitím dvourozměrné rasterizace nemohlo dojít k nalezení všech protnutých voxelů. Pro jednu uspořádanou dvojici koordinát $[x, y]$ byl vygenerován nanejvýš 1 fragment na 1 trojúhelník. Pokud tak trojúhelník zasahuje do více než jednoho voxelu se stejnými souřadnicemi x a y v různé hloubce, bude i přes to vygenerován pouze 1 fragment. Z tohoto důvodu je nutné nezahrnuté voxely identifikovat.

Cílem fragment shaderu je tedy zjistit, zda rovina trojůhelníka neprotíná více voxelů před i za fragmentem. Vzhledem k tomu, že rovina trojúhelníka nemůže být kvůli jeho předchozímu promítnutí v geometry shaderu kolmá na kteroukoliv z os x a y prostoru obrazu (angl. screen-space), dostačuje tento test provést pouze pro hrany voxelu, které jsou rovnoběžné s osou z. Fragmenty jsou obecně vzorkovány v jejich středu. Pozice rohových

bodů fragmentu je tak možné vyjádřit jako $F + (\pm 0,5; \pm 0,5; 0)$, kde F je pozice fragmentu v prostoru obrazu.

Například souřadnici z průniku P roviny trojúhelníka s pravou spodní hranou voxelu rovnoběžnou k ose z lze vyjádřit jako $P_z = F_z + d_x \cdot 0,5 + d_y \cdot 0,5$, kde F je střed fragmentu, d_x a d_y jsou parciální derivace roviny trojůhelníka podle x a y prostoru obrazu.

K výpočtu parciálních derivací poskytuje OpenGL funkce `dFdx()` a `dFdy()`. Ty jsou vyhodnoceny na základě rozdílu hodnoty jejich vstupního výrazu v bloku 2×2 fragmentů. Tímto způsobem je možné získat parciální derivace elevace povrchu v doméně hloubky obrazu.

Ve výsledku je podstatné pouze zjistit maximální a minimální hloubku, ve které je kterýkoliv ze za sebou ležících voxelů rovinou protnut. Vzhledem k tomu se použijí pouze absolutní hodnoty získaných derivací. Minimální hloubka se určí jako $h_{min} = F_z - |d_x| \cdot 0.5 - |d_y| \cdot 0.5$ a maximální hloubka jako $h_{max} = F_z + |d_x| \cdot 0.5 + |d_y| \cdot 0.5$.

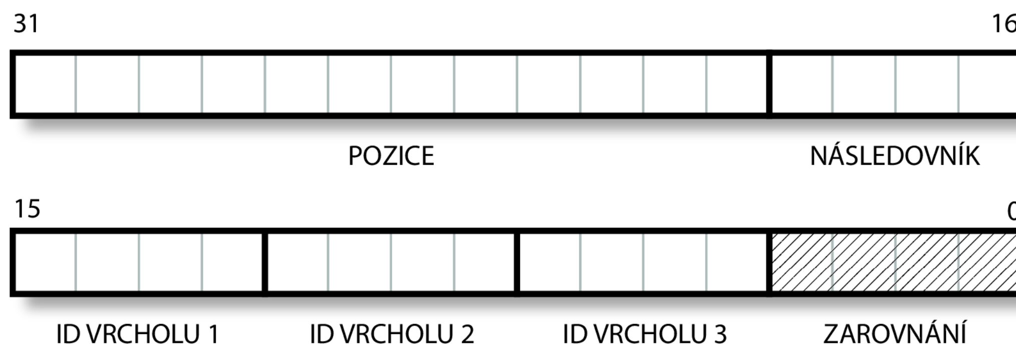
Z toho je možné odvodit (vzhledem k tomu že voxely mají v prostoru obrazu jednotkové rozměry) kolik voxelů bylo zasaženo před i za voxelem odpovídajícím zpracovávanému fragmentu. Všechny tyto voxely je nutné zapsat do příslušného bufferu typu SSBO. Vykonávání jednotlivých fází vykreslovacího řetězce je ale v hardware masivně paralelizováno. K tomu aby nedocházelo k souběhu, jmenovitě k datovým kolizím, užívá se atomického čítače, který slouží jako alokátor ve výstupním bufferu. Každé vlákno tak před zápisem inkrementuje hodnotu čítače a zároveň získává jeho původní hodnotu pomocí atomické operace `atomicCounterAdd()`. Získaná hodnota slouží jako alokací přidělený index, na jehož pozici ve výstupním bufferu může vlákno bezpečně zapisovat.

Před zápisem je potřeba voxely podrobit inverzní transformaci pohledu za pomoci předané transformační matice, která je vstupním atributem fragmentu. Zapisované informace o fragmentu popisují jeho pozici ve voxelové mřížce a identifikátory vertexů primitiva, které k vygenerování voxelu přispělo. Zapsat přímo identifikátor primitiva není vhodné, jelikož tento identifikátor se neshoduje s indexem daného trojúhelníka v objektu bufferu prvků. Jiné je to pro identifikátory vrcholů, které se skutečně shodují s indexy do objektu bufferu vrcholů.

Procesem voxelizace může být vytvořeno i více voxelových fragmentů se shodnou pozicí. U nich se ale liší identifikátory vrcholů trojúhelníka, ze kterého byl voxelový fragment vygenerován. Množiny takových fragmentů budou v následujících krocích stavby oktalového stromu shlukovány a budou sloužit jako listová data stromu.

Nutné je také podotknout, že test kolizí roviny trojúhelníka s nezahrnutými voxely neuvažuje ohraničení daného trojúhelníka. Tím může docházet k výskytu falešně pozitivních případů. Není ovšem příliš pravděpodobné, že by k nim mělo docházet v nezanedbatelně velké míře, vzhledem k volbě pohledu v geometry shaderu.

Struktura výstupních voxelů. Výstupem operace voxelizace je pole voxelů, kde všechny z nich byly vytvořeny z jednotlivých vstupních trojúhelníků pro konkrétní pozice voxelové mřížky v prostoru podle toho, které buňky této mřížky byly trojúhelníky protnuty. Tím však takto vzniklá data obsahují zdánlivé duplicity. Pro jednu pozici v prostoru totiž může existovat více vygenerovaných záznamů, které se liší pouze identifikátorem primitiva, které přispělo k jejich vygenerování.



Obrázek 4.2: Bytová reprezentace struktury voxelových fragmentů. Od nejvíce významného bytu: vektor pozice buňky ve voxelové mřížce, odkaz na následovníka pro zřetězení záznamů a indexy vrcholů obsaženého trojúhelníka.

Jak je ukázáno na obrázku 4.2, jednotlivé voxelové fragmenty jsou popsány svou pozicí ve voxelové mřížce, dále identifikátory trojice vrcholů trojúhelníka , který daný voxel protíná, a odkazu na další voxelový záznam se stejnou pozicí. Tento odkaz poslouží v pozdější fázi procesu stavby struktury k vytvoření jednosměrně vázaného seznamu voxelových záznamů jakožto listových dat výsledného oktalového stromu. Poslední 4 byty jsou prázdné a slouží pouze k zarovnání struktury v paměti na násobek 16 bytů podle pravidla `std430` uspořádání dat bufferu.

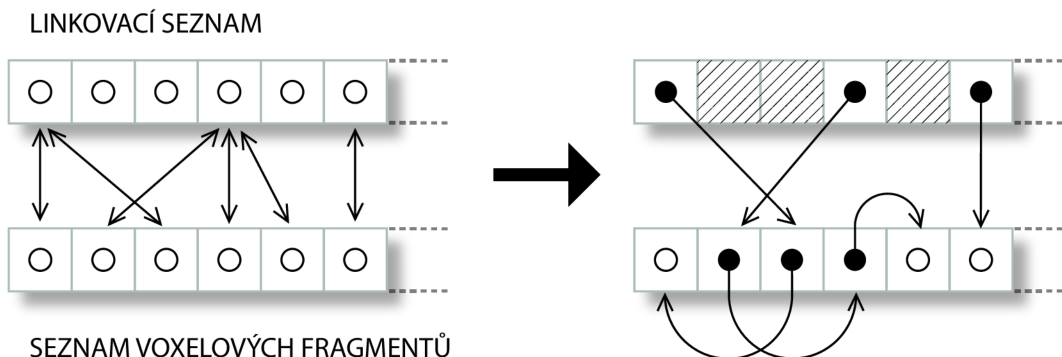
4.4 Tvorba struktury stromu

Po voxelizaci scény je možné sestavit strukturu samotného oktalového stromu. Algoritmus je implementován pomocí compute shaderu OpenGL. Vlákna compute shaderu spuštěná pro jednotlivé voxelové fragmenty sestupují stromem na aktuální úroveň. Uzel do kterého vlákno sestoupilo je označen nastavením příznaku nalezení následovníka. V dalším kroku se zkontroluje, které uzly stromu na aktuální úrovni mají nastavený příznak nalezení následovníka a pro tyto se vloží 8 potomků do bufferu uzlů stromu (opět se používá atomický čítač pro alokaci prostoru v bufferu). Tyto potomky je ale třeba nejprve inicializovat a to zápisem jejich pozice, zneplatněním jejich odkazů na potomky a vynulováním příznaku nalezení následovníků. Odkazy na ně se zapíší do uzlu, ze kterého byli vygenerováni.

První dva kroky algoritmu se opakují pro všechny úrovně stromu (kromě kořenové a poslední). Nasledně se inicializuje seznam hlaviček jednosměrně vázaných seznamů listových dat, které budou sestaveny v následujícím kroku. Tento seznam má velikost 8^h položek, kde h je hloubka stromu. Každá položka musí být nastavena na neplatnou hodnotu.

Poté se pro každý koncový voxel vypočte Mortonovo kódování jeho pozice. Jedná se o pouhé prolínání bitů všech tří koordinát do jednoho bloku 4 bytů. Tím se získá lineární index voxelu, pro který platí, že dva voxely nacházející se ve vzájemné blízkosti budou mít také blízké hodnoty jejich lineárních indexů. Tento index se použije k přístupu do seznamu hlaviček jednosměrně vázaných seznamů listových dat. Na daném místě v seznamu zapíše svůj mortonovský lineární index. Zároveň načte hodnotu která byla v seznamu uložena před zápisem. To se učiní atomickou operací `atomicExchange()`. K danému místu v seznamu bude přistupovat více vláken, které zpracovávají voxel se stejným lineárním indexem. Tím, že se více vláken pokusí vyměnit hodnotu v seznamu atomickou operací, dojde k jejich zřetězení, jak je ukázáno na obrázku 4.3. Např. první vlákno, které vymění hodnotu bude

posledním prvkem jednosměrně vázaného seznamu, protože načte koncový neplatný odkaz. Obdobně poslední vlákno načte odkaz na druhý prvek lineárního seznamu a do seznamu hlaviček vloží odkaz sám na sebe (jakožto na první prvek).



Obrázek 4.3: Princip využití atomické výměny hodnoty při propojování jednosměrně vázaného seznamu

Struktura uzlů stromu. Struktura uzlů oktalového stromu odpovídá tomu, jak je naznačeno na obrázku 4.4. Uzly jsou identifikovány svou pozicí na dané úrovni stromu. Informace o tom, na které úrovni stromu se daný uzel nachází, se neuchovává, protože je možné ji odvodit během průchodu stromu. Uchovává se však odkaz na prvního potomka. Potomci jsou vždy alokováni po osmici v souvislém bloku v bufferu uzlů stromu. Poslední položkou je příznak nalezení následovníka v poli dříve vygenerovaných voxelů. Obdobně jako u struktury voxelů je i zde použito zarovnání struktury na násobek 16 bytů.

Obrázek 4.4: Bytová reprezentace struktury uzlů oktalového stromu. Od nejvýznamějšího bytu: vektor pozice oktantu na dané úrovni stromu, odkaz na prvního z 8 potomků a příznak pro určení dělení uzlu.

Jelikož je algoritmus stavby struktury stromu ve své podstatě iterativní, je vykonáván v po sobě následujících volání programu compute shaderu. Avšak nadbytečná režie přesunu jednotlivých výsledků iterací zpět na CPU, aktualizace konfigurace uniformních proměnných a opětovné volání compute shaderu může způsobit prodlevy, které vedou k nedostatečnému využití a zatížení prostředků grafického procesoru. Z tohoto důvodu je režie algoritmu řešena pomocí nepřímých volání `glDispatchIndirect()`, které tento problém zcela potlačují. Compute shader si tak sám vytváří parametry pro jeho příští volání ve

vyčleněném bufferu. Algoritmus je složen z předem daného počtu iterací, z čehož je možné odvodit počet potřebných nepřímých volání z CPU.

4.5 Ukázková aplikace

K předvedení funkcionality navržené knihovny slouží ukázková aplikace *demoApp*, která je vytvořena za použití frameworku *Qt* pro tvorbu uživatelského grafického prostředí. Aplikace je jednoduchým nástrojem pro zobrazování modelů. Pro ně vytváří akcelerační strukturu, která zajišťuje rychlé vyhledání trojúhelníku zasaženého paprskem, který je do scény vržen při kliknutí myši směrem pohledu. Tak umožňuje uživateli vybrat a zvýraznit požadovaný trojúhelník ve scéně. Vzhled aplikace je ukázán na obrázku 4.5.

Obrázek 4.5: Snímek obrazovky okna ukázkové aplikace

Kapitola 5

Měření

V této kapitole jsou popsána data a postupy, které byly použity pro měření efektivity navrhovaného řešení. Zároveň jsou zde vysvětleny možné příčiny časových prodlev a zpomalení.

5.1 Testovací scény

Cílem měření bylo zjistit, jaký vliv má na rychlost výstavby akcelerační struktury počet a rozmístění trojúhelníků ve scéně. Testování výkonnosti budování oktalového stromu bylo prováděno na třech testovacích scénách. Jedná se o scény: Stanford Happy Buddha, Stanford Dragon a Crytek Sponza. Jejich podoba je ukázána na obrázku 5.1.Všechny tyto scény jsou běžně užívané k testování výkonnosti vykreslovacích algoritmů a v hojné míře i metod pro urychlování s nimi spojených výpočtů, jako jsou i akcelerační struktury. Jednotlivé scény byly zdecimovány na následující velikosti: Stanford Buddha – 3 538 trojúhelníků; Stanford Dragon – 2 243 trojúhelníkú; Crytek Sponza – 26 220 trojúhelníků.

Stanfordský budha je trojrozměrným modelem malé sošky obtloustlé lidské postavy. Jako takový má tento model topologii trojúhelníkové sítě koncentrovanou ke stěnám modelové obálky. To znamená, že vygenerovaný oktalový strom pro scénu tvořenou tímto modelem bude mít listové uzly situované v nejvyšší míře právě u krajů obalovacího tělesa scény a směrem ke středu modelu bude na nejnižších úrovních stromu dutý. Tvar modelu je také zajímavý tím, že obsahuje řadu děr, které jsou vhodné pro testování problému "průletu uchem jehly", který se zejména neblaze projevuje pro paprskové vykreslovací metody.

Stanford Dragon na rozdíl od stanfordského budhy má tvar esovitě zakrouceného hada, který vyplňuje prostor jeho obalovacího tělesa. Povrch modelu tak více zasahuje do středu prostoru jeho obálky. Výsledný oktalový strom pro tento tvar tak při podobném počtu trojúhelníků jako u modelu budhy obsahuje větší počet listových uzlů.

Crytek Sponza je třetí zvolenou scénou pro měření rychlosti stavby oktalového stromu. Oproti předchozím dvěma modelům se zásadně liší. Opět představuje problém "průletu uchem jehly", tentokrát však spíše pro případy v dálce. Model totiž ztvárňuje nádvoří palácové stavby s arkádami a sloupořadím. Vytvořený oktalový strom tak pro tuto scénu bude mít listové uzly rovnoměrně rozmístěny po celé scéně.

Obrázek 5.1: Scény použité při měření. Zprava: veselý budha, stanfordský drak a palácové nádvoří

5.2 Postup a výsledky měření

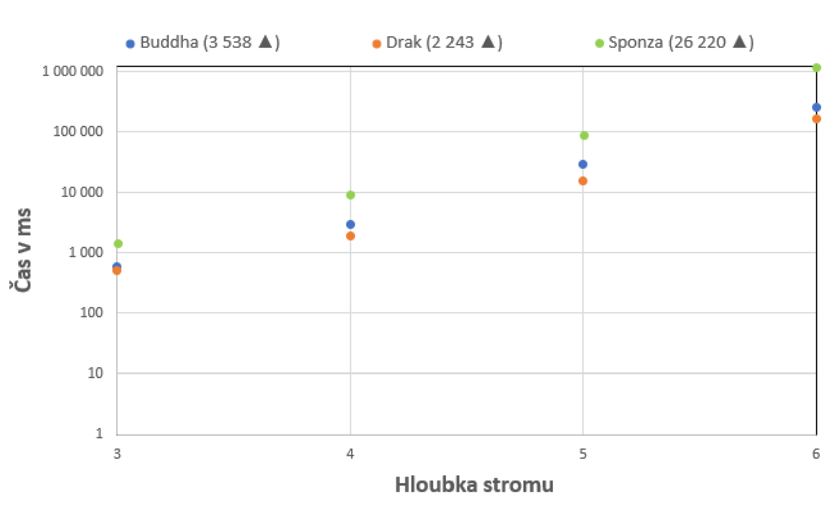
Měření doby vykonávání metody pro generování akcelerační struktury bylo provedeno za použití časových čítačů grafického procesoru využitím příkazů OpenGL a jimi zpřístupněného mechanismu tzv. dotazů (angl. querries). Ty umožňují vložit požadavek na načtení hodnoty interního stavu OpenGL přímo do fronty příkazů. Tím je pak možné přesně změřit dobu vykonávání jednotlivých bloků příkazů.

Měření bylo prováděno na třech scénách uvedených v předchozí podkapitole. Pro všechny tyto scény byly sestaveny oktalové stromu o hloubce 3, 4, 5 a 6. Měření byla provedena opakovaně a z aritmetických průměrů naměřených údajů byla vytvořena tabulka 5.1, která ukazuje vliv počtu trojúhelníků a topologie scény na rychlost výstavby akcelerační struktury. Všechny scény byly měřeny na GPU NVIDIA GeForce GTX960 architektury Maxwell.

Scéna	Počet trojúhelníků	Hloubka stromu	Čas voxelizace (ms)	Čas stavby struktury stromu (ms)	Čas přenosu dat (ms)	Celkový čas (ms)
Stanford Happy Buddha	3 538	3	0,7	370	20	592
		4	0,5	2 695	36	2 921
		5	0,5	29 970	72	30 266
		6	1,1	255 150	77	255 455
Stanford Dragon	2 243	3	0,3	245	11	512
		4	0,4	1 629	23	1 909
		5	0,7	15 512	37	15 811
		6	0,7	162 059	57	163 345
Crytek Sponza	26 220	3	2,3	1 132	86	1 481
		4	2,6	8 964	118	9 372
		5	2,5	88 053	218	88 597
		6	3,7	1 181 685	239	1 182 155

Tabulka 5.1: Naměřené časové hodnoty jednotlivých fází výpočtu pro různé scény

Jak je na první pohled z grafu na obrázku 5.2 patrné, s rostoucí hloubkou stromu a zvyšujícím se počtem trojúhelníků ve scéně prudce roste i čas potřebný k vybudování oktalového stromu. I pro relativně jednoduché scény s pouhými tisícovkami grafických primitiv se nepodařilo sestavit akcelerační strukturu v čase, který by vyžadovaly aplikace v reálném čase, nebo alespoň interaktivní aplikace.



Obrázek 5.2: Graf zobrazující závislost počtu trojúhelníků ve scéně a hloubky generované struktury na čase potřebném k vybudování oktalového stromu.

Dostatečně efektivní se zdá být fáze voxelizace scény, která se pro měřené případy vždy pohybuje v řádu milisekund. Problémem je ale fáze stavby struktury stromu, kdy dochází ke značnému zpomalení v důsledku nutnosti ověřit velký počet kombinací dvojic uzlů stromu na rozdělované úrovni a buněk voxelové mřížky scény. Ve fázi přenosu dat z grafického procesoru do hlavní paměti dochází k dalším časovým prodlevám, které se v porovnání s těmi v předchozí fázi nezdají být příliš významné, ale pro větší hloubky stromu již samy překračují práh obnovovací frekvence aplikací v reálném čase.

Kapitola 6

Závěr

Tato práce se zabývá problematikou dlouhých inicializačních časů běžně užívaných akceleračních struktur a představuje možný způsob budování oktalových stromů pro trojúhelníkové scény za využití výpočetního potenciálu grafického hardware.

V rámci práce vznikla knihovna poskytující reprezentaci zvolené akcelerační struktury, která pro práci s ní poskytuje základní operace. Vytvořená knihovna byla implementována v ukázkové aplikaci, která demonstrovala její možné využití.

Za úspěch zvoleného přístupu lze považovat využití voxelizace pro nahrazení detekce kolize buněk oktalového stromu při jeho stavbě. Její vykonání trvá dostatečně krátkou dobu na to, aby bylo možné provést i další operace v časovém intervalu aplikací v reálném čase.

Výsledná implementace ale trpí vážným dopadem nedostatečného využití prostředků grafického procesoru nevyváženým rozvržením pracovních úkonů a přítomností závislostí mezi jednotlivými kroky.

Pokračování této práce by se na výše zmiňovaný problém mělo zaměřit. Například použití představené metody k budování SVO by mělo vést ke snížení počtu generovaných uzlů stromu. To by odlehčilo celkovou výpočetní zátěž. Případnou úpravou by mohlo být vynechání fáze tvorby stromové struktury a proces by tak pouze tvořil uniformní mřížku jako akcelerační strukturu.

Další možností vhodnou k prozkoumání by mohla být implementace algoritmu stavby stromu zdola nahoru. Ta však vyžaduje seřazení buněk voxelové mřížky scény podle jejich umístění v prostoru, což by mohlo způsobit podobné výkonnostní problémy.

Literatura

[1] Agate, M., Grimsdale, R. L. a Lister, P. F. The HERO algorithm for ray-tracing octrees. In: *Advances in Computer Graphics Hardware IV*. Springer, 1991, s. 61–73.

[2] Amanatides, J., Woo, A. et al. A fast voxel traversal algorithm for ray tracing. In: *Eurographics*. 1987, sv. 87, č. 3, s. 3–10.

[3] Bittner, J. a Havran, V. RDH: ray distribution heuristics for construction of spatial data structures. In: *Proceedings of the 25th Spring Conference on Computer Graphics*. 2009, s. 51–58.

[4] Crassin, C. *GigaVoxels: A voxel-based rendering pipeline for efficient exploration of large and detailed scenes*. 2011. Disertační práce. Université de Grenoble.

[5] Crassin, C. a Green, S. Octree-based sparse voxelization using the GPU hardware rasterizer. *OpenGL Insights*. CRC Press. 2012, s. 303–318.

[6] Cui, J., Chow, Y.-W. a Zhang, M. A voxel-based octree construction approach for procedural cave generation. 2011.

[7] Dammertz, H., Hanika, J. a Keller, A. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In: Wiley Online Library. *Computer Graphics Forum*. 2008, sv. 27, č. 4, s. 1225–1233.

[8] Espe, A. E. *Real-Time Ray Tracing of Animated Sparse Voxel Octrees on FPGA*. 2019. Diplomová práce. NTNU.

[9] Glassner, A. S. Space subdivision for fast ray tracing. *IEEE Computer Graphics and applications*. IEEE. 1984, sv. 4, č. 10, s. 15–24.

[10] Hapala, M., Karlík, O. a Havran, V. When it makes sense to use uniform grids for ray tracing. Václav Skala-UNION Agency. 2011.

[11] Hasselgren, J. *Efficient Compression and Rasterization Algorithms for Graphics Hardware*. 2006. Disertační práce. Department of Computer Science, Lund Institute of Technology, Lund University.

[12] Havran, V. *Heuristic ray shooting algorithms*. 2000. Disertační práce. Faculty of Electrical Engineering, Czech Technical University, Prague.

[13] Havran, V. About the relation between spatial subdivisions and object hierarchies used in ray tracing. In: *Proceedings of the 23rd Spring Conference on Computer Graphics*. 2007, s. 43–48.

[14] Ize, T., Wald, I. a Parker, S. G. Ray tracing with the BSP tree. In: IEEE. *2008 IEEE Symposium on Interactive Ray Tracing.* 2008, s. 159–166.

[15] Laine, S. a Karras, T. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics.* IEEE. 2010, sv. 17, č. 8, s. 1048–1059.

[16] MacDonald, D. *Space subdivision algorithms for ray tracing.* 1991. Disertační práce. University of Waterloo, Computer Science Department.

[17] Meagher, D. Geometric modeling using octree encoding. *Computer graphics and image processing.* Elsevier. 1982, sv. 19, č. 2, s. 129–147.

[18] Möller, T. a Trumbore, B. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools.* Taylor & Francis. 1997, sv. 2, č. 1, s. 21–28.

[19] Pantaleoni, J. VoxelPipe: A programmable pipeline for 3D voxelization. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics.* 2011, s. 99–106.

[20] Qin, Y., Lin, J. a Huang, X. Massively parallel ray tracing algorithm using GPU. In: IEEE. *2015 Science and Information Conference (SAI).* 2015, s. 699–703.

[21] Revelles, J., Urena, C. a Lastra, M. An efficient parametric algorithm for octree traversal. Václav Skala-UNION Agency. 2000.

[22] Samet, H. Implementing ray tracing with octrees and neighbor finding. *Computers & Graphics.* Elsevier. 1989, sv. 13, č. 4, s. 445–460.

[23] Schmittler, J., Woop, S., Wagner, D., Paul, W. J. a Slusallek, P. Realtime ray tracing of dynamic scenes on an FPGA chip. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware.* 2004, s. 95–106.

[24] Stibora, R. *Building of SBVH on Graphical Hardware.* 2016. Diplomová práce. Faculty of Informatics, Masaryk University.

[25] Thrane, N., Simonsen, L. O. et al. A comparison of acceleration structures for GPU assisted ray tracing. Citeseer. 2005.

[26] Wächter, C. a Keller, A. Instant ray tracing: The bounding interval hierarchy. *Rendering Techniques.* 2006, sv. 2006, s. 139–149.

[27] Wald, I. a Havran, V. On building fast kd-trees for ray tracing, and on doing that in O (N log N). In: IEEE. *2006 IEEE Symposium on Interactive Ray Tracing.* 2006, s. 61–69.

[28] Weghorst, H., Hooper, G. a Greenberg, D. P. Improved computational methods for ray tracing. *ACM Transactions on Graphics (TOG).* ACM New York, NY, USA. 1984, sv. 3, č. 1, s. 52–69.

[29] Williams, A., Barrus, S., Morley, R. K. a Shirley, P. An efficient and robust ray-box intersection algorithm. In: *ACM SIGGRAPH 2005 Courses.* 2005, s. 9–es.

[30] Yang, X., Xu, D.-q. a Zhao, L. A fast SAH-based construction of Octree. Václav Skala-UNION Agency. 2009.

Příloha A

Obsah přiloženého paměťového média