DPG semestrální práce - zadání číslo 5
Ray Tracing BVH constructed via SAH approximation

Bc. Patrik Schiller
Katedra počítačové grafiky a interakce,
Fakulta elektrotechnická, ČVUT Praha,
Prosinec 2020

Abstract

Práce se zaměřuje na implementaci výstavby datové struktury zvané *Bounding Volume Hierarchy*[1](hierarchie obálek), pomocí techniky původně vyvinuté pro kd-stromy. Stavba je urychlena použitím cenové funkce *SAH* (`Surface Area Heuristic`) a metody zvané *binning* (dělení do přihrádek). Při výstavbě dochází k velmi rychlému uniformnímu dělení primitiv do přihrádek, mezi kterými je s pomocí funkce SAH jednoduše vybrána nejlepší dělící rovina. Tímto způsobem je prostor efektivně dělen na dvě poloviny v každém kroku výstavby stromu BVH. Implementace byla navíc doplněna `SSE` instrukcemi umožňujícími paralelní zpracování vektorových dat v jedné instrukci, čímž dochází k urychlení některých výpočtů. Řešení bylo doplněno měřením efektivity struktury na deseti scénách s různým počtem primitiv a jejich distribucí v prostoru. Měření se soustředilo především na čas výstavby a čas traverzace, popřípadě čas syntézy obrázků bez stínů i se stíny.

Keywords: Ray Tracing, Ray Casting, BVH, binned BVH, SAH, binning

1. Úvod

Metoda sledování paprsku (*Ray Tracing*) je postupem času stále populárnějším způsobem, jak renderovat[2] kvalitní obrázky, které vychází ze zákonů šíření světla prostorem. Díky tomu jsou snímky poměrně realistické a poutavější, než snímky získané například pomocí metody rasterizace. Bohužel velkou nevýhodou metody sledování paprsku je její rychlost, která drasticky klesá se zvyšujícím se počtem primitiv[3] či světelných zdrojů ve scéně. Vzhledem k čím dál vyšším nárokům na rychlost (včetně real-time renderování) je nutné při syntéze obrazu pomocí metod ray tracingu použít akceleračních struktur, které výrazně snižují počet incidentních operací (výpočty průsečíku paprsku s objektem). Mezi nejpoužívanější struktury patří *kd-stromy* nebo hierarchie obálek (*Bounding Volume Hierarchy*).

Motivací pro použití právě zmíněné hierarchie obálek je zájem o metody sledování paprsků a možnosti jejich urychlení. Navíc práce se stromovou strukturou je určitě výhodou do budoucna vzhledem k tomu, že vyhledávací stromy jsou v IT velmi důležité. Jejich použití v počítačové grafice je o to zajímavější, že je výsledek možné pozorovat v podobě hezkého a rychle vykresleného snímku.

[1]Zkráceně BVH
[2]Synonymem renderování je syntéza obrazu
[3]Primitivum - elementární geometrický objekt (trojúhelník, koule..)

Cílem této práce je aplikovat techniku vyvinutou pro výstavbu kd-stromu na výstavbu hierarchie obálek (BVH). Tento problém je poměrně detailně rozepsán v článku [1] a proto z něj tato semestrální práce vychází. Pro větší pohodlí je implementace začleněna do již existujícího renderovacího frameworku `nanoGolem`[4], který poskytuje pan *Vlastimil Havran*[5] pro potřeby předmětu `DPG`. Díky tomu je potřebná implementace omezena především na algoritmus výstavby a drobné úpravy v pár dalších souborech.

2. Popis algoritmu

Algoritmus implementovaný v této práci vychází z poznatků při zkoumání kd-stromů. Vzhledem k tomu, že hierarchie obálek a kd-stromy jsou si celkem podobné (pouze jinak pracují se svým obsahem), je možné tyto poznatky s menšími úpravami aplikovat i na hierarchie obálek (BVH). Právě touto skutečností se zabývá zmíněný článek[1], na kterém staví implementace této práce. Vzhledem k tomu, že BVH dělí prostor na základě pouze jednoho bodu (těžiště) oproti třem bodům u kd-stromu (tři vrcholy trojúhelníku), je dělení primitiv jednodušší a rychlejší, a to především díky tomu, že samotný bod se vždy nachází pouze na jedné straně hranice (oproti celému trojúhelníku). Detailnějšímu rozboru BVH se věnuje následující podkapitola.

2.1. Hierarchie obálek - BVH

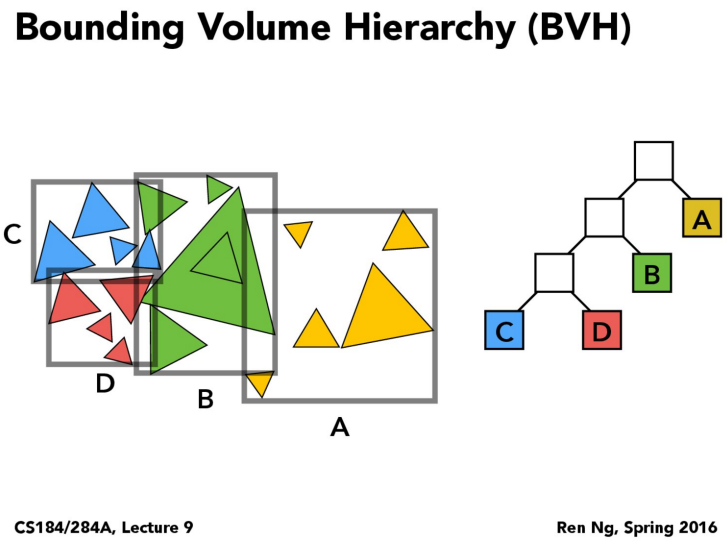
Hierarchie obálek je rekurzivní dělení prostoru realizované pomocí binárního stromu, které pracuje s obálkami objektů a jejich množin. Obálky jsou reprezentovány jednoduchými prostorovými objekty, pro které existuje jednoduché parametrické vyjádření průsečíku s paprskem. Mezi takové patří například koule, kvádr nebo osově zarovnaná obálka známá jako *AABB*[6]. Právě AABB je použito i v této práci.

V každém kroku rekurzivní výstavby BVH (dělení uzlu stromu na dva potomky) je výchozí (pod)prostor rozdělen na dvě poloviny, které jednoznačně vymezují dvě vzniklé podmnožiny primitiv. Primitiva jsou jednoznačně rozdělena na základě souřadnic jejich těžišť (buď těžiště primitiva nebo jeho AABB). To je rozdílné oproti metodám kd-stromů, kde v úvahu připadají například všechny tři vrcholy trojúhelníku. Díky tomu nemůže v BVH nastat případ, kdy primitivum náleží na obě strany dělící roviny (dělící rovina vede skrze primitivum).

Existuje několik přístupů, na základě kterých prvky scény dělit. Mezi základní patří *prostorový medián* nebo různé cenové modely. Mezi nejpoužívanější patří cenová funkce *SAH* (`Surface Area Heuristic`), která oproti prostorovému mediánu bere v potaz počet primitiv na obou stranách dělící hranice spolu s obsahem povrchu jejich obálky (AABB).

Výše zmíněné rozhodování na základě těžišť nijak nepopírá skutečnost, že primitiva mohou (částečně) náležet do více obálek (respektive jejich obálky se překrývají). Toto je zohledněno v případě výpočtů SAH, samotné dělení tím ovlivněno není.

[4]Framework pro testování různých algoritmů a datových struktur pro syntézu obrazu
[5]havran_AT_fel.cvut.cz
[6]Axis Aligned Bounding Box



Obrázek 1: Ukázka hierarchie obálek - vztah prostoru (nalevo) k BVH stromu (napravo) [4]

2.2. Výstavba BVH s pomocí přihrádek

Je dobré zmínit, že způsobů výstavby BVH je více - od shora dolů dělením (*top-down*, `partitioning`), od spoda nahoru shlukováním (*bottom-up*, `clustering`), nebo postupným zatřizováním. V této práci je využita metoda top-down, vzniklá struktura se řadí mezi *TD-BVH*[7].

Samotná výstavba je rozdělena na dvě fáze: `přípravná fáze` (2.2.1) a `fáze dělení` (2.2.2).

2.2.1. Přípravná fáze

Vzhledem k tomu, že je hlavním cílem co největší úspora času, tak je dobré si data, u kterých to má smysl, předzpracovat. Tím se dá vyhnout zbytečným opakujícím se výpočtům v rekurzivním dělení, což by výstavbu zbytečně zpomalovalo. Hned na začátku je možné předalokovat pole s primitivy a jejich obálkami. Dále je možné předpočítat těžiště všech primitiv a také je uložit do pole. Tím vzniknou tři stejně velká pole, kde každý index obsahuje potřebné informace o daném primitivu. Dobré je také inicializovat pole s předem stanoveným počtem přihrádek. Každá přihrádka obsahuje počítadlo primitiv, která obsahuje, a jejich celkovou obálku (v článku značeno jako *bin-bounds*). Z článku [1] vyplývá, že nejvýhodnější počet přihrádek je 16. Je možné použít i menší počet, naopak více jak 16 již nemá téměř žádné výhody.

[7] Top-Down Bounding Volume Hierarchy

2.2.2. Fáze dělení

Po inicializaci následuje rekurzivní fáze dělení scény do stromu BVH. Nejprve dojde k rozdělení vstupní množiny primitiv do předpřipravených přihrádek (*binning*). Dělení je provedeno dle souřadnic v ose, ve které je obálka vstupní množiny primitiv největší[8]. Primitiva jsou rozdělena dle souřadnice jejich těžiště, každá přihrádka pokrývá stejně velkou část intervalu reprezentujícího dělící osu. Pro `K` přihrádek je osa rozdělena na K intervalů o velikosti $1/K$. Projekce primitiv do přihrádek (binning) se řídí následujícím vzorcem (rovnice 1), který upravuje původní vzorec tak, aby používal předpočítané konstanty k_0 a k_1 (rovnice 2).

$$binID_i = k_1(c_{i,k} - k_0) \ . \tag{1}$$

Uvedená rovnice uvádí vztah pro výpočet ID přihrádky, do které má být umístěno $i-t$ primitivum. Proměnná $c_{i,k}$ je jediná měnící se hodnota, která reprezentuje souřadnici těžiště i-tého primitiva v ose k, tedy v ose, ve které je obálka primitiv největší. Konstanty k_0 a k_1 jsou předpočítány vždy na začátku fáze dělení podle následujících vztahů.

$$k_0 = cb_{min,k} \tag{2}$$

$$k_1 = \frac{K \cdot (1-\epsilon)}{cb_{max,k} - cb_{min,k}} \tag{3}$$

Každé těžiště je počítáno ve vztahu k minimální ($cb_{min,k}$) a maximální hranici ($cb_{max,k}$) těžišť množiny primitiv, se kterou pracuje aktuální úroveň rekurze. Tyto hranice je nutné před dalším rekurzivním zanořením přepočítat spolu s obálkou pro každou nově vzniklou podmnožinu primitiv - tedy polovinu dělení. Epsilon (ϵ) zde zastupuje odsazení od pravé hranice pro případ, že by některé těžiště náleželo do pravé hranice poslední přihrádky (a tedy by nešlo vypočítat validní ID přihrádky). K pak reprezentuje celkový počet přihrádek.

Po rozdělení primitiv do přihrádek je možné určit nejvýhodnější dělení. Díky rozdělení do přihrádek existuje pouze *K-1* možností, jak primitiva rozdělit. Pro každou z těchto možností je potřeba vypočítat cenovou funkci SAH podle následujícího vzorce, kde $Cost_j$ (rovnice 4) značí cenu pro j-tou dělící rovinu.

$$Cost_j = A_{L,j} \cdot N_{L,j} + A_{R,j} \cdot N_{R,j} \ . \tag{4}$$

Pro výpočet je potřeba znát počet primitiv na levé a pravé straně hranice ($N_{L,j}$ a $N_{R,j}$) a zároveň obsah povrchu obálky primitiv na pravé a levé straně ($A_{L,j}$ a $A_{R,j}$). Tyto hodnoty získáme součtem primitiv a sjednocením obálek na obou stranách hranice.

Jakmile je nalezena nejlepší dělící rovina, je potřeba okolo ní přeuspořádat primitiva v předalokovaném poli (s návazností na zbylá pole s těžišti a obálkami). Vzhledem k tomu, že každá úroveň rekurze obsahuje ukazatel na první a poslední prvek, který patří do právě zpracovávané podmnožiny, je možné v tomto intervalu také primitiva přeuspořádat podle výsledků výpočtu dělící roviny, a to takzvaně *in-place* [9]. Jinak řečeno, každý výpočet dělení v daném uzlu pracuje nad intervalem dat v rozsahu *<begin, end)*. Vzhledem k tomu, že tento postup je velmi podobný s jedním krokem *Quicksort* řadícího algoritmu (konkrétně `partitioning` - dělení dle vybraného *pivota*), je možné toto řešení přepoužít. Jako pivot je zvoleno

[8] ...ve vzorcích značeno jako malé K

[9] ...in-place znamená, že nepotřebujeme žádné pomocné pole či jinou DS

ID nejlevnější přihrádky (a tedy i dělící roviny), a jako dělící podmínka slouží opětovný výpočet ID přihrádky. Podle podmínek `větší než` a `menší než` jsou primitiva prohazována okolo pivota. Výsledkem tohoto procesu jsou přeuspořádaná primitiva a iterátor *mid* ukazující na prostřední prvek ve zpracovávaném intervalu *begin* a *end*, který data rozdělí na dva podintervaly *<begin, mid)* a *<mid, end)*.

Na základě výpočtu pivota a prohození primitiv je možné spočítat hranice těžišť c_b (centroid bounds) a obálky obou vzniklých podmnožin. Následně je opět rekurzivně volána dělící fáze na obou vzniklých podmnožinách - tedy celý postup se opakuje pro obě nově vzniklé podmnožiny. Výsledkem rekurzivního volání je vytvořený uzel BVH stromu. V případě vyhodnocení rekurzivního volání na obou podmnožinách primitiv daného uzlu jsou vráceny dva uzly (vnitřní nebo listy), levý a pravý, které jsou spojeny pomocí nového vnitřního uzlu. Nový vnitřní uzel je následně vrácen o úroveň výš v rekurzi.

2.2.3. Ukončení algoritmu

Rekurzivní sestup algoritmu je ukončen jedním ze tří případů. Prvním je hraniční počet primitiv ve vzniklých podmnožinách - v této práci je dělení prováděno až do počtu dvou primitiv, kdy jsou vytvořeny dva listy a spojeny jedním vnitřním uzlem. Druhým případem je příliš malý rozsah hranic těžišť, kdy další dělení nemá smysl. Třetí případ nastane, když předpokládaná cena výpočtu SAH je větší, než cena vytvoření listu s více primitivy (toto nebylo v této práci uvažováno).

2.2.4. Využití SSE

Vzhledem k tomu, že v článku [1] bylo doporučováno využití *SIMD instrukcí*, došlo k jejich zohlednění i v této práci. SIMD instrukce (`Single Instruction Multiple Data`) umožňují provést danou výpočetní operaci na množině více dat najednou. Množinou je zde myšlen vektor s pevnou velikostí, například se čtyřmi nebo osmi prvky. Jinak řečeno, pomocí SIMD instrukce je možné například sečíst 8 čísel v jedné instrukci. Vzhledem k tomu, že velká část výpočtů v této práci se týká vektorů ve 3D prostoru, je možné výpočty akcelerovat právě pomocí SIMD. Pro implementaci byla vybrána skupina instrukcí *SSE* (oproti AVX) pracující se 128-mi bitovými (16-ti bytovými) vektory, obsahujícími 4 složky po 4 bytech. První tři složky reprezentují souřadnice x, y a z, poslední složka slouží jako bitové zarovnání. Jako SIMD datový typ byl použit $\_m128$ pracující s datovým typem `Float`.

SIMD instrukce byly využity pro výpočty obálek těžišť primitiv pomocí funkce *minimizeSIMD()* a *maximizeSIMD()* využívající instrukci *_mm_max_ps* [2]. Podobným způsobem byla implementována funkce *includeSIMD()* pro sjednocení obálek (respektive kumulování obálky scény obálkami primitiv). O něco složitější je poslední metoda pro výpočet samotných těžišť *computeCentroidSIMD()*, která pracuje s instrukcemi *_mm_add_ps* a *_mm_mul_ps* [2] a s vektorem konstant $\_mm\_set1\_ps(0.5f)$ potřebným pro násobení konstantou při výpočtu těžišť. Pro potřeby výpočtů bylo lehce upraveno rozhraní třídy `CVector3D`, kde byly souřadnice vystaveny pod SIMD proměnnou *_16_byte_vec* pomocí deklarace *Union*.

3. Detaily implementace, problémy

Tato práce byla poměrně náročná, jedna z nejtěžších během studia. Hned na začátku bylo potřeba pochopit anglicky psaný vědecký článek, což bez opakovaného pročítání a vypisování poznámek nebylo možné. Pouze příprava na první prezentaci zabrala okolo deseti hodin. Další časově náročnou záležitostí byla potřeba se zorientovat ve frameworku nanoGolem, pochopit vazby jednotlivých částí a zjistit, jak do projektu implementovat novou traverzační strukturu. Výhodou použití frameworku je, že nebylo potřeba implementovat samotný renderer, což určitě nějaký čas ušetřilo. Na druhou stranu neznalost frameworku nanoGolem stěžuje orientaci a zpomaluje implementaci zadaného tématu.

Implementace semestrální práce vycházela z existující třídy `TDBVH`, která realizuje výstavbu a traverzaci hierarchie obálek s pomocí prostorového mediánu. Součásti, které bylo možné přepoužít, byly zachovány, implementace se týkala především výstavby BVH s pomocí SAH a přihrádkování. Traverzační funkce *findNearestI()* zůstala beze změny. Tím byla snížena pravděpodobnost zavlečení chyby do samotného procesu syntézy obrazu.

Nejvíce času zabralo hledání chyby ve funkci *partitionSplit()* spojené s chybným dělením primitiv a vytvářením obálek v druhé části metody *buildRecursively()*. Výpočty ID přihrádek vracely nesmyslná čísla, dělení primitiv často končilo zacyklením. Kompletní vyřešení této chyby zabralo okolo 20-ti hodin.

Mezi menší problémy se řadí doplnění projektu o SIMD instrukce, které byly pro mě velká neznámá (za jejich úvodní pochopení vděčím YouTube kanálu `javidx9`[3]), a doimplementování stínových paprsků do projektu frameworku nanoGolem. Při započítání času stráveného nad psaním reportu, měřením a přípravou projektu k odevzdání odhaduji čas strávený nad touto prací okolo 100 hodin.

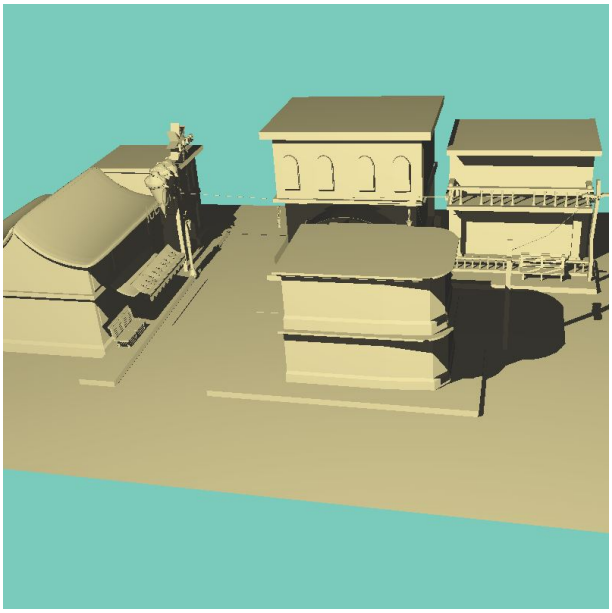
4. Naměřené výsledky

Tato kapitola se věnuje měření implementované datové struktury dle [1] nad testovacími daty. Jako testovací data posloužilo deset vzorových scén s různým počtem primitiv a jejich různorodou distribucí v prostoru scény. Tím mělo dojít k otestovaní většiny krajních podmínek, které by mohly negativně ovlivnit výsledný tvar a hloubku BVH.

Měření bylo provedeno nejprve bez stínových paprsků, zaměřené na počet listů a uzlů v BVH, průměrnou hodnotu SAH a čas výstavby BVH. Spolu s výstavbou byla měřena kvalita vzniklé BVH na základě času syntézy obrazu (*render time*), průměrném počtu incidentních operací s primitivy, průměrném počtu traverzačních kroků a průměrném času traverzace jednoho paprsku. Měření bylo provedeno s 16-ti přihrádkami a s počtem jedno primitivum na jeden list (listů má být stejný počet, jako primitiv). V naměřených datech se bohužel nachází chyba, kde se počet primitiv a listů liší o jedna, protože byl v kódu špatně nastavený rozsah iterace.

U syntézy se stínovými paprsky se měřil čas syntézy obrazu (bez výstavby) a rozdíly v počtech incidentních operací a průměrném času traverzace jednoho paprsku. Více v tabulce 1. Během syntézy obrazu se stínovými paprsky bylo potřeba odladit správnou odchylku odsazení počátku paprsku od plochy objektů, aby v obrázku nevznikaly artefakty (*shadow acne*).

V tabulce 1 může být zarážející průměrná hodnota SAH, která je v některých případech opravdu vysoká. To je zapříčiněno použitím různých měřítek u různých scén. Tento problém nijak neomezuje funkčnost algoritmu avšak je možné ho jednoduše odstranit přidáním dělení obou stran (L a R) v rovnici 4 plochou obálky celé scény.



Scéna city_B + stíny.

Scéna conference_B + stíny.

4.1. Testovací sestava

Testovaní bylo z časových (a vládních) důvodů provedeno na vlastním počítači. Testovací sestava se skládá z: Procesor AMD Ryzen 7 3700x s frekvencí 3,6 GHz, s osmi jádry a šestnácti vlákny, vyrovnávací paměť L3 32768 kB, L2 4096 kB. Procesor byl doplněn o 16 GB DDR4 pamětí RAM, dva moduly po 8 GB s frekvencí 3200 MHz. Jako operační systém byl použit Windows 10 Pro 64bit. K implementaci, kompilaci a testování bylo použito vývojové prostředí Visual Studio 2019, v módu *Release* s použitím jednoho procesorového vlákna.

4.2. Výsledky měření

Naměřená data byla zanesena do následující tabulky 1. Každé scéně jsou věnovány dva řádky, kdy první obsahuje měření bez stínových paprsků, druhý řádek obsahuje naměřená data ze syntézy obrazu se stínovými paprsky. V globálním nastavení `global.env` byl počet vzorků `SetRayTracing::cntSamples` nastaven na 10 a limit hloubky `SetRayTracing::depthLimit` na 1.

5. Závěr

Cílem semestrální práce bylo implementovat specifickou metodu výstavby hierarchie obálek (BVH) s pomocí cenové funkce SAH a metody přihrádkování dle článku [1]. Tento způsob výstavby má být rychlejší než klasická výstavba s pomocí prostorového mediánu, přitom ale má mít výsledná struktura stejnou kvalitu.

Výstupem práce je třída `Schiller_BVH` (soubory *Binned_BVH.h* a *Binned_BVH.cpp*) reprezentující implementaci hierarchie obálek s pomocí metody přihrádkování a cenové funkce SAH. Implementace je doplněna úpravami potřebných tříd a souborů ve frameworku nanoGolem - především `CVector3D`, `CRT_CoreWithCamera_App` a `CRT_00_App`.

Scene	#Prims	#Leaves	#Nodes [*total*]	SAH [*Average*]	T_B [*ms*]	T_R [μs]	N_{IT} [*ray*]	N_{TR} [*ray*]	N_Q [*per ray*]	Render time [*ms*]
A10	218652	218651	434301	2273040	497	0.14	0.77	5.7	640000	140
+ shadows						0.2	1	8.5	792013	212
Armadillo	345944	345943	691885	$2,64 * 10^{11}$	734	0.29	0.93	9.8	640000	240
+ shadows						0.35	0.95	13	933099	400
Asian Dragon	7219045	7219044	14438087	313953	15920	0.56	1.2	12	640000	420
+ shadows						0.57	1.1	14	785422	520
City	68497	68496	136991	$2,76 * 10^{9}$	136	0.28	2.3	14	640000	250
+ shadows						0.29	1.6	15	1338779	520
City2	75420	75419	150837	$2,66 * 10^{7}$	120	0.27	1.6	11	640000	240
+ shadows						0.33	1.8	14	1011377	430
Conference	282755	282754	565507	13391	564	0.79	8.8	39	640000	570
+ shadows						0.72	7.4	36	1242050	1000
Fforest	174117	174116	348231	318.676	399	0.57	4.4	28	640000	450
+ shadows						0.64	7.9	33	1276012	950
Park	29174	29173	58345	264673	51	0.59	7.1	34	640000	450
+ shadows						0.61	7.2	37	1442358	1000
Sibenik	80479	80478	160955	132359	448	1.3	6.0	54	640000	740
+ shadows						1.0	11	69	1279989	1800
Teapots	200748	200747	401493	1673.67	390	0.51	3.1	29	640000	400
+ shadows						0.52	4.6	30	1728836	1050

Tabulka 1: `Legenda k tabulce:` *#Prims* - počet primitiv ve scéně, *#Leaves* - počet listů v hotovém BVH stromu, *#Nodes* - celkový počet uzlů v BVH (včetně listů), *SAH* - průměrná hodnota funkce SAH, T_B - čas výstavby BVH, T_R - čas traverzace jednoho paprsku, N_{IT} - průmerný počet incidentních operací s primitivy na jeden paprsek, N_{TR} - průměrný počet traverzací stromem BVH na jeden paprsek, N_Q - počet dotazů do BVH při syntéze obrazu (= počet hledaných průsečíků ve scéně, bez stínů konstantní a rovno poštu pixelů, se stíny je počet variabilní), *Rendertime* - celkový čas čisté syntézy obrazu (bez stavby BVH)

Hotová datová struktura byla otestována na deseti vzorových scénách s různým počtem primitiv a jejich rozložením ve scéně. Časy výstavby se jeví horší, než v referenčním článku [1], a však zhoršení není nijak drastické. Zpomalení může být způsobeno neefektivní implementací dělení primitiv, nebo příliš jemným dělením v úrovni listů (jeden list na jedno primitivum), a nebo špatným použitím SSE instrukcí. Časy traverzací (syntézy obrazu) jsou poměrně nízké, v případě výpočtů stínů je možné syntézu urychlit - místo hledání nejbližšího průsečíků hledat jakýkoliv.

Práce byla určitě dobrým přínosem a to především z hlediska zkušeností. Výsledek není nejlepší, ale zajisté poslouží svému účelu. Implementaci je možné optimalizovat a klidně 10x zrychlit, ovšem to je záležitost na další desítky hodin programování. Pro nejlepší výsledek by bylo zřejmě potřeba napsat nový raytracer spolu s datovou strukturou (BVH) pro lepší kontrolu nad kódem. V případě opakování této práce (s již nabytými zkušenostmi) bych se zaměřil na lepší využití SSE a na variabilní počet primitiv v listech BVH. Je totiž velmi pravděpodobné, že zmíněná distribuce jednoho primitiva na jeden list není vhodné řešení (vzhledem k času výstavby).

Poděkování

Vzhledem k náročnosti a obsáhlosti práce bych chtěl poděkovat svým spolužákům za jejich cenné rady, díky kterým jsem větší část problémů vyřešil snáze, než kdybych na to byl sám.

6. Reference

Reference

[1] Ingo Wald et al.: *On fast Construction of SAH-based Bounding Volume Hierarchies*. SCI Institute, University of Utah Intel Corp, Santa Clara, CA, 2007 [cit. 18.10.2020]. Dostupné na: `http://www.sci.utah.edu/~wald/Publications/2007///FastBuild/download//fastbuild.pdf/`

[2] CS3330: *A quick guide to SSE/SIMD*. University of Virginia [online]. [cit. 17.12.2020]. Dostupné na: `https://www.cs.virginia.edu/~cr4bd/3330/F2018/simdref.html`

[3] javidx9: *Intrinsic Functions - Vector Processing Extensions*. Youtube [online]. Copyright © 2020 Google LLC [cit. 15.12.2020]. Dostupné na: `https://www.youtube.com/watch?v=x9Scb5Mku1g`

[4] Hugh Chen: *Bounding Volume Hierarchy*. cs184-en. [cit. 27.12.2020]. Dostupné na: `http://hughchen.github.io/html/cs184/asst3/part2.html`