Diplomová práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Optimalizace hierarchií obalových těles

Bc. Vít Gardoň

Vedoucí: Ing. Jaroslav Sloup
Obor: Otevřená informatika
Specializace: Počítačová grafika
Květen 2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Gardoň** Jméno: **Vít** Osobní číslo: **499091**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Optimalizace hierarchií obalových těles

Název diplomové práce anglicky:

Optimization of Bounding Volume Hierarchies

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jaroslav Sloup Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **06.02.2025** Termín odevzdání diplomové práce: **23.05.2025**

Platnost zadání diplomové práce: **20.09.2026**

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis proděkana(ky) z pověření děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Gardoň** Jméno: **Vít** Osobní číslo: **499091**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Optimalizace hierarchií obalových těles

Název diplomové práce anglicky:

Optimization of Bounding Volume Hierarchies

Pokyny pro vypracování:

Seznamte se s metodou optimalizace hierarchií obalových těles [1] a detailně prostudujte její paralelní implementaci PRBVH pomocí technologie CUDA [2]. Rozšiřte PRBVH o možnost použít při stavbě hierarchie a následném procházení k-DOPy [4] a orientovaná obalová tělesa (OBB). Dále implementujte rozšíření o metriku T-SAH [5] pro optimalizaci stavby BVH pro animované scény, proveďte potřebné změny datových struktur a do frameworku doplňte načítání animovaných modelů. Upravte framework PRBVH tak, aby bylo možné v budoucnu snadno doplnit i nové typy metrik pro optimalizaci BVH.
Implementovaná rozšíření porovnejte na nejméně deseti statických a deseti animovaných scénách různé komplexnosti s původní metodou PRBVH, která využívá jako obalová tělesa pouze osově zarovnané kvádry (AABB). Vytvořte alespoň jednu novou komplexnější animovanou scénu. Implementované metody stavby optimalizovaných BVH vyhodnoťte z hlediska rychlosti stavby, vrhání různých typů paprsků, ceny SAH i dalších metrik (EPO a LCV) popsaných v [6].

Seznam doporučené literatury:

[1] J. Bittner, M. Hapala, V. Havran: Fast Insertion-based Optimization of Bounding Volume Hierarchies. Computer Graphics Forum, Vol. 32, No. 1, 2013. https://doi.org/10.1111/cgf.12000
[2] D. Meister, J. Bittner: Parallel Reinsertion for Bounding Volume Hierarchy Optimization. Computer Graphics Forum, Vol. 37, No. 2, 2018. https://github.com/meistdan/prbvh
[3] D. Meister, J. Bittner: Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing. Journal of Computer Graphics Techniques (JCGT), Vol. 11, No. 4, p. 1-19, 2022. https://github.com/meistdan/hippie
[4] M. Káčerik, J. Bittner: SAH-Optimized k-DOP Hierarchies for Ray Tracing. Proceedings of the ACM on Computer Graphics and Interactive Techniques, Vol. 7, No. 3, pp.1-16, 2024. https://doi.org/10.1145/3675391
[5] J. Bittner, D. Meister: T-SAH: Animation Optimized Bounding Volume Hierarchies. Computer Graphics Forum, Vol. 34, No. 2, 2015. https://doi.org/10.1111/cgf.12581
[6] T. Aila, T. Karras, S. Laine: On Quality Metrics of Bounding Volume Hierarchies. Proceedings of the 5th High-Performance Graphics Conference, pp.101–107, 2013. https://doi.org/10.1145/2492045.2492056

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

PROHLÁŠENÍ

Já, níže podepsaný

Příjmení, jméno studenta:	Gardoň Vít
Osobní číslo:	499091
Název programu:	Otevřená informatika

prohlašuji, že jsem diplomovou práci s názvem

Optimalizace hierarchií obalových těles

vypracoval samostatně a uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací a Rámcovými pravidly používání umělé inteligence na ČVUT pro studijní a pedagogické účely v Bc a NM studiu.

Prohlašuji, že jsem v průběhu příprav a psaní závěrečné práce použil nástroje umělé inteligence. Vygenerovaný obsah jsem ověřil. Stvrzuji, že jsem si vědom, že za obsah závěrečné práce plně zodpovídám.

V Praze dne 22.05.2025

Bc. Vít Gardoň
.....
podpis studenta

Poděkování

Děkuji panu Ing. Jaroslavu Sloupovi za odborné vedení práce, za všechny rady a hlavně za spoustu času stráveného na konzultacích.

Poděkování patří také mým rodičům, bez kterých bych se na své studijní pouti nikdy tak daleko nedostal.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně včetně všech podepsaných zdrojových kódů, a také že jsem uvedl veškerou použitou literaturu v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

.....
Vít Gardoň

V Praze, 22. května 2025

Abstrakt

Práce je zaměřena na využití paralelní optimalizace hierarchií obalových těles na grafické kartě pomocí znovuvkládání uzlů do hierarchie. Věnuje se rozšíření implementace PRBVH o obalová tělesa 14-diskrétní orientovaný polytop (14-DOP) a zkosený orientovaný kvádr (SOBB). Dále rozšiřuje implementaci na animované scény za pomoci temporálního rozšíření heuristiky plochy obalového tělesa (T-SAH). Implementované techniky jsou porovnávýny na testovací sadě scén pomocí základních metrik i metrik EPO a LCV, o které byla aplikace rozšířena. Aplikace je připravená pro snadné používání pomocí konfiguračního souboru a pro rozšíření dalšími obalovými tělesy či metrikami pro optimalizaci pomocí PRBVH.

Klíčová slova: CUDA, BVH, hierarchie obalových těles, optimalizace, SAH, heuristika plochy obalového tělesa, T-SAH, k-DOP, PRBVH, SOBB

Vedoucí: Ing. Jaroslav Sloup

Abstract

This thesis focuses on parallel optimization of bounding volume hierarchies on a graphics card by reinserting nodes into the hierarchy. It extends the PRBVH implementation by including a 14-discrete polytope (14-DOP) and a skewed oriented bounding box (SOBB). It also extends the implementation to animated scenes using the temporal extension of the surface area heuristic (T-SAH). The implemented techniques are compared on a test set of scenes using the basic metrics as well as the EPO and LCV metrics, which have been implemented in the application. The application is ready to be easily used with a configuration file and extended with additional bounding volumes or metrics for optimization using PRBVH.

Keywords: CUDA, BVH, bounding volume hierarchy, optimization, SAH, surface area heuristic, T-SAH, k-DOP, PRBVH, SOBB

Title translation: Optimization of Bounding Volume Hierarchies

Obsah

1 Úvod **1**

1.1 Cíle práce 2

2 Analýza a návrh řešení **3**

2.1 Hierarchie obalových těles 3

2.2 Základní metrika SAH 5

2.3 Typy obalových těles 7

2.4 Další metriky 10

2.5 Tvorba hierarchií 17

2.5.1 Lineární BVH - LBVH 18

2.5.2 Opětovné vkládání uzlů v hierarchii obalových těles - RBVH 19

2.5.3 Paralelní opětovné vkládání uzlů v hierarchii obalových těles - PRBVH 19

2.5.4 Optimalizace hierarchie pro animace 24

2.5.5 Rotace uzlů stromu 26

2.5.6 Další optimalizační algoritmy na GPU 27

2.6 NVIDIA framework 29

2.7 Návrh implementace 31

3 Implementace **33**

3.1 Vylepšení frameworku 33

3.2 Přechod na novou verzi CUDA . 35

3.3 Rozšíření implementace - třída HybridBVH 37

3.3.1 Implementace rozděleného 14-DOPu 39

3.3.2 Varianty uložení SOBB 40

3.3.3 Generování normál SOBB . . . 41

3.3.4 Výběr normál SOBB 41

3.3.5 Správný výpočet plochy pro SOBB 42

3.3.6 Náročnost výpočtu EPO 45

3.3.7 Implementace T-SAH 49

3.3.8 Nově vytvořené scény 50

4 Měření **51**

4.1 Metody měření 51

4.1.1 Kernel vrhání paprsků 54

4.2 Analýza výsledků 57

4.2.1 Porovnávání variant obalových těles 57

4.2.2 Statické scény 62

4.2.3 Animované scény 66

4.2.4 Stavba pomocí refitu z AABB 72

4.2.5 Porovnání výsledků s články . 74

5 Závěr **82**

Literatura **84**

A Návod na implementaci nového typu obalového tělesa **88**

B Návod na implementaci nové metriky optimalizované pomocí PRBVH **90**

C Struktura odevzdaných zdrojových kódů **91**

Obrázky

2.1 Ukázka dvou možných hierarchií nad stejnými trojúhelníky 6

2.2 Ukázka 14-DOPu 9

2.3 Ukázka zkoseného kvádru generovaného průnikem tří desek . . 10

2.4 Ukázka situace, kdy se musí projít další uzly i po najití prvního průsečíku s trojúhelníkem. 11

2.5 Ukázka započtené plochy při výpočtu EPO 12

2.6 2D ukázka zorného pole bodu $\overrightarrow{x}$ pokrytého uzlem N. 15

2.7 Rozdělení scény obalovým tělesem AABB na 27 částí pomocí polorovin AABB 16

2.8 Ukázka Mortonových kódů pro oblast 4x4. 19

2.9 Oblasti na cestě uzlu na jeho novou pozici. 20

2.10 Ukázka přesunutí uzlu na jeho novou pozici. 22

2.11 Ukázka oblastí hierarchie, kde se mění hodnota EPO. 23

2.12 Ukázka vývoje animované scény Exploding dragon 25

2.13 Ukázka možných rotací uzlů . . 26

2.14 Ukázka treeletu 28

2.15 Workflow diagram fungování hlavních částí frameworku 30

3.1 Vizualizace částí SOBB vyčnívající z předka. 44

3.2 Ukázka hierarchie obalových těles pro výpočet metriky EPO. 46

3.3 Náhled dvou nově vytvořených scén. 50

4.1 Hodnota SAH obalového tělesa AABB napříč snímky pro scénu Exploding dragon. 75

4.2 Poměr času stráveného v kernelech pro scénu San Miguel. 77

4.3 Vývoj hodnoty SAH během optimalizace pomocí PRBVH a trvání jednotlivých iterací pro scénu Hairball 78

4.4 Vývoj hodnoty T-SAH a trvání jednotlivých iterací během optimalizace scény Exploding dragon. 79

4.5 Vizualizace počtu uzlů hierarchie a součtu ploch těles v závislosti na hloubce uzlu. 80

4.6 Vizualizace 19. úrovně obalových těles na scéně San Miguel. 81

Tabulky

4.1 Přehled použitých scén. 53

4.2 Měření rychlosti vrhání paprsků v závislosti na počtu perzistentních warpů. 55

4.3 Měření rychlosti vrhání paprsků v závislosti na velikosti bloku kernelu. 56

4.4 Měření rychlosti vrhání paprsků v závislosti na rozložení 14-DOPu v paměti. 58

4.5 Měření rychlosti vrhání paprsků v závislosti na reprezentaci SOBB v paměti. 58

4.6 Měření rychlosti vrhání paprsků pro SOBB v závislosti na počtu normál. 59

4.7 Měření rychlosti vrhání paprsků v závislosti na počtu a strategii výběru normál SOBB. 60

4.8 Porovnání hodnoty SAH pro SOBB 16 pro různé způsoby ořezávání. . . 61

4.9 Porovnání hodnoty SAH pro SOBB 16 pro optimalizované PRBVH s různými způsoby ořezávání. 62

4.10 Soubor měření statických scén. 64

4.11 Porovnání průměrného zrychlení výkonu napříč statickými deseti scénami. 66

4.12 Soubor změřených animovaných scén. 67

4.13 Průměrné poměry metrik pro animované scény 72

4.14 Porovnání tvorby obalových těles vytvořených optimalizací PRVBH oproti optimalizaci AABB a následném refitu na dané těleso. . . 73

4.15 Porovnání naměřených hodnot SAHu s původními články. 74

4.16 Porovnání výkonu AABB, 14-DOPu a SOBB 3 s totožným tvarem těles. 76

4.17 Porovnání SAHu a výkonu AABB pro různé hodnoty parametru *mod*. 79

Kapitola 1

Úvod

Počítačová grafika je oblastí informatiky, která se zabývá generováním, zpracováním a zobrazováním obrazových dat. Její význam neustále roste s tím, jak stoupají nároky na vizuální kvalitu v oblasti filmového a herního průmyslu i virtuální a rozšířené reality. Základním kamenem realistického vykreslování scén je aproximace globálního osvětlení. Tradičně se při vykreslování v reálném čase používá rasterizace, která převádí 3D scénu do 2D obrazu poměrně efektivním, avšak zjednodušeným způsobem. K dosažení realistických výsledků používá stínové mapy, zjednodušené světelné zdroje a další triky.

Dokonalejší metodou vykreslování je ray tracing, který nabízí možnost simulovat paprsky světla a umožňuje tím snadněji dosáhnout realističtějších výsledků. Předpokladem k použití této metody je dostašný výkon grafické karty. Ještě před několika lety jsme si mohli o vykreslování pomocí ray tracingu v reálném čase nechat jen zdát. Dnes se již ray tracing stává normou (i když se stále musí používat agresivní denoising algoritmy kvůli nedostatku paprsků), jako tomu doposud byla rasterizace. Některé hry si dokonce bez grafické karty, která ray tracing podporuje, nezahrajete.

Přestože se již dnes ray tracing běžně používá, a to nejen v oblasti offline renderování, je stále kam se posouvat. Nemůžeme spoléhat pouze na větší paralelizaci výpočtů vrhání paprsku scénou a na hardware dedikovaný k jejímu procházení. Je potřeba vylepšit i datové struktury akcelerující výpočet.

Naivní vrhání paprsků je velmi neefektivní. Zkoušet vypočíst průsečík paprsku s každým primitivem (převážně s trojúhelníky) jde provést v lineárním $O(n)$ čase, ale použitím kvalitní datové struktury se může dostat k $O(log(n))$.

Mezi hlavní datové struktury patří k-d stromy [1] a hierarchie obalových těles (tzv. BVH[1]) [2]. Tato práce je zaměřena na BVH, jelikož s ní pracují optimalizační algoritmy a metriky, ze kterých tato práce vychází. Právě s BVH pracují NVIDIA RTX karty, které umí jejich strukturu velmi efektivně procházet [3].

1.1 Cíle práce

Nyní popíši jednotlivé části této práce. Kapitola 2 rozebírá různé typy hierarchií a metrik, kterými se dá odhadnout jejich kvalita. Popisuje také algoritmy použitelné pro postavení lepší hierarchie. V kapitole 3 je popsáno, jak byl upraven a rozšířen framework, na kterém je práce založena. Popisována je implementace nových obalových těles a metrik. Měření provedená s nově naimplementovanými obalovými tělesy a metrikami jsou zdokumentována v kapitole 4, kde se vyhodnocují výsledky a porovnávají se s předešlými pracemi.

Práce má za cíl prozkoumat, zda jiná obalová tělesa mají větší potenciál optimalizovat SAH než AABB, protože dokáží těsněji obalit geometrii. Zajímá nás, zda vůbec existují hierarchie, které by vrhání paprsků urychlily bez ohledu na to, jak dlouho se hledají.

[1] BVH - Bounding Volume Hierarchy

Kapitola 2

Analýza a návrh řešení

V této části práce se zabýváme hierarchií obalových těles a metrikami jako jsou SAH, T-SAH, EPO, LCV a další popisující kvalitu hierarchií. Prozkoumává algoritmy použitelné pro tvorbu hierarchií, a to především ty, které lze použít na grafické kartě. Je zde popsán návrh implementace, který bude použit pro rozšíření aplikace o nová obalová tělesa a metriky.

2.1 Hierarchie obalových těles

Jak je již z asymptotické složitosti zjevné, používá BVH stromovou (většinou binární) strukturu. Každý vnitřní uzel obsahuje referenci na potomky, kteří mohou opět být vnitřními uzly nebo listy. Listy obsahují reference na jeden nebo více trojúhelníků, zároveň každý uzel obsahuje definici obalového tělesa, které obaluje veškeré trojúhelníky a uzly v podstromu tohoto uzlu. Při vrhání paprsků se tato hierarchie prochází. Začneme v kořenu stromu a otestujeme průsečík obalových těles potomků s paprskem a rekurzivně nebo pomocí zásobníku se zanoříme do těchto protnutých potomků. Když se dostaneme až k listu, musíme vypočítat průsečík se všemi trojúhelníky v něm.

Během procházení je možné některé uzly přeskočit, jelikož po protnutí s paprskem známe jejich vzdálenost od počátku paprsku. Pokud je tato vzdálenost k tělesu uzlu větší než nejbližší již nalezený průsečík s trojúhelníkem, víme, že není možné, aby nějaký trojúhelník patřící do podstromu testovaného uzlu byl blíž než aktuální kandidát na nejbližší trojúhelník.

Typy hierarchií

Většina používaných algoritmů je implementována jako binární. Binární hierarchie je nejjednodušší typ hierarchie, jelikož při jejich vytváření hledáme dva uzly, které spojíme, nebo dělíme trojúhelníky na dvě skupiny. To limituje počet možných řešení a usnadňuje implementaci pomocí binárního fungování počítačů.

Nemusíme se však omezovat pouze na binární hierarchie. Algoritmus procházení binárního BVH bude fungovat i na hierarchii s více potomky. Pokud každý uzel má místo dvou potomků například čtyři, stane se tím hierarchie mělčí. Když budeme předpokládat, že binární hierarchie je maximálně vyvážena, její výška se blíží $log_2(n)$. Když povolíme více potomků, tak je to $log_k(n)$, kde k je počet potomků. Hierarchie s větší aritou mají výhodu, jelikož hloubka stromu může značně ovlivnit výpočet, a to především na GPU. Stromy s větší aritou umožňují lepší využití SIMD instrukcí a spočítání více průsečíků najednou. Více potomků se také může umístit v paměti za sebou a tím zlepšit jednotnost načítání dat.

Tyto širší stromy [4] však potřebují při procházení mnohem větší zásobník nebo se musí značně zkomprimovat [5]. Na zásobník se odkládají ty uzly, které byly při procházení protnuty, ale ještě nebyly nezpracovány. Nejdřív se totiž vyplatí zpracovat nejbližší uzel, jelikož se tím zvyšuje šance na případné uřezání uzlů, když jsme již našli kandidáta na nejbližší trojúhelník. Vezmeme-li vyváženou hierarchii o 256 trojúhelnících, tak při dvou potomcích na uzel má 9 úrovní (včetně kořene) a tudíž potřebujeme zásobník o velikosti alespoň 8. Na cestě k listu narazíme na osm uzlů, kde se vydáme do levého nebo pravého uzlu, a zbylý uzel si musíme zapamatovat v případě, že ho paprsek také protne. Avšak když použijeme hierachii s aritou 4, tak strom má jen 5 úrovní, ale na každém ze čtyř uzlů, kde si vybíráme cestu, si musíme zapamatovat až tři zbylé protnuté uzly. Tím pádem potřebujeme zásobník o velikosti 12. Tento problém se zhoršuje se zvětšující scénou. Nemluvě o tom, že uvažujeme ideální situaci, kdy je hierarchie vyvážená. Tyto počty jsou pouze teoretické. V praxi se málokdy vytvoří hierarchie, kde by existovala konfigurace paprsku, aby protnul všechny uzly na každé vrstvě. To může nastat jen u opravdu nekvalitní hierarchie nad specifickou geometrií scény.

Řešením tohoto problému může být třeba technika krátkého zásobníku (short stack) [4]. Ta umožňuje drasticky snížit velikost zásobníku. Kromě zásobníku má i pro každou úroveň hierarchie uložené číslo, určující kolik protnutých potomků již zpracovala na posledním navštíveném uzlu dané úrovně. Použití Zásobníku je standardní, jen když se zaplní, začnou se zapomínat nejstarší uzly. Když je zásobník prázdný, začne se znovu od kořene a pomocí

uložených hodnot se dojde na místo, kde se skončilo, ale zásobník je již opět naplněný.

Tato technika má nevýhodu v tom, že vyžaduje při návratu opět testovat průsečíky s obalovými tělesy uzlů, jelikož uložená hodnota v úrovni je počet zpracovaných protnutých potomků a ne počet potomků, kteří byli protnuti. Naštěstí situací, kdyby se protnulo více potomků, není tolik, a proto se zásobník nezaplňuje příliš rychle.

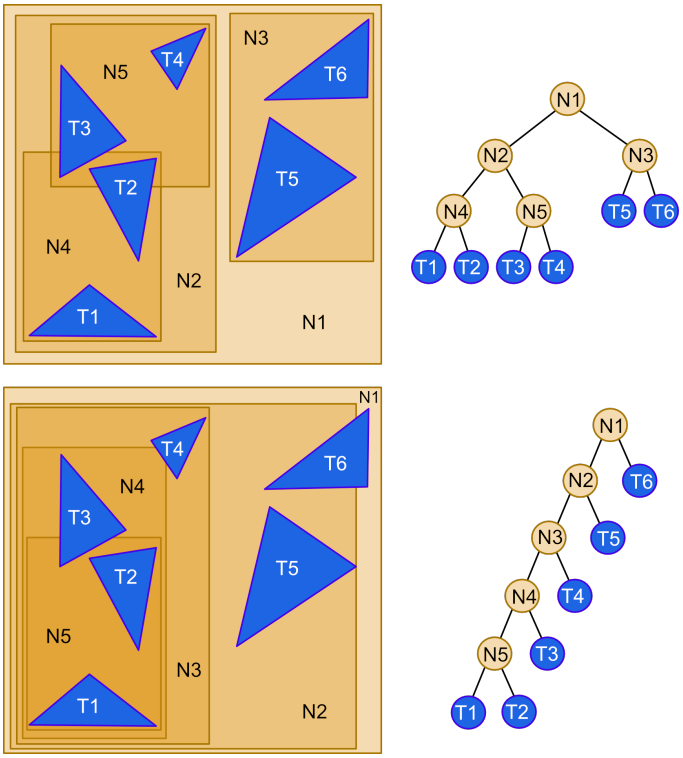
2.2 Základní metrika SAH

Způsobů jak uspořádat trojúhelníky do hierarchie je velké množství. Je očividné, že pokud chceme dosáhnout časové složitosti $O(log(n))$, musíme se co nejvíce přiblížit vyváženému stromu. Avšak i vyvážených stromů je velký počet. My ale hledáme takový vybalancovaný strom, kde budeme muset procházet co nejméně uzlů. Potomkem neprojdeme, pouze pokud paprsek nekoliduje s obalovým tělesem uzlu. Právě s pravděpodobností protnutí obalového tělesa se snaží optimální hierarchii najít metrika založená na výpočtu plochy obalového tělesa SAH (surface area heuristic) [6]. V základní definici SAH se počítá s tím, že paprsek má stejnou pravděpodobnost, že přiletí z jakéhokoliv směru a místa. Dále platí předpoklad, že paprsky jsou nezastíněné jinou geometrií. Situaci můžeme popsat jako rovnoměrný mrak paprsků. Když si tedy představíme obdélníkový otvor s plochou $1m^2$, tak i když jej umístíme na libovolné místo, bude pravděpodobnost, že jím proletí paprsek, stejná. V případě že velikost otvoru zdvojnásobíme (nyní $2m^2$), zdvojnásobí se i pravděpodobnost na průnik paprsku. Pokud obdélníkový otvor nahradíme plochou obalového tělesa uzlu, dojde k tomu, že pravděpodobnost na protnutí s uzlem je závislá na plošw obalového tělesa. Pokud bychom chtěli a zároveň měli o scéně dostatečnou znalost, mohli bychom počítat i s nerovnoměrným rozložením paprsků a dát tak větší váhu paprskům z určitého směru. Tato možnost v této práci však není prozkoumávána, jelikož se zaměřuje na obecné použití scény, ale bylo by možné ji pro takovéto případy upravit.

Nyní když víme, že záleží na ploše obalového tělesa, můžeme formulovat rekurzivní rovnici 2.1 na výpočet ceny, kterou se budeme snažit optimalizovat.

$$c(N) = \begin{cases} C_T + \frac{\mathrm{SA}(N_L)}{\mathrm{SA}(N)} c(N_L) + \frac{SA(N_R)}{\mathrm{SA}(N)} c(N_R), & \text{vnitřní uzel,} \\ C_I |N|, & \text{list.} \end{cases} \tag{2.1}$$

Rovnice 2.1: $c(N)$ je SAH uzlu včetně jeho podstromu, $|N|$ je počet trojúhelníků v uzlu, $SA(N)$ je plocha obalového tělesa. N_L a N_R jsou potomci uzlu. C_I je cena průsečíku trojúhelníku v listu a C_T cena průchodu k novému uzlu hierarchie včetně průsečíku jeho obalového tělesa.



Obrázek 2.1: Ukázka dvou možných hierarchií nad stejnými trojúhelníky. Hierarchie dole má SAH přibližně o 17% větší.

V případě předpokladu, že paprsky jsou rovnoměrně rozložené a nezastíněné, můžeme použít zjednodušenou nerekurzivní verzi výpočtu 2.2, která se bude hodit pro její nezávislost na ostatních uzlech při výpočtech na grafické kartě [7].

$$c(N) = \frac{1}{\mathrm{SA}(N)} \left[C_T \sum_{N_i} \mathrm{SA}(N_i) + C_I \sum_{N_l} \mathrm{SA}(N_l)|N_l| \right] \tag{2.2}$$

Rovnice 2.2: $c(N)$ je SAH uzlu, N_l je počet trojúhelníků v uzlu, $SA(N)$ je plocha obalového tělesa. N_i je množina všech vnitřních uzlů a N_l je množina všech listů.

Na obrázku 2.1 je vidět výhodu mělčí hierarchie. Paprsky, které mají nejbližší průsečík v hlubokém listu, musí ve spodní hierarchii vždy spočítat průsečík se všemi trojúhelníky nad ním, jelikož je potřeba zkontrolovat, že ty nejsou ještě před ním. Mělčí hierarchie má těchto trojúhelníků méně. Pokud bychom vzali všechny paprsky, které protnou trojúhelník T1 jako první, tak ve spodní hierarchi musíme vždy zkontrolovat všechny trojúhelníky. Stejná situace nastane u vrchní hierarchie, pouze pokud jsou protnuta všechna obalová tělesa. Paprsků, které jsou toho schopny, je relativně málo, a proto bude mít vrchní hierarchie lepší výkon. Toto je velmi zjednodušený příklad. Ve scénách s miliony trojúhelníků se takové hierarchie nevyplácí ještě víc. Hluboké hierarchie nejsou také vhodné pro GPU, jelikož častěji nastává situace, kdy část vláken najde list a chce vypočítat bod protnutí trojúhelníku, ale musí čekat na ostatní, jelikož zrovna spadají do hluboké větve. Tím vzniká velké zpomalení.

2.3 Typy obalových těles

Naší snahou je získat BVH s co nejmenší SAH cenou. Je tedy potřebné najít obalové těleso, které by minimalizovalo potřebnou plochu pro obalení podstromu. Toto těleso musí být co nejvíce "těsné", aby se tím minimalizovala situace, kdy se těleso protne, ale paprsek poté již uvnitř neprotne obalová tělesa či trojúhelníky potomků. To by jen zbytečně zvětšovalo SAH.

Musíme však počítat i s konstantou C_T, která se bude pro různé typy obalových těles měnit. Je totiž jednodušší spočítat průsečík paprsku s kvádrem, než třeba s konvexním obalovým tělesem tvořeným trojúhelníky. V této situaci se již výhoda větší těsnosti tělesa může vytratit, protože konstanta C_T je příliš velká. Je proto nutné použít těleso dostatečně těsné a zároveň takové, aby byl jeho výpočet průsečíku dostatečně rychlý. Tento výběr je také závislý na scéně. Jeden typ tělesa může excelovat v jednodušší scéně, ale naprosto selhat ve složitější.

Dále si popíšeme typy obalových těles relevantních k této práci.

Osově zarovnaný kvádr

Osově zarovnaný kvádr (Axis Aligned Bounding Box dále jen AABB) je jeden z nejjednodušších typů obalových těles. K jeho uchování v paměti stačí dva body reprezentující minimální a maximální hodnoty na osách XYZ. Jelikož paprsek protíná vždy plochu zarovnanou s jednou z os souřadnicového systému, je jeho výpočet průsečíku rychlý (nemusí se počítat skalární součin paprsku s osou). Jeho nevýhodou však je situace, kdy trojúhelník nebo pár uzlů nejsou zarovnány s některou z rovin XY, XZ nebo YZ. Výsledné těleso je pak zbytečně velké, obsahuje spoustu prázdného prostoru a tudíž zvětšuje SAH.

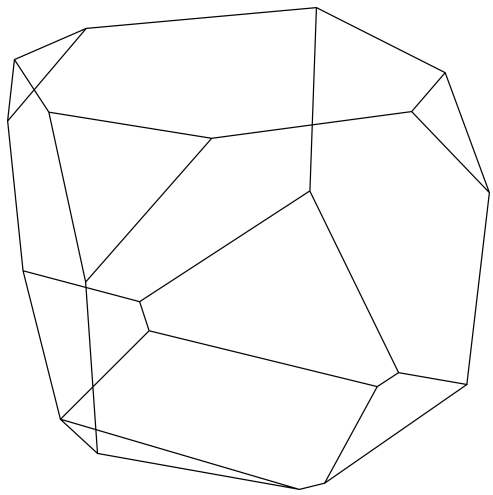
Orientovaný kvádr

Mezi často používaná obalová tělesa patří i OBB (oriented bounding box). Jedná se obecnější variantu AABB. Lépe umožňuje přizpůsobit se geometrii díky volnému natočení kvádru. Velkou nevýhodou jsou však obtíže při hledání nejlepšího OBB. Možností, jak těleso natočit, je totiž příliš. Proto se většinou uchylujeme k výběru orientací pomocí nějakého jiného tělesa, což zmenší prohledávaný prostor řešení na mnohem menší množinu řešení, ze které je nejlepší OBB vybráno [8].

K-diskrétní orientovaný polytop

Nutnosti otáčet kvádr k dosažení větší těsnosti obalového tělesa se snaží zabránit k-DOP (k-Discrete Oriented Polytope). Jedná se o těleso tvořené průnikem několika polorovin. Existuje více druhů k-DOPů. Hodnota k definuje počet unikátních polorovin. Ty jsou předdefinované a nemění se uzel od uzlu. Poloroviny tvoří páry, které jsou na jedné ose. Na každou osu tedy připadá hodnota minima a maxima, které dohromady dávají průnik dvou polorovin tvořící tzv. desku (slab). Pokud bychom použili 6-DOP s osami $(1, 0, 0)$, $(0, 1, 0)$ a $(0, 0, 1)$, tak bychom ve výsledku dostali AABB. Zajímavější situace nastává, když se osy rozšíří ještě o diagonály $(1, 1, 1)$, $(-1, 1, 1)$, $(1, -1, 1)$, $(-1, -1, 1)$. Potom tato série definuje 14-DOP. Jedná se vlastně o AABB s ořezanými rohy (Obrázek 2.2). To umožňuje vytvořit těsnější obalová tělesa,

jelikož hluchá místa nakloněných trojúhelníků či nezarovnaných obalových těles uzlů se dají vyloučit z výsledného tělesa. Výpočet protnutí paprsku s k-DOPem je podobný AABB a můžeme říci, že jeho komplexita roste lineárně s počtem os. Os je však více jak dvakrát tolik, což znamená, že výpočet je teoreticky více jak dvakrát tak drahý.



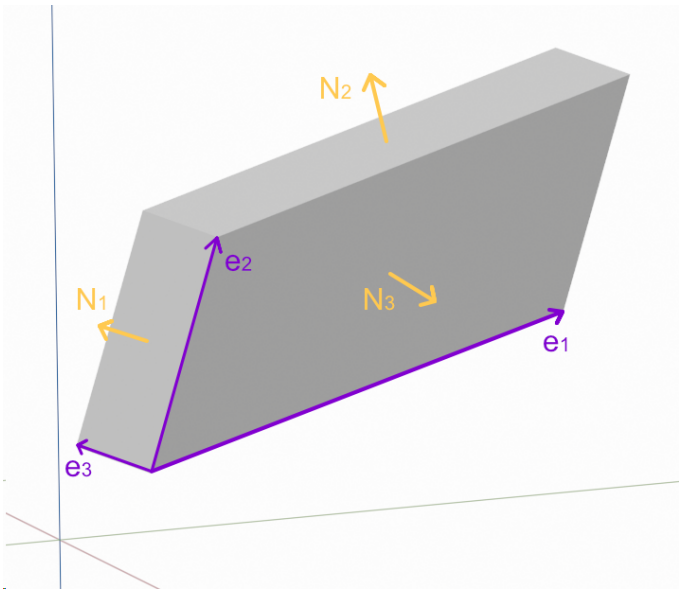
Obrázek 2.2: Ukázka 14-DOPu - Příklad ukazuje ořezané rohy. Zadní pravý horní roh zůstává celý, jelikož polorovina jím neprochází.

14-DOP je schopen dosahovat lepších hodnot SAH metriky, ale problém nastává při výpočtu jeho plochy, která je potřebná k optimalizaci hierarchie pomocí SAH. Výpočet již není tak přímočarý, jak tomu bylo u AABB. Je možné vypočítat vrcholy průniků rovin pomocí iteračních metod a následně spočítat plochu mezi nimi. Iterační metody jsou však velmi pomalé oproti přesnému výpočtu AABB. Naštěstí Káčerik a Bittner [9] definovali exaktní metodu, jak spočítat plochu 14-DOPu. Tato metoda vypočítá plochu trojbokých jehlanů, které jsou ukrojeny z rohů AABB, a tu následně odečte od původního AABB. Mohou však nastat situace, kdy se tyto jehlany překrývají a je proto nutné udělat korekci, která zabrání dvojitému odečtení některých míst.

Zkosený orientovaný kvádr

Zkosený orientovaný kvádr [10] (Skewed Oriented Bounding Box dále jen SOBB) neboli rovnoběžník ve 3D (viz Obrázek 2.3) je útvar tvořený průnikem šesti polorovin s tím, že páry vždy sdílí normálu. Jedná se o relativně jednoduché těleso. Výpočet průsečíku paprsku s SOBB se od AABB liší pouze směrem normál. Tento směr však musí být někde uložen, čímž se zvětšuje ve-

likost uzlů finální hierarchie. Pro množinu bodů existuje nekonečno možností jak vybrat tři definující normály. Hledat nejoptimálnější variantu je možné pomocí pokročilých optimalizačních technik, ale tento přístup by se jen těžko používal v takovém měřítku, jaké si konstrukce BVH vyžaduje. Vystačíme si s výběrem normál z k-DOPu, nad kterým je SOBB postaveno.



Obrázek 2.3: Ukázka zkoseného kvádru generovaného průnikem tří desek podle normál N_1, N_2 a N_3 nebo třemi hranami e_1, e_2 a e_3.

Další typy obalových těles

Existují i další obalová tělesa [11]. Každé má své unikátní klady a zápory. K významým patří tažená koule (swept sphere neboli capsule), která se často používá při detekci kolizí v herních enginech, nebo prostá koule, jenž je jednou z nejjednodušších obalových těles.

2.4 Další metriky

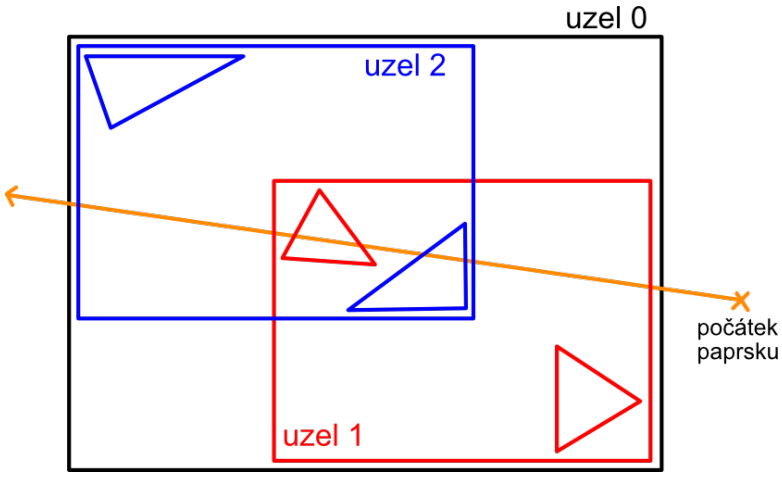
I když SAH je nejrozšířenější, není jedinou metrikou, která pomáhá určit kvalitu hierarchie a následný výkon. Existují jiné metriky zaměřující se na jiné vlastnosti hierarchie než je plocha obalového tělesa a tudíž pravděpodobnosti jeho protnutí.

Dále zmíníme vybrané metriky relevantní pro tuto práci.

Metrika EPO

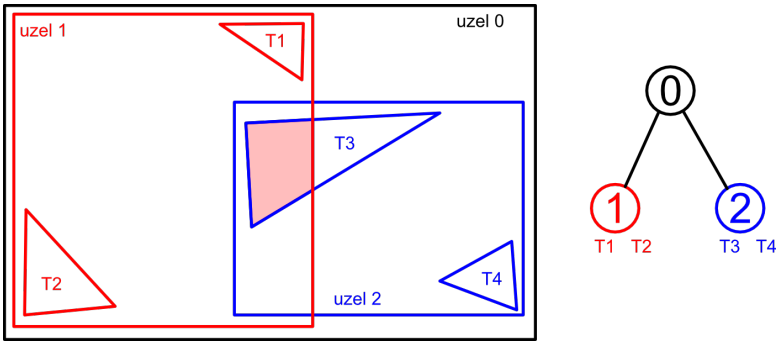
EPO neboli End-point overlap [12] (překrytí koncových bodů) je metrika zaměřující se na neefektivitu hierarchie pramenící z překryvu obalových těles. Pokud procházíme hierarchii, narazíme na list a v něm protneme trojúhelník, musíme ještě doprojít všechny uzly, které mají vstupní bod blíž než je protnutý trojúhelník, abychom vyloučili možnost, že v těchto uzlech se nachází nejbližší trojúhelník. Počet takovýchto uzlů jde omezit tím, že v hierarchii bude co nejmenší překryv obalových těles nepatřících do jednoho podstromu.

Podívejme se na situaci na obrázku 2.4. Algoritmus procházení hierarchii je napsán tak, aby procházel nejprve potomky, kteří jsou blíž počátku paprsku. To zajistí větší šanci, že díky již nalezenému průsečíku můžeme přeskočit později zpracovávané vzdálenější uzly. Může však nastat situace, jako je tomu u uzlů 1 a 2, kdy je uzel 2 dál a přesto obsahuje trojúhelník, který je nejblíže. Tato situace nastala, protože trojúhelník uzlu 2 zasahuje do uzlu 1. Je zjevné, že kvalitní hierarchie by měla těchto situací obsahovat co nejméně. Pokud by si uzly vyměnily prostřední trojúhelníky, tak by tato situace nenastala.



Obrázek 2.4: Ukázka situace, kdy se musí projít další uzly i po najití prvního průsečíku s trojúhelníkem.

EPO změří pro všechny uzly plochu trojúhelníků, které nepatří do podstromu analyzovaného uzlu, ale přesto se nacházejí v obalovém tělese uzlu (viz Rovnice 2.3). Čím menší tento překryv bude, tím rychlejší by následné vrhání mělo být. Stejná situace jako nastává u koncového bodu paprsku může



Obrázek 2.5: Ukázka započtené plochy při výpočtu EPO. Pouze červeně vyznačená část je započítána pro uzel 1. Hodnota EPO pro uzel 2 je nulová.

nastat u jeho počátku. Je lepší, když se počátek vyskytuje právě v jednom listu, než abychom museli prohledávat hierarchii "na druhou stranu", kudy paprsek nemíří jen proto, abychom ověřili, že trojúhelník není před paprskem.

$$EPO := \sum_{n \in N} C_n \frac{SA((S \setminus Q(n)) \cap n)}{SA(S)} \tag{2.3}$$

Rovnice 2.3. C_n je cena za zpracování uzlu, je to buď C_i pro vnitřní uzly nebo $C_t Tri_{count}$ pro listy. Tri_{count} je počet trojúhelníků v listu, S jsou veškeré trojúhelníky ve scéně, n je aktuálně zpracovaný uzel s jeho obalovým tělesem. $Q(n)$ je množina trojúhelníků, které patří do podstromu uzlu n. Funkce $SA()$ vypočte plochu obalového tělesa uzlu.

Z rovnice 2.3 je vidět, že je potřeba započítávat pouze plochu trojúhelníků, která leží uvnitř jiných uzlů a ne celou plochu trojúhelníku. Na obrázku 2.5 je červeně vyznačena jediná část, která bude započtena. Trojúhelník T_3 totiž částečně leží v uzlu 1.

Metrika LCV

Leaf count variability [12] (variabilita počtu listů) se snaží kvantifikovat, jak dobře se bude hierarchie procházet pomocí SIMD instrukcí a SIMT[1] instrukcí na GPU. Popíšeme zde problémy nastavající při vrhání paprsků na GPU, ale podobné problémy nastávají při zpracování na CPU pomocí SIMD instrukcí.

Mějme běžný kernel (Algorithm 1), který pro paprsek hledá nejbližší průsečík s trojúhelníkem. Hierarchie je postavená tak, že trojúhelníky se nacházejí až v listech. Musíme tedy projít přes několik vnitřních uzlů. Počet uzlů, které musíme navštívit, než dorazíme k listu je hloubka listu. Ta není pro většinu hierarchií pro všechny listy stejná. To znamená, že některá vlákna ve warpu opustí první cyklus dřív a musí čekat na ostatní, která ještě svůj list hledají. V moderní CUDA vlákna sice nejsou synchronní a mohou být v jiné části programu, ale i tak musí zpracovávat stejnou instrukci. Když polovina vláken potřebuje násobit a druhá polovina dělit, tak se navzájem blokují a vytížení je jen 50%. Chceme tedy, aby vlákna dělala pořád to stejné a nemusela na sebe čekat. Když vlákna následně zpracují trojúhelníky v listu některá z nich mají ještě v zásobníku uzly, které je potřeba zkontrolovat. Těchto uzlů je také opět jiný počet a dochází k divergenci vláken, nemluvě o tom, že instrukce pro zpracování listu a uzlu se značně liší. To vše znamená, že se výkon snižuje.

Chceme tedy říci, že hierarchie, kde vlákna prochází stejný počet vnitřních uzlů a listů, jsou lepší, jelikož nenastává divergence. LCV se zaobírá pouze počtem navštívených listů. Je to vlastně odchylka od počtu projitých listů pro množinu vzorkových paprsků (viz Rovnice 2.4). Odchylka je větší, když se počet listů vlákno od vlákna hodně mění. Menší číslo je tedy lepší.

$$LCV := \sqrt{E[N_l^2] - E[N_l]^2} \tag{2.4}$$

Rovnice 2.4. N_l je počet navštívených listů vzorkovým paprskem.

Metrika pro paprsky uvnitř scény

Článek od Fabianowskiho [13] poukazuje na to, že SAH počítá s paprsem s počátkem mimo scénu. To však může platit pouze pro primární paprsky, jelikož sekundární i stínové vždy začínají na některém z trojúhelníků. Ten se

[1] SIMT - single instruction multiple treads

Algorithm 1 Zjednodušený while while kernel pro procházení BVH

```
stack ← []
leaf ← null
while stack is not empty do
    while stack is not empty And leaf is null do
        node ← stack.pop()
        if node is leaf then
            leaf ← node break
        end if
        TminLeft ← intersectNode(ray, node.left)
        nodeFirst ← node.left
        TminRight ← intersectNode(ray, node.right)
        nodeSecond ← node.right
        if (TminLeft >= 0) ≠ (TminRight => 0) then    ▷ Jeden uzel protnut
            if TminLeft >= 0 then
                stack.push(nodeFirst)
            else
                stack.push(nodeSecond)
            end if
        else
            if TminLeft >= 0 then                     ▷ Oba uzly protnuty
                if TminLeft > TminRight then
                    swap(nodeFirst, nodeSecond)
                end if
                stack.push(nodeSecond)
                stack.push(nodeFirst)
            end if
        end if
    end while
    if leaf is not null then
        tri_address ← leaf.start
        while tri_address < leaf.end do
            intersectLeaf(ray, tri_address)
            tri_address ← tri_address + 1
        end while
        leaf ← null
    end if
end while
```

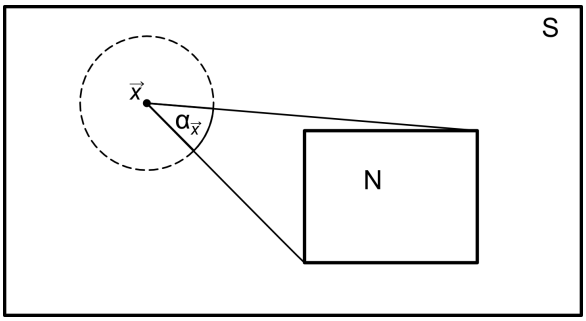
samozřejmě ve scéně nachází, a tudíž i paprsky z něj vychází. Článek přistupuje k pravděpodobnosti protnutí obalového tělesa uzlu z jiného pohledu.

Paprsky jsou stále rovnoměrně rozložené ve scéně, tudíž v každém bodě si můžeme představit paprsek vycházející na všechny strany. Pravděpodobnost protnutí paprsku a obalového tělesa je tedy, do jak velké části "zorného pole"(viz Obrázek 2.6) počátku paprsku těleso zasahuje. Pokud bude paprsek začínat těsně nad povrchem tělesa, je šance na protnutí skoro 50 na 50, jelikož necelá polovina zorného pole je pokryta tělesem. Naopak paprsek s počátkem daleko od tělesa může mít pokrytou jen jeho nepatrnou část.

Je tedy potřeba vypočítat zakrytou oblast zorného pole obalovým tělesem a to jde pomocí projekce na jednotkovou sféru a následné výpočtení obsahu vzniklého obrazce. Tento výpočet se musí opakovat pro každý bod mimo těleso. Když ještě přidáme pravděpodobnost, že paprsek začíná v obalovém tělese, dostaneme rovici 2.5 určující pravděpodobnost navštívení uzlu.

$$p_N = \frac{V(N)}{V(S)} + \frac{1}{V(S)} \int_{S \setminus N} \frac{\alpha_{\overrightarrow{x}}}{4\pi} d\overrightarrow{x} \tag{2.5}$$

Rovnice 2.5. $V()$ funkce objemu, S obalové těleso scény, N těleso, pro které se počítá pravděpodobnost, $\alpha_{\overrightarrow{x}}$ oblast na jednotkové sféře zastíněná tělesem N.



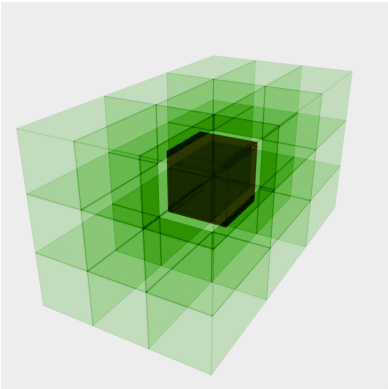
Obrázek 2.6: 2D ukázka zorného pole bodu $\overrightarrow{x}$ pokrytého uzlem N.

Tento integrál nejde přesně vypočítat, a proto se musíme uchýlit k aproximaci. Jednou z možných aproximací je rozdělit scény na části a výpočítat $\alpha_{\overrightarrow{x}}$ pouze pro jejich středy tj. budeme předpokládat, že se jedná o průměrnou hodnotu pro body v oblasti. Jelikož body blízko obalového tělesa budou podhodnocené a body daleko od tělesa naopak nadhodnocené, tak by se jejich součet 2.6 měl blížit reálné hodnotě. Tuto metriku jde lehce použít pro AABB, jelikož scéna jde rozdělit pomocí šesti polorovin AABB na 27 částí, z toho

jedna je vnitřek AABB. Hodnota se vypočte pro všech zbývajících 26 částí (viz Obrázek 2.7) a objem těchto částí je také lehce vypočten.

$$p_N \approx \frac{V(N)}{V(S)} + \frac{1}{V(S)} \sum_i V(R_i) \frac{\alpha_i}{4\pi} \tag{2.6}$$

Rovnice 2.6. R_i je oblast scény mimo aktuálně analyzované obalové těleso. α_i je plocha zabírající obalové těleso na jednotkové sféře se středem ve středu oblasti R_i.



Obrázek 2.7: Rozdělení scény obalovým tělesem AABB na 27 částí pomocí polorovin AABB. Červená krychle uvnitř je obalové těleso, pro které se pravděpodobnost počítá.

OSAH

Occlusion surface area heuristic [14] je vylepšení metriky SAH, která předpokládá, že paprsky jsou rovnoměrně rozložené v prostoru a nejsou ničím zastíněny. To je však z praktického hlediska nereálné. OSAH pracuje s viditelností trojúhelníků. Trojúhelníkům, které nejsou viditelné, je přiřazena menší důležitost při tvorbě BVH. Viditelnost jde určit například informací o průchodu hierarchie z předchozích snímků. Pro výpočet OSAH ceny slouží rovnice 2.7.

$$
\begin{aligned}
p_L &= w\frac{N_L^V}{N_L^V + N_R^V} + (1.0 - w)\frac{S_L}{S} \\
p_R &= w\frac{N_R^V}{N_L^V + N_R^V} + (1.0 - w)\frac{S_R}{S} \\
C_N &= C_t + C_i(p_L \cdot N_L + p_R \cdot N_R)
\end{aligned}
\tag{2.7}
$$

Rovnice 2.7. w je konstanta určující jak moc brát v potaz viditelnost. N_L^V a N_R^V jsou počty viditelných trojúhelníků v potomcích. p_L a p_R jsou pravděpodobnosti navštívení potomků, které se používájí v rekurzivní variantě SAH rovnice místo poměrů ploch potomka a předka. C_N je výsledná OSAH cena pro hierarchii s uzlem N jako kořenem.

2.5 Tvorba hierarchií

Potom co máme vybraný typ obalového tělesa, který chceme používat, je třeba vytvořit hierarchii. Pravděpodobnost, že náhodně vytvořená hierarchie bude špatná, je opravdu velká, proto je potřeba tuto špatnou hierarchii vylepšit přesouváním uzlů v hierarchii nebo již při vytváření hierarchie použít algoritmy, které nějak chytře a systematicky skládají hierarchii, jenž by se měla přiblížit optimálnímu řešení.

Algoritmy pro vytváření hierarchií se dají rozdělit do tří skupin podle toho, jak se dostanou k výsledné podobě hierarchie [15]:

- **Shora dolů**
  Na začátku algoritmů shora dolů (top-down) máme množinu trojúhelníků, kterou postupně dělíme na menší a menší díly, čímž vytváříme hierarchii. V každé vrstvě hierarchie se množina analyzuje a rozhodne se, jak se má rozdělit. Zástupcem této skupiny může být algoritmus od Wald [16], který rozřadí trojúhelníky na jedné ose do košů podle souřadnic jejich středů a následně hledá, jak nejlépe tyto koše rozdělit podle SAH metriky. Toto se následně rekurzivně opakuje, dokud nemáme listy s kýženým počtem trojúhelníků nebo by jejich dělení už jen zvětšovalo SAH.
- **Odspodu nahoru**
  Algoritmy odspodu nahoru (bottom-up) postupně shlukují jednotlivé trojúhelníky a následné stromy k sobě. Každým spojením dvou uzlů se stromy zvětšují dokud nevznikne jeden velký, který reprezentuje ce-

lou hierarchii. Mezi tyto algoritmy patří například LBVH [17] nebo PLOC [18].

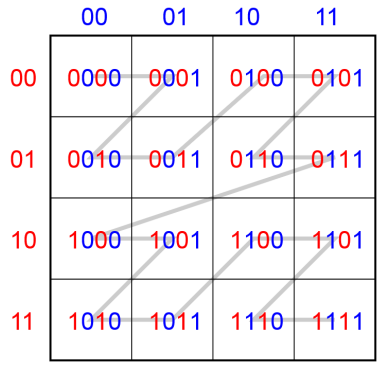
- **Inkrementální a iterativní algoritmy**
  Jako inkrementální se dají označit ty algoritmy, které postupně přidávají trojúhelníky do hierarchie a případně hierarchii pozmění, aby se minimalizovala použitá metrika. Iterativní algoritmy již mohou pracovat na hotové hierarchii a postupně ji vylepšovat. Spadá pod ně například rotace uzlů [19] ,ATRBVH [20], RBVH [21] a PRBVH [7], který je rozšiřován v této práci.

Dále si popíšeme podrobněji algoritmy využité v této práci (LBVH, RBVH, PRBVH a algoritmus optimalizace BVH pro animace založené na T-SAH metrice). Nakonec krátce zmíníme vylepšování hierarchie pomocí rotace uzlů a algoritmy PLOC a ATRBVH, které lze také využít pro optimalizaci BVH na GPU.

2.5.1 Lineární BVH - LBVH

LBVH [17] vychází z předpokladu, že chceme při tvorbě hierarchie spojovat uzly, které jsou blízko sebe. Nedává smysl vytvářet uzel hluboko v hierarchii, který by se pak táhl přes celou scénu. Všechny uzly nad ním by musely být také velké, což by zbytečně zvětšovalo SAH. Je tedy potřeba najít uzly, které jsou lokálně blízko sebe. K tomu poslouží Mortonův kód, který do sebe dokáže zakódovat lokalitu středu obalového tělesa pomocí prokládání bitů jednotlivých souřadnic XYZ. Následné seřazení trojúhelníků podle Mortonových kódů dává blízká tělesa k sobě (viz Obrázek 2.8). Pak už jen stačí slučovat okolní tělesa. Jaká sousední tělesa sloučit rozhodne délka společného prefixu jejich kódů. Na prvních místech v Mortonově kódu jsou uloženy nejvýznamější bity souřadnic a proto dává smysl slučovat nejprve uzly s co nejvíce podobným prefixem, jelikož jsou nejblíž u sebe, čímž se zmenšuje potřebné těleso.

Vytvoření hierarchie pomocí tohoto způsobu je deterministické a vcelku rychlé. Je vhodný pro implementaci na GPU. Výsledný SAH se ale může měnit v závislosti na orientaci scény a to klidně o více než 5%. I tak je to velmi dobrý způsob, jak vytvořit základní hierarchii, která jde následně jinými algoritmy optimalizovat.



Obrázek 2.8: Ukázka Mortonových kódů pro oblast 4x4. Převzato z [22].

2.5.2 Opětovné vkládání uzlů v hierarchii obalových těles - RBVH

Jednou z možností optimalizace existující hierarchie je přesunutí uzlů na jiné pozice v hierarchii a tím vylepšit výsledný SAH. Tuto techniku popsal Bittner a spol. [21]. Tato optimalizace postupně odebírá neefektivní uzly z hierarchie, pro které našla lepší pozici minimalizující SAH, a vkládá je na nově zvolenou pozici. Tento algoritmus dokáže najít kvalitní lokální minimum, ale jelikož se jedná o sekvenční algoritmus, tak je to na úkor rychlosti. Ta je až o několik řádů (klidně i tisíckrát) menší než u LBVH.

2.5.3 Paralelní opětovné vkládání uzlů v hierarchii obalových těles - PRBVH

Problém s rychlostí opětovného vkládání uzlů řeší paralelní verze RBVH pod názvem PRBVH, která byla vytvořena Bittnerem a Meisterem [7]. Ta používá čtyř-krokovou optimalizaci, která umožňuje na grafické kartě vyjmout a opětovně vložit více uzlů najednou.

Algoritmus začíná s jakoukoliv BVH hierarchií. Čtyř-kroková optimalizace se následně opakuje, dokud se nenajde lokální minimum měřené metriky (v tomto případě SAH). To se pozná tím, že se hodnota metriky nebude po několik iterací výrazně měnit.

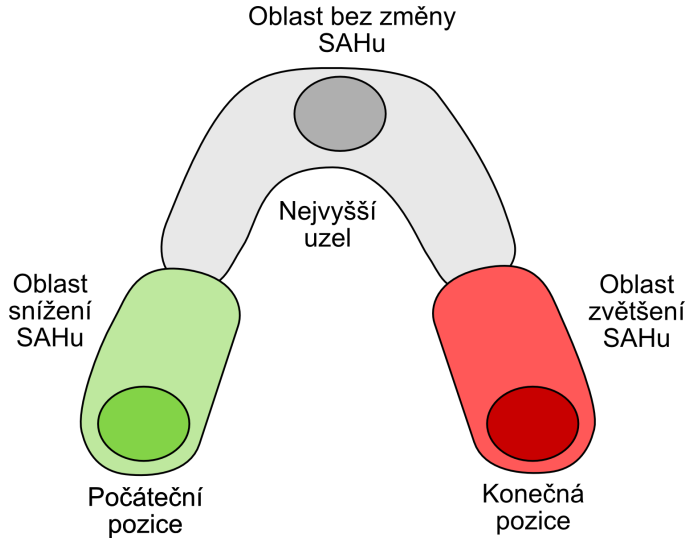
Aby se dosáhlo ještě lepších výsledků, je možné tuto techniku zkombinovat se simulovaným žíháním popsaném v navazujícím článku [23]. Simulované ží-

hání během optimalizace záměrně někdy nevybere nejlepší krok, aby umožnilo uniknout z lokálního minima, do kterého greedy algoritmy vedou. Pravděpodobnost, se kterou je vybrán horší krok, se postupem času zmenšuje, čímž se zajistí konvergence.

Postup optimalizace

Nyní popíšeme čtyři hlavní kroky tvořící jednu iteraci optimalizace. Během nich se spouští kernely jednotlivých kroků a pro každý uzel, pro který se hledá nová pozice, se vytvoří dedikované vlákno, které ho zpracuje.

- **Nacházení nových optimálních pozic uzlů - kernel findBestNode**
  Pro každý uzel, který se má optimalizovat, se začne hierarchie procházet nahoru a postupně se začne zanořovat do sousedních uzlů, kde se pokusí najít optimální místo pro znovuvložení uzlu. Aby se vyhnulo využití zásobníku, tak se hierarchie prochází v post-order pořadí. Na cestě ke konečné destinaci se sčítají rozdíly mezi SAHem před, po odebrání a po vložení. Cesta k nové destinaci obsahuje 3 oblasti (Obrázek 2.9). Ty umožní vypočíst spodní mez SAHu, pokud vložíme uzel někam do sousedního podstromu. To nám umožní zavrhnout některé cesty, které by vkládaly uzel do části hierarchie odpovídající úplně jiné části scény.



Obrázek 2.9: Oblasti na cestě uzlu na jeho novou pozici. V zelené oblasti je hodnota snižování SAHu pozitivní, v šedé oblasti nedochází ke snížení SAHu a v červené oblasti se SAH začíná zvětšovat.

Každé vlákno si drží maximální pokles hodnoty SAH a pozici, která

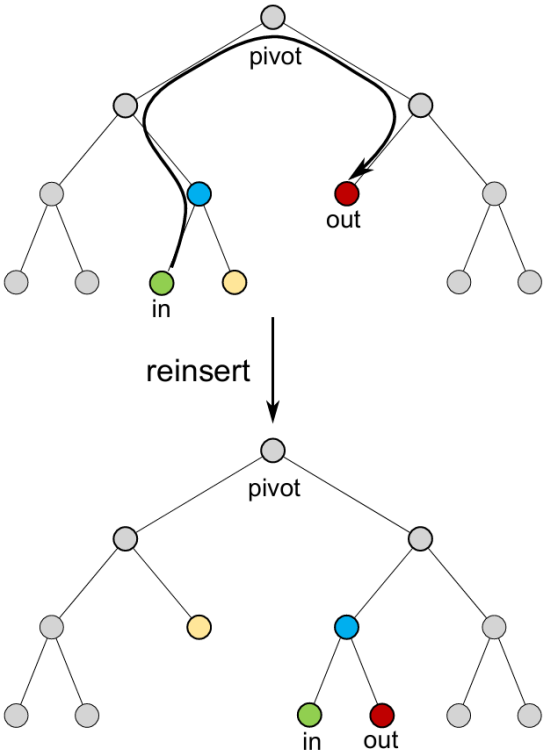
jí docílila. Tyto dvě hodnoty se uloží do bufferů pro použití v dalších kernelech.

Takto by se postupovalo pro všechny listy hierarchie. To však není potřeba, jelikož čím více uzlů chceme přesunout, tím větší je pravděpodobnost kolize, která jeden z kolidujících přesunů znemožní. Proto je výhodné brát každý N-tý uzel. Algoritmus pracuje s hodnotou mod, která určuje vzdálenost mezi uzly vybranými k optimalizaci uloženými v bufferu. Například hodnota $mod = 4$ znamená, že se budou optimalizovat uzly 0, 4, 8, atd.. V další iteraci se toto "okno"posune tak, aby se postupně na řadu dostaly všechny uzly. Když nastane situace, že zlepšení metriky je moc malé, hodnota mod se zmenšuje až na $mod = 1$.

- **Uzamknutí zámků uzlů - kernel lockNodes**
  Když je vypočtena destinace pro každý uzel, je možné, že nastane situace, kdy si dva uzly vyberou stejné místo nebo se jeden uzel chce umístit do podstromu jiného uzlu, který se chce také přesunout. Tím nastávají kolize, které je potřeba vyřešit. PRBVH používá greedy strategii, kdy se projde cesta, po které se má uzel přesunout na své nové místo a uzamykají se navštívené uzly. Pokud dvě cesty kolidují, použije se ta, která patří uzlu, jehož přesunutí způsobí větší snížení metriky.

- **Kontrola zámků uzlů - kernel checkNodes**
  Po uzamčení uzlů se projdou všechny cesty ještě jednou a zkontrolují, zda uzlu stále patří všechny zámky na jeho cestě. Pokud tomu tak není, uzel se označí jako nepoužitý tím, že se jeho redukce metriky nastaví na 0.

- **Přesunutí uzlů - kernel reinsert**
  Nyní nastává čas na změnu topologie hierarchie. Zde přesuneme každý uzel s nenulovou redukcí metriky na jeho cílové místo tak, že vyjmeme předka počátečního uzlu z hierarchie a na jeho místo umístíme sourozence počátečního uzlu. Vyjmutého předka poté umístíme na vybrané místo a uzel, který na něm byl před změnou, se použije jako nový sourozenec počátečního uzlu (viz Obrázek 2.10).



Obrázek 2.10: Ukázka přesunutí uzlu na jeho novou pozici. Zelený uzel je přesouván. Červený uzel je místo, kam se přesunovaný uzel umístí. Modrý uzel je předkem přesunovaného uzlu. Převzato z [7].

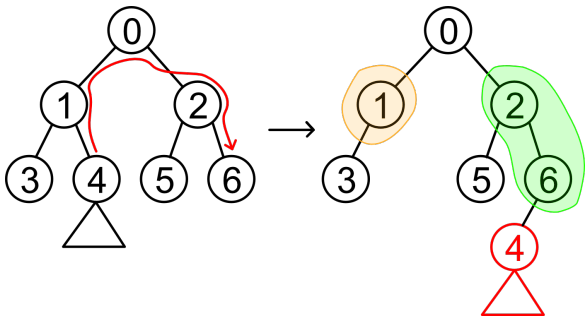
Po přesunu musíme aktualizovat hodnotu metriky. Pro vypočtení správné hodnoty SAH je potřeba provést tzv. refit, kdy aktualizujeme všechna obalová tělesa tak, aby co nejtěsněji obsahovala veškeré trojúhelníky a tělesa v jejich podstromech.

Optimalizace pomocí EPO a LCV

RBVH a PRBVH v základní podobě pracují s metrikou SAH. Teoreticky je možné tuto metriku nahradit nějakou novou, např. EPO či LCV, a optimalizovat hierarchii podle ní. Efektivní implementaci EPO a LCV ale brání problémy, které jsou rozebrány níže.

Ze čtyř kroků optimalizace nás zajímá pouze hledání nové pozice uzlu. Musíme být schopni najít v aktuální hierarchii tu nejlepší pozici pro právě zpracovávaný uzel. U SAHu se to řeší postupným slučováním obalových těles. SAH hodnota závisí pouze na tělesech, kterých se "dotkneme". To bohužel pro EPO neplatí.

Při přesunu podstromu s kořenem v uzlu 4, jako je tomu na obrázku 2.11, se hodnota EPO po přesunutí uzlu 4 změní na dvou místech. Uzly 2 a 6 (zelená oblast), které se nově staly předky podstromu, začnou obsahovat uzly podstromu 4 a tutíž jejich přesah do těchto uzlů již nemá být započten do EPO. Je proto nutné projít strom nahoru a pro každého předka spočítat EPO pro trojúhelníky v přesunutém podstromu. Dále je potřeba pro všechny uzly, které bývaly předky podstromu (uzel 1, oranžová oblast), opět spočíst hodnotu EPO se všemi trojúhelníky v podstromu. EPO oranžové oblasti přičteme a zelené oblasti odečteme. Pokud je EPO zelené oblasti větší než oranžové, tak jsme nalezli pozici, která EPO zlepší. Můžeme provést stejné zamykání uzlů jako je tomu u původní SAH verze.



Obrázek 2.11: Ukázka oblastí hierarchie, kde se mění hodnota EPO. Po přesunu uzlu dojde v zelené části ke snížení, zatímco v oranžové ke zvýšení EPO.

Tímto jsme ale vyzkoušeli pouze jednu možnou novou pozici.Musíme hledat dál. Možná existuje způsob, jak ořezat hierarchii, abychom nemuseli zkoušet všechny možné pozice uzlu (pruning), ale ten se nepodařilo nalézt. Je totiž zjevné, že i tak by tato technika byla velmi pomalá. Při každém výpočtu změny je potřeba projít přesunutý podstrom tolikrát, kolik uzlů je v zelené a oranžové oblasti. Dal by se vytvořit buffer, kam by se uzly uložily, a podstrom by se pak četl jen jednou. I to by však nestačilo. Už pro scénu s velikostí nad 200 000 trojúhelníků trvá samotný výpočet EPO docela dlouho, natož počítat ho mnohanásobně na každém vláknu.

LCV pro optimalizaci taky není vhodné, jelikož tato metrika nedává žádnou nápovědu, co je na hierarchii špatně. Víme jen, že počet projitých listů se hodně liší, ale již nevíme, v jaké části hierarchie k tomu dochází.

2.5.4 Optimalizace hierarchie pro animace

Pokud chceme použít vrhání paprsků nejen pro statické scény, ale i pro animace, musíme zajistit, že hierarchie a tělesa v ní respektují nové pozice trojúhelníků, které se pro každý snímek mohou měnit. K tomu stačí provést operaci refit.

Avšak co se stane, když se struktura scény značně změní? Objekty ve scéně se mohou přesouvat a tím pádem zvětšovat obalová tělesa uzlů. Scéna, která byla optimalizována pro jeden snímek, bude tudíž umožňovat rychlé vrhání pro tento snímek, ale pro snímky s jinou strukturu může být vrhání velmi pomalé. Nabízejí se nám tři řešení:

- Můžeme pro každý snímek spočítat novou hierarchii.
- Můžeme po změně pozic trojúhelníků znovu spustit optimalizační algoritmus. To by optimalizovalo ne nutně celou scénu, ale vylepšily by se jen oblasti se změnami. Jedná se o vylepšenou verzi první možnosti, jelikož nemusíme vždy začínat optimalizaci úplně od začátku.
- Můžeme vytvořit hierarchii, která by nebyla výborná pro vrhání v žádném časovém bodě, ale byla by pro všechny průměrně dobrá. Nalezením této průměrné hierarchie bychom ušetřili spoustu času na vytváření scény po každé změně. Na tomto principu je založený T-SAH.

T-SAH

T-SAH [24] je rozšířením běžného SAHu o časovou složku. Dá se říct, že T-SAH budeme vypočítávat jako vážený průměr SAHu pro reprezentativní snímky. Je žádané, aby reprezentativní snímky zahrnovaly co nejvíce rozdílné snímky, aby se zajistilo, že hierarchie bude optimalizována pro všechny situace, které nastanou. Dobrým způsobem je rovnoměrné rozložení snímků. SAH v čase t jde vyjádřit rovnicí 2.8.

$$C(t) = \frac{c_T}{S(t)} \sum_{N \in vnitřní} S_N(t) + \frac{c_I}{S(t)} \sum_{N \in listy} S_N(t) J_N \tag{2.8}$$

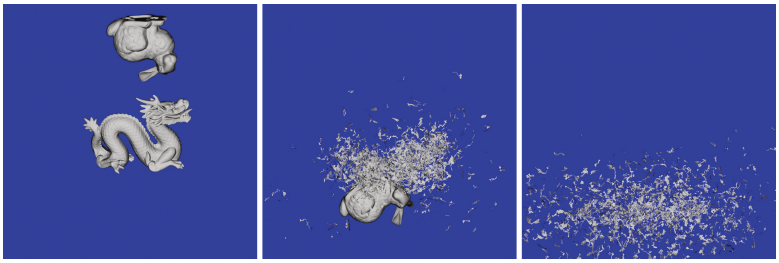
Rovnice 2.8: J_N je počet trojúhelníků v listu. C_T a C_I jsou konstanty pro traverzaci uzlu a výpočet průsečíku obalového tělesa a paprsku. $S(t)$ a $S_N(t)$ jsou plocha scény a plocha obalového tělesa uzlu N v čase t.

Když nyní spojíme výpočet SAHu v čase t a výpočet váženého průměru v závislosti na konstantě k, Dá se T-SAH vypočítat pomocí rovnice 2.9.

$$\widetilde{C} = \frac{1}{\sum_i C_i^k} \sum_i C_i^{k+1} \tag{2.9}$$

Rovnice 2.9: C_i je cena SAH pro vzorkový snímek i. Konstanta k určuje, jak moc se máme zaměřit při optimalizaci na snímky s vyšším SAHem. Pokud bude $k = 0$, tak se vypočte pouze běžný průměr, jelikož všechny časové snímky budou mít váhu 1.

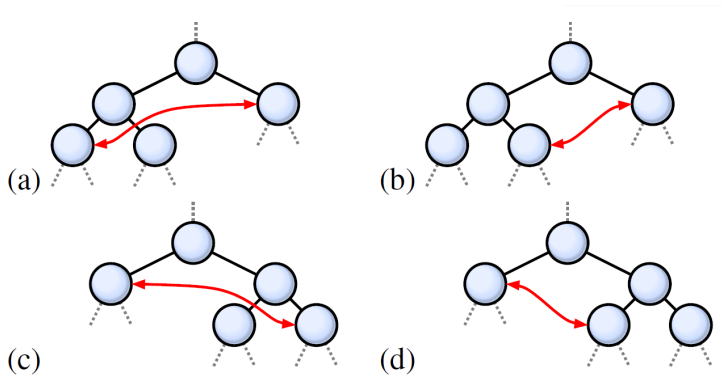
Zprůměrování scén bude fungovat pouze, když vybereme dobré reprezentativní snímky. Pokud si zvolíme snímky na začátku animace a později se scéna výrazně změní, hierarchie na to nebude vůbec připravená a může dojít k razantnímu snížení výkonu.



Obrázek 2.12: Ukázka vývoje animované scény Exploding dragon

2.5.5 Rotace uzlů stromu

Pokud nechceme používat globální přesuny uzlů, které umožňuje PRBVH, můžeme se přiklonit k lokálním úpravám hierarchie pomocí rotace uzlů [19]. Využívá se zde levých a pravých rotací podobně jako v AVL a red-black stromech. Ty rotují uzly v závislosti na hloubce uzlu. My však můžeme rotovat podle metriky SAH. Algoritmus prochází hierarchii odspodu nahoru a v každém uzlu vyzkouší všechny možné rotace 2.13, ze kterých se vybere ta, jenž nejvíce snižuje SAH. Rotace nijak neovlivní SAH uzlů ve vyšší úrovni, a proto není potřeba zdlouhavě procházet hierarchii. Avšak jeden průchod většinou není dostatečný a musí se opakovat, dokud se SAH snižuje. Tento postup může lehce vést do lokálního minima, a proto se vyplatí použít simulované žíhání, aby do něj nezapadl.



Obrázek 2.13: Ukázka možných rotací uzlů. Převzato z [19].

2.5.6 Další optimalizační algoritmy na GPU

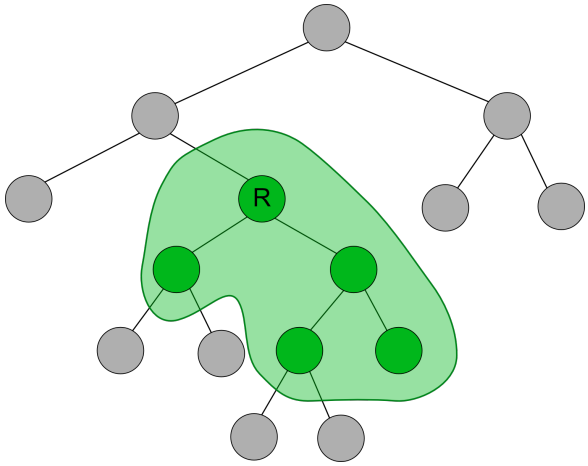
Pro efektivní paralelní implementaci tvorby hierarchie musíme být schopni jednoduše rozdělit vláknům svou práci a zároveň nemůžeme lehce dynamicky alokovat paměť. Proto je většina těchto algoritmů typu odspodu nahoru, kde "pouze"spojujeme uzly dohromady, což není paměťově náročné. Naopak algoritmy shora dolů si potřebují někde ukládat seznam trojúhelníků patřících do jednotlivých ramen hierarchie. Poměr a počet trojúhelníků v ramenech práci znesnadňuje. To by se dalo vyřešit spojovým seznamem. Další výhodou postupu odspodu nahoru je monotonní snižování počtu vláken. Když se dvě vlákna setkají v jednom uzlu, stačí jedno vlákno ukočit, zatímco druhé pokračuje dál a přebírá zodpovědnost za celý podstrom. Naopak při postupu shora dolů se potřeba vláken každou úrovní zvyšuje. Vlákna jsou však vytvořena pouze na začátku při spuštění kernelu. Vlákna by musela být nějak synchronizována a čekat, než na ně dojde řada. Tento problém nenastává v takové míře na CPU, jelikož tam se dají lehce vytvořit vlákna dynamicky. Další možnost je zpracovávat hierarchie po vrstvách, kdy každé spuštění kernelu zpracuje jednu vrstvu. Vrstvy však neobsahují dostatek uzlů na to, aby se tento přístup vyplatil. Zapínání kernelů a komunikace s CPU je totiž pomalá a grafická karta preferuje jeden velký úkol raději než spoustu malých.

Nyní si představíme dva algoritmy, které využívají postup odspodu nahoru a jsou vhodné pro využití na grafické kartě.

ATRBVH

ATRBVH [20] je inkrementální technika, která umožňuje pomocí průchodů zdola nahoru vylepšovat kvalitu hierarchie. Pracuje s tzv. treelety (viz Obrázek 2.14), což jsou podstromy hierarchie, jejichž hlubší uzly byly spojeny do větších listů. Treelety mají omezený počet uzlů a tudíž na nich jde provést lokální optimalizace. Treelety již používal Karras a Aila viz [25].

Algoritmus postupuje nahoru od listu ke kořenu. Pokud oba dva potomci uzlu byli již zpracováni, začne se optimalizovat treelet s počátkem v daném uzlu. Nejprve je ale treelet potřeba vytvořit. K tomu se prochází podstrom dolů od kořene treeletu a rozšiřuje se o největší uzly. Používá se greedy strategie. Počet uzlů v treeletu je pevně omezen.



Obrázek 2.14: Ukázka treeletu s kořenem R. Zelené uzly patří do treeletu.

Následně se listy treeletu spojují podobně jako u LBVH. Jako metrika, podle které se spojují, slouží plocha obalového tělesa, které obklopí oba dva uzly. Tyto velikosti jsou pro všechny páry treeletů předpočtené. Poté se vezme ten pár uzlů, který vytvoří nejmenší obalové těleso. Uzly jsou z množiny listů odebrány a nahrazeny spojeným uzlem. Pro ten je potřeba opět spočíst velikosti obalových těles po spojení se zbývajícími listy. Tento postup pokračuje dokud nezůstane pouze kořen. Tato část je velmi sériová a jediné, co se dá paralelizovat, je výpočet obalových těles sloučených uzlů.

Takto se postupuje ke kořenu a každým krokem se hierarchie optimalizuje. Po dokončení průchodu je možné průchod opakovat, dokud se SAH zlepšuje. Tento algoritmu je i přes sériovou optimalizaci treeletu vhodný pro grafické karty, jelikož treeletů je hodně a v jedné hladině hierarchie jsou na sobě nezávislé.

PLOC

Paralelní lokálně seřazené shlukování (Parallel Locally-Ordered Clustering) [18] je bottom-up algoritmus vytvořený pro efektivní spuštění na GPU. Podobně jako LBVH používá Mortonovy kódy. Pomocí nich prohledá okolí, snaží se pro daný uzel vybrat sousední uzel, se kterým tvoří obalové těleso s nejlepším obsahem. Vyhledávání nejlepšího uzlu je $O(n^2)$ těžké, a proto se definuje vzdálenost r určující oblast v poli seřazeném podle Mortonových kódů od indexu právě zpracovávaného uzlu. Okolí však nemusí nejlepší uzel obsahovat. Tento přibližný výsledek je ale dostačující. Původní článek popisuje použití AABB, ale algoritmus jde upravit pro jiné typy obalových těles např. SOBB [10].

Když se pro každý nespojený uzel najde nejlepší soused a tento soused považuje zpracovávaný nespojený uzel také za nejlepší, spojí se a vytvoří společného předka, který je v algoritmu nahradí. Hledání a spojování se opakuje, dokud nezůstane jen jediný uzel. Tento postup je velmi paralelizovatelný, jelikož nejvíce času zabere hledání nejlepšího uzlu a to je nezávislé na dalších uzlech.

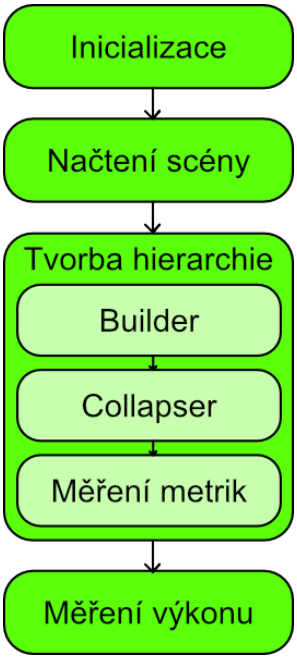
Během algoritmu je potřeba vyřešit několik drobností. Například možná existence více nejlepších uzlů, kdy se preferuje uzel s menším indexem, nebo nutnost neustále držet si množinu aktivních uzlů, které ještě nebyly s ničím spojeny. K tomu se využívá prefixová suma, aby se pro aktivní uzly našla pozice v novém poli aktivních uzlů.

2.6 NVIDIA framework

Tato práce je založena na implementaci použité v článku o PRBVH [7], která používá framework [26] vytvořený výzkumníky firmy NVIDIA. Framework využívá technologie CUDA, což omezuje jeho využití pouze na karty této značky. Framework má dva módy. Interaktivní mód, který vytvoří okno, kde se uživatel může pohybovat po scéně a manuálně ji analyzovat, a benchmark mód, který změří výkon vrhání paprsků na předem definovaných pohledech kamery.

Framework se stará o základní fungování aplikace, aby se uživatelé mohli zaměřit jen na to, co je náplní jejich výzkumu. Nyní popíši jeho hlavní části, jeho fungování a práci s ním. Jednotlivé části jsou vidět na obrázku 2.15. Veškerý výstup benchmarku je výpis logu, kde jdou najít všechny potřebné naměřené hodnoty.

- **Inicializace**
  Framework začíná základní inicializaci systémů a knihoven, na kterých je postavený, vytvoří si okno a zkontroluje přítomnost podporované grafické karty. Framework používá CUDA driver API, které je schované za abstrakční vrstvou několika tříd jako je Buffer, Kernel a CudaModule.



Obrázek 2.15: Workflow diagram fungování hlavních částí frameworku

- **Načítání scény**
  Pomocí argumentu programu se dají nastavit jeho vlastnosti. Například zda se má použít benchmark nebo interaktivní mód či umístění souboru se scénou. Pokud soubor není definovaný, použije se předdefinované umístění. Framework poté načte scénu ze souboru jako vertex a index buffer. Načítání bylo kvůli potřebám nové implementace rozšířeno o další možnosti (viz kapitola 3).

- **Tvorba hierarchie**
  Uživatel se bude primárně pohybovat v této sekci. Zde je načtená scéna předána Builderu, což je třída zodpovědná za vytvoření hierarchie. Ta použije implementovaný algoritmus a jejím výstupem je třída CudaBVH obsahující všechny buffery potřebné k vrhání paprsků. Ty jsou primárně dva. Jeden buffer pro uložení dat trojúhelníkům, které jsou uloženy již v transformovaném stavu, jelikož prošly woopifikací[2] [27]. V druhém bufferu jsou uloženy objekty popisující jednotlivé uzly a obalová tělesa jejich potomků. AABB využívá CudaBVHNode. Zde jsou uloženy informace jako indexy potomků či rozsah indexů trojúhelníků patřící tomuto uzlu, pokud je to list. Je zde také index předka, ale hlavně jsou zde všechna data potřebná k protnutí obalových těles potomků. Ta jsou uložena zde, aby se mohla načíst najednou.

[2]Transformace trojúhelníku za účelem zrychlení výpočtu průsečíku a snížení paměťové náročnosti.

Předpřipravený builder pouze vytvoří hierarchii BVH. Tu interně využije tzv. Collapser, který z ní vytvoří optimalizované buffery pro uzly a trojúhelníky. Collapser umožňuje shlukovat listy, aby obsahovaly pevně daný počet trojúhelníků nebo shlukování udělat adaptivně, aby se ještě více minimalizoval SAH. Následně může Builder spočítat potřebné metriky (především SAH).

- **Měření výkonu**
  Nakonec se vezmou buffery z CudaBVH a použijí se jako vstup pro renderovací kernel. Framework automaticky vyzkouší všechny kombinace definovaných typů paprsků, kernelů (v této implementaci vždy jen jeden) a pozic kamer. Kamery a kernely jsou předány jako argument programu. Naměřené výsledky v MRay/s zprůměruje napříč kamerami a vypíše průměrné hodnoty pro jednotlivé kombinace typů paprsků a kernelů.

  Měřené typy paprsků jsou:

  - **Primary** - Primární paprsky jsou vrhané z pozice kamery, hledají nejbližší průsečík.
  - **AO - Ambient cclusion** - Tzv. paprsky zastínění okolím jsou rovnoměrně rozložené na polokouli kolem normály z povrchu trojúhelníku. Mají omezenou vzdálenost, ve které hledají průsečík. AO paprskům stačí najít jakýkoliv průsečík, nemusí se jednat o ten nejbližší.
  - **Diffuse** - Jsou to vlastně jen AO paprsky, které hledají nejbližší průsečík a nemají nijak omezenou vzdálenost. Nejsou koherentní jako primární paprsky a ani omezené jako AO, a proto jsou nejpomalejší na výpočet.

2.7 Návrh implementace

Cílem implementace je měření, porovnání výkonu a funkčnosti obalových těles AABB, 14-DOP a SOBB na statických scénách za pomoci metod LBVH a PRBVH a na animovaných scénách s využitím T-SAH metriky optimalizované metodou PRBVH.

Abychom dokázali zpracovávat i 14-DOPy a SOBB, budeme muset rozšířit PRBVH kód. Jelikož AABB, 14-DOP ani SOBB se od sebe moc neliší, tak nejsou potřeba velké úpravy, ale kernely potřebují přístup k dalším bufferům, které budou držet informace o dalších čtyřech osách u 14-DOPu, popřípadě proměnném počtu os u SOBB.

Po projití implementace byly identifikovány následující kernely, které budou potřebovat upravit:

- **refit** - Kernel pro obepnutí obalového tělesa kolem svého obsahu.
- **setupLeaves** - Kernel, který spočte atributy obalového tělesa pro jednotlivé trojúhelníky.
- **findBestNode** - Kernel hledající optimální pozici uzlu.
- **computeCost** - Kernel pro výpočet SAHu celé hierarchie.

Drobné úpravy musí proběhnou i u Collapser třídy, kde je potřeba ukládat různá data podle typu obalového tělesa.

Následně bude aplikována metoda T-SAH na všechny tři typy obalových těles. Poté bude potřeba upravit načítání scén, které nyní nepodporuje animace. Následně bude potřeba upravit všechny kernely až na ty, které nepracují s hodnotami uloženými v obalových tělesech tak, aby podporovaly možnost zpracování více scén najednou. Jelikož při optimalizaci hierarchie se bude pozice uzlů optimalizovat pomocí metriky T-SAH, není potřeba si držet více hierarchií. Například kernel, který přesouvá uzly v hierarchii, proto zůstane stejný.

V práci chceme prokázat, že PRBVH se dá úspěšně použít pro 14-DOP i SOBB a že T-SAH metrika je použitelná nejen pro AABB, ale i pro 14-DOP a SOBB. Provedená měření by měla prokázat, že kombinace technik bude moci konkurovat optimalizaci hiearchie pro každý snímek. Předpokládáme, že vstupní snímky pro výpočet T-SAHu budou mít po načtení odpovídající trojúhelníky na stejných indexech, aby bylo možné vydedukovat pohyb trojúhelníků ve scéně. Bez tohoto předpokladu by bylo potřeba mít ve scénách jednotlivé trojúhelníky unikátně označené.

Následně umožníme pro všechny tři typy obalových těles spočíst metriky EPO, LCV a další obecné vlastnosti hierarchií.

Aby bylo možné tato rozšíření provést, bude zapotřebí zkontrolovat a případně upravit framework, aby již existující kernely PRBVH pro AABB fungovaly na novější verzi CUDA, která přináší změny, jako jsou nezávislá vlákna nebo změna práce s texturami.

Kapitola 3

Implementace

Tato kapitola popíše, jak byl framework, na kterém je aplikace vystavěna, upraven, aby se dal používat na moderních grafických kartách pro statické i animované scény. Popisuje i implementaci nových metrik a různých variant jednotlivých obalových těles.

3.1 Vylepšení frameworku

Aby byl program co nejpřizpůsobivější, byl vytvořen systém, který dokáže veškerá nastavení načíst z JSON [28] souboru, který scénu popisuje. Soubor obsahuje nastavení o tvorbě hierarchie a vykreslování, jako je třeba typ obalového tělesa a jeho varianta i informace o souboru, kde se nachází data. Veškerá nastavení jsou popsána v souboru `scene_template.json`. Framework nyní nově podporuje i načítání souborů ve fromátu PLY [29].

Zde je výčet nejdůležitějších nastavení, která se nezaobírají vstupní scénou:

- **morton60** - Zda použít 60 bitovou variantu Mortonových kódů.
- **adaptiveLeafSize** - Zda adaptivně shlukovat listy do jednoho většího pro zlepšení SAHu.
- **maxLeafSize** - Pokud se neshlukuje adaptivně, tak do jaké velikosti listu se má shlukovat.
- **mod** - Základní hodnota rozestupu uzlů při hledání nové optimální pozice.
- **use_kdop** - Zda použít 14-DOP místo AABB.
- **split_kdop** - Zda využít rozdělenou reprezentaci 14-DOPu při vrhání.
- **use_sobb** - Zda použít SOBB místo AABB.
- **sobb_normal_count** - Počet unikátních normál k-DOPu, nad kterým je postavené SOBB. Například hodnota 16 vyústí v použití 32 polorovin tvořících 16 desek. Pro hodnoty 16, 24 a 32 jsou připravené množiny, které byly použity v původním článku [10]. Jiná hodnota způsobí vygenerování náhodných normál a jejich relaxaci. První tři normály jsou vždy osy XYZ.
- **sobb_indirect_layout** - Zda použít nepřímou reprezentaci SOBB.
- **sobb_use_clipping** - Zda použít ořezávání SOBB předky, čímž se spočte přesnější SAH.
- **sobb_clipping_parent_limited** - Zda limitovat počet předků, které ořezávají SOBB.
- **sobb_clipping_parent_limit** - Maximální počet předků, které oříznou SOBB.
- **sobb_optimization_with_clipping** - Zda optimalizovat SOBB pomocí ořezávání, když je zapnuté.
- **from_scratch** - Zda má každý snímek znovu spustit optimalizaci.
- **use_reinsertion** - Zda použít PRBVH namísto LBVH.
- **use_tsah** - Zda se má použít T-SAH místo SAHu pro výpočet hierarchie.
- **K_exponent** - Exponent pro výpočet váženého průměru T-SAH.
- **computeEPO** - Zda se vypočítá metrika EPO.
- **computeLCV** - Zda se vypočítá metrika LCV.
- **persistent_warp_count** - Počet perzistentních warpů pro kernel vrhání paprsků.

Avšak největším vylepšením je možnost odklonit se od předpřipravených statických snímků, které nejde lehce sehnat kvůli nutnosti mít po načtení trojúhelníky ve všech scénách na stejných indexech. Nově jde načíst scéna přímo ve formátu GLTF 2.0 [30]. Ten podporuje jak animaci pevných těles, tak skeletální animace. Díky tomu jde lehce vytvořit jakoukoliv animaci, například v Blenderu, a hned ji použít ve frameworku. Morph targets neboli shape keys v Blenderu nejsou frameworkem podporovány. V souboru JSON jde také určit jména objektů, které se mají při načítání z GLTF ignorovat, což lze použít například pro snazší testování různých variant scény.

3.2 Přechod na novou verzi CUDA

Jelikož je implementace prováděna primárně na grafické kartě NVIDIA GTX 1660 Ti a je plánováno testování naprogramovaných algoritmů na NVIDIA RTX 4080, bylo nutné tento framework převést na novou verzi CUDA 12.6 a vyšší compute capability 7.5. To zapříčinilo, že některé předpřipravené kernely přestaly správně fungovat nebo dokonce nešly vůbec zkompilovat. Bylo tedy nutné je upravit. S příchodem CUDA 9.0 přestala být vlákna ve warpu[1] [31] plně synchronizovaná. Nyní má každé vlákno svůj počítač instrukcí. To způsobilo, že místa, kde se předpokládalo, že vlákna budou instrukce zpracovávat současně, mohou být zpracována v jiných okamžicích. Některá vlákna se mohou opozdit a dodat svá data později. S tím ostatní vlákna ale nepočítají. Je tedy nutné kernely upravit a přidat synchronizace na správná místa. V některých případech je nutné nahradit původní nesynchronní instrukce novými synchroními variantami.

Bylo tudíž nutné projít veškeré kernely a přidat synchronizace. Důležitým místem pro synchronizace je práce se sdílenou a globální pamětí, kde se na místo zapisuje a zároveň z něj jiné vlákno čte. Tato místa vyžadují použití `__syncthreads()`, která synchronizuje veškerá vlákna v bloku[2]. Toto je potřeba udělat po každém zápisu do sdílených proměnných, aby se zamezilo situaci, kdy by vlákno četlo z indexu, který jiné vlákno ještě nestihlo aktualizovat. To může nastat hlavně během paralelní redukce.

[1] Warp - Skupina 32 vláken, která provádějí stejnou instrukci současně, což umožňuje efektivní paralelní zpracování na GPU.

[2] Blok - Skupina vláken vykonávající program, které mohou využívat sdílenou paměť [31]

Další potřebnou změnou bylo přidání `__threadfence()` pro kernely, kde vlákna postupně procházela hierarchii od listů ke kořenu. `__threadfence()` zajistí, že všechny zápisy napříč všemi vlákny na zařízení jsou dokončené. Používá se totiž proměnná `termCounters`, která pro každý uzel obsahovala počet vláken, která daný uzel navštívila. Když vlákno uzel navštíví, tak atomicky přičte 1 k jeho počítadlu a vlákno uzel zpracuje, jen pokud již byl navštíven jiným vláknem. V tomto případě je hodnota počítadla již 1. Pokud je vlákno první, které uzel navštíví, svůj průchod ukončí. Tento mechanismus zajišťuje, že vlákno, které uzel zpracovává, si může být jisté, že oba (v tomto případě) dva podstromy potomků uzlu jsou již zpracované. Problém nastával, když se dvě vlákna patřící každé do jiného bloku setkala na uzlu. Bez ___threadfence() fungoval `termCounters` správně, jelikož se s ním pracovalo pomocí atomických operací, ale problém byl s ostatními hodnotami, které se ne vždy propisovaly do ostatních bloků. Přidáním této funkce se tak zajistila synchronizace dat. Toto bylo potřeba například u kernelu refitování obalového tělesa nebo výpočtu počtu uzlů v podstromu.

Poslední velkou změnou je vynucené použití `cudaTexture_object`. Nová verze CUDA již neumožňuje použití textur namapovaných na reference textur [32], ale vyžaduje použití flexibilního objektu, který drží veškeré potřebné informace o textuře, jako jsou ukazatel do paměti, datový typ, typ interpolace či chování souřadnic při přístupu k textuře. Tato změna vyžadovala vcelku jednoduchou změnu kódu, kdy se jen do kernelů začaly posílat správně připravené `cudaTexture_object` a použila se jiná funkce při čtení z textury. Implementace umožňuje přepínat mezi čtením z textury nebo přímo z globální paměti pomocí ukazatele při renderování za použití preprocesoru. Jednoduchým měřením se ukázalo, že přístup přes ukazatele je o dost neefektivnější a tudíž veškerá další měření byla prováděna s texturami [33]. Je to pravděpodobně způsobené tím, že při běžném čtení z globální paměti se využívá pouze globální vyrovnávací paměť, ale při použití textur se využije také dedikovaná vyrovnávací paměť textur a nedochází tak k tolika "cache missům". Tato skutečnost by vyžadovala hlubší analýzu, která je spíše návrhem pro možné rozšíření práce.

3.3 Rozšíření implementace - třída HybridBVH

Framework je připravený tak, že patřičné třídě Builder odpovídající typu implementovaného algoritmu předá scénu nahranou ze vstupního souboru. Tu pak interně algoritmus zpracuje, jak chce. Výsledkem jsou data ve třídě `CudaBVH`, která se předávají jako reference do Builderu. Dříve naimplementované algoritmy pracují s několika buffery představující konfiguraci "structure of arrays"[3]. Je zde buffer pro index předka, levého a pravého potomka uzlu atd..

Pro zachování předešlé implementace došlo ke zkopírování implementace `PRBVHBuilderu` a její úpravě vylepšeními ve třídě `HybridBVHBuilder`. Jelikož implementace používala pouze AABB, bylo potřeba přidat další dva čtyř-složkové buffery. Jeden pro minimální hodnoty na osách 4-7 14-DOPu (diagonály) a druhý pro maximální hodnoty. Následně bylo potřeba upravit kernely používající buffery s hodnotami AABB a zakomponovat do nich i nové buffery. Aby byla implementace co nejvíce flexibilní, byly přepracovány potřebné kernely na šablonové ___device___ inline funkce s parametrem `volumeType`. Tento parametr určuje, jaký typ obalového tělesa se bude používat. Následně byly oblasti kódu, které používaly nové buffery, obklopeny podmínkou kontrolující správné použití k-DOP a SOBB (viz ukázka 1).

```
template<int volumeType, bool useTSAH>
inline __device__ void hybrid_computeCost(
const int numberOfNodes,
const int numberOfTriangles,
float* sceneBoxAreaBuffer,
const float ct,
const float ci,
Vec4f * nodeBoxesMin,
Vec4f * nodeBoxesMax,
Vec4f* nodeDiagonalMin,
Vec4f* nodeDiagonalMax,
float* sceneCostBuffer,
const unsigned int sceneCount
){
    if(volumeType == KDOP_BVH){/* code... */}
}
```

Listing 1: Ukázka ___device___ šablonové funkce.

[3]Structure of arrays - rozložení paměti, kde jednotlivé části struktur mají své vlastní pole.

Při kompilaci framework použije NVCC kompilátor, který vygeneruje pro specifickou instanci šablony kryptické jméno. To nelze lehce získat předem, a proto je vytvořen pro každou potřebnou instanci šablony vlastní kernel, který volá šablony s předpřipravenými parametry (viz ukázka 2). Framework totiž odděluje kompilaci programu samotného a CUDA kernelů. Nejde tedy kernely přímo použít pomocí `kernel«<128, 32»>(arg1, arg2)` jako to umožňuje CUDA runtime API [33].

```
extern "C" __global__ void hybrid_computeCost_KDOP_TSAH(
    const int numberOfNodes,
    const int numberOfTriangles,
    float* sceneBoxAreaBuffer,
    const float ct,
    const float ci,
    Vec4f * nodeBoxesMin,
    Vec4f * nodeBoxesMax,
    Vec4f* nodeDiagonalMin,
    Vec4f* nodeDiagonalMax,
    float* sceneCostBuffer,
    const unsigned int sceneCount
){
    hybrid_computeCost<BVH_KDOP, true>(
        numberOfNodes,
        numberOfTriangles,
        sceneBoxAreaBuffer,
        ct,
        ci,
        nodeBoxesMin,
        nodeBoxesMax,
        nodeDiagonalMin,
        nodeDiagonalMax,
        sceneCostBuffer,
        sceneCount
    );
}
```

Listing 2: Ukázka kernelu pro volání šablony.

Přestože jsou kernely kompilovány do jednoho modulu, ze kterého se potom získají, jsou stále od sebe oddělené. Využíváme šablon a spoléháme na to, že kompilátor dokáže odebrat kontroly typu obalového tělesa. Protože jsou funkce inline, rozbalují se přímo do kernelu. Kód před optimalizací by měl tedy být stejný, jako bychom napsali každý kernel zvlášť. To je ale z hlediska přehlednosti nepředstavitelné, jelikož každá změna by se musela provést ve více kernelech a vznikalo by mnoho chyb kvůli zapomenutým změnám.

Stejně jako v předešlé implementaci byla definována třída `AABB`, vytvořil jsem třídu `KDOP`, která se stará o slučování bodů či jiných k-DOPů do jednoho a také obsahuje výpočet plochy 14-DOPu potřebné pro SAH. O to stejné jsem se pokusil při implementaci SOBB. Výpočet plochy SOBB je snadný a stačí k němu determinant matice (viz rovnice 3.3). SOBB však ve své finální reprezantaci v CudaBVH nejdou efektivně sjednotit. SOBB jsou všemožně zkosené kvádry, které nemusí mít stejné osy. My si nemůžeme být jisti, kde všude uvnitř jsou trojúhelníky rozmístěny, a proto musíme zajistit, že všechny vrcholy slučovaných SOBB jsou ve výsledném obalovém tělese. Tím se však může vytvořit zbytečně velké těleso. To by mohl být mnohem menší, pokud bychom znali k-DOPy, ze kterých se slučovaná obalová tělesa vytvořila. Z tohoto důvodu jsem se rozhodl slučování neimplementovat. Sloučené těleso je však potřebné pro výpočet SAHu kořenu hierarchie. Za normálních podmínek by toto nebyl problém, jelikož plocha obalového tělesa kořene a celé scény je stejná a tudíž pravděpodobnost na jeho navštívení je 1. Kvůli možnosti porovnávat kvalitu různých těles se však jako plocha scény používá plocha AABB. Jedná se vlastně pouze o normalizační konstantu. Rozhodl jsem se proto jako plochu kořene SOBB vždy použít plochu AABB. SAH SOBB je tedy lehce zvětšen, jelikož SOBB má vždy menší nebo stejný obsah jako AABB, ale podle jednoduchých testů je to v průměru přibližně o 0.75, což je zanedbatelné.

3.3.1 Implementace rozděleného 14-DOPu

Původní článek o 14-DOPech [9] navrhuje úpravu pro procházení hierarchie tvořenou 14-DOPu. Místo přesné kontroly kolize paprsku s 14-DOPem se nejdříve zkontroluje pouze kolize s AABB. Ta je součástí i plného výpočtu kolize 14-DOPu. Nejdříve se načte pouze první část 14-DOPu obsahující data pro AABB. Provede se výpočet průsečíku a když je AABB protnuto, načtou se zbylá data z druhého bufferu, pomocí kterých se zkontroluje zbytek os 14-DOPu. Toto lze učinit, jelikož víme, že AABB je rovno nebo větší než 14-DOP a vždy ho plně obsahuje. Proto můžeme využít AABB jako další obalové tělosо kolem 14-DOPu. Tato optimalizace spoléhá na to, že se ušetří čas za zbytečné načítání dat a za výpočty zbylých os, které nebudou využity, když ani AABB není protnuto.

Tato metoda má i své nevýhody. Jelikož některá vlákna využijí dřívější ukončení testu a některá ne, dochází k divergenci, která snižuje výkon. Naštěstí se kontroluje levý a pak pravý potomek uzlu, tudíž je zde nějaká šance, že dříve ukončená vlákna poběží současně s těmi, která ještě kontrolují první 14-DOP. To je možné, jelikož kontrola protnutí AABB a 14-DOPu využívá skoro totožné instrukce, jen v jiném počtu.

Třída `HybridBVHCollapser` vytvoří z jednotlivých bufferů ve formátu "structure of arrays", které se používaly při tvoření hierarchie, optimalizovanou verzi uzlů s kompaktním rozložením. Bylo tak zvoleno, aby se uzly mohly načítat pouze z jednoho bufferu (dva při rozdělení 14-DOPu) a dobře se tak používala vyrovnávací paměť.

3.3.2 Varianty uložení SOBB

SOBB je tvořené šesti hodnotami určujícími minimum a maximum na osách polorovin. K tomu je ještě potřeba definovat poloroviny pro tyto hodnoty. Existují dvě varianty jak uložit potřebná data:

- **Nepřímé rozložení** - Uzel hierarchie obsahuje 6 hodnot rozsahu a následně do patnácti bitů zakóduje index tří normál. Ty musí být během zpracování uzlu načteny, což zvětšuje nároky na rychlost paměti. Pokud by bylo potřeba testovat SOBB postavené na více než 64-DOPu, tak je tuto část potřeba upravit.
- **Přímé rozložení** - Normála a její rozsah jsou zakódovány do čtyřčíselného vektoru (viz 3.1) Pro zpracování je potřeba dekódovat reprezentaci a spočíst maximální hodnotu. Může se zdát, že bude potřeba použít odmocnina pro zjištění délky přeškálované normály, ale to není potřeba. Spočítat průsečík paprsku jde i bez toho.

$$\begin{aligned} \rho &= \frac{1}{\max(p_{\max} - p_{\min}, \epsilon)} \\ s &= \rho \cdot \begin{bmatrix} n_x & n_y & n_z & p_{\min} \end{bmatrix}^T \end{aligned} \tag{3.1}$$

Rovnice 3.1. $p_{\max}$ a $p_{\min}$ určují rozsah na ose páru polorovin. ρ je škálovací konstanta. n je normála poloroviny s hodnotou $p_{\max}$.

Při výpočtu průsečíku používáme naškálovanou normálu $\rho \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^T$, na které je rovina p_{min} reprezentována jako rovina ρp_{min}. Díky škálování víme, že vzdálenost 1 na této ose je vlastně vzdálenost $p_{max} - p_{min}$ na ose jednotkové. Tudiž rovina p_{max} je na naškálované normále reprezentována rovinou ve vzdálenosti $p_{min} + 1$.

3.3.3 Generování normál SOBB

SOBB potřebuje k rychlému vybírání orientace kvádru předdefinovanou množinu normál, která tvoří k-DOP, ze kterého se následně vyberou tři nejlepší normály. Teoreticky můžeme zvolit jakoukoliv množinu normál, která obsahuje alespoň jednu lineárně nezávislou trojici normál. Je však výhodné použít normály co nejvíce rovnoměrně rozložené. To zajistí, že existuje dostatek možných orientací SOBB a tudíž se zvýší šance na nalezení nejoptimálnější orientace, která minimalizuje SAH. Má implementace umožňuje zvolit počet normál s tím, že prvních 7 se shoduje s normálami 14-DOPu. Ty jsou zvoleny, jelikož jejich rozložení je již velmi rovnoměrné. Pokud se mají použít další normály, jsou náhodně vygenerovány. Posléze jsou v několika iteracích všechny normály (kromě prvních tří odpovídajících osám XYZ) posunovány po jednotkové sféře. Tím se zajistí, že normály neukazují stejným směrem a že jsou rovnoměrně rozložené. Je potřeba brát na vědomí, že větší počet normál lineárně zvětšuje paměťové nároky na tvorbu. Konečná reprezentace je na počtu normál nezávislá, jelikož se vždy vyberou jen tři.

Má implementace omezuje počet normál na 32 kvůli optimalizaci paměti, ale toto omezení jde lehce obejít upravením nepřímé reprezentace (tam jsou indexy zakódovány do 15 bitů, což by pro více normál nestačilo) nebo použítím přímé reprezentace, která není nijak omezená.

Pokud je zvoleno 16, 24 nebo 32 normál, použije se předdefinovaná množina normál, která byla použita v SOBB článku [10].

3.3.4 Výběr normál SOBB

Článek [10] navrhuje několik strategií jak redukovat k-DOP na SOBB. Liší se především svou asymptotickou složitostí. Čím více výběr zrychlíme, tím se zvyšuje pravděpodobnost, že nevybereme optimální rozložení SOBB. Avšak normál se používá dost a jsou rovnoměrně rozložené, tak i když vybereme neoptimální normály, určitě se budou blížit optimu a budou mít lepší SAH než AABB.

- **Strategie** $O(k^3)$ - S touto primitivní stategií se projdou veškeré trojice normál a vybere se ta, která vytvoří SOBB s nejmenší plochou (Rovnice 3.3).
- **Strategie** $O(k^2)$ - První normálu můžeme vybrat tak, že použijeme tu s nejužším prostorem mezi polorovinami na této normále. Zbylé dvě normály vybereme naivním brute-force projitím všech kombinací zbylých normál. Ušetříme tak jeden cyklus oproti první strategii.
- **Strategie** $O(k)$ - Tato strategie navazuje na první krok druhé strategie a po výběru první normály vybere druhou normálu takovou, že obsah rovnoběžníku tvořeného těmito normálami je minimální. To zajišťuje, že je SOBB co nejvíce zkosené, čímž je větší šance na menší plochu SOBB. Poslední normála se vybere opět taková, že dohromady mají vybrané normály co nejmenší plochu.

$$\text{Area} = 2 \cdot (|e_1 \times e_2| + |e_1 \times e_3| + |e_2 \times e_3|) \tag{3.2}$$

Rovnice 3.2. e_x vektor ve stejném směru jako je odpovídající hrana. Jeho délka odpovídá délce hrany

$$\text{Area} = 2 * \left(\frac{d_0 d_1 + d_1 d_2 + d_1 d_3}{\left| \det \begin{bmatrix} n_1 & n_2 & n_3 \end{bmatrix} \right|} \right) \tag{3.3}$$

Rovnice 3.3. n_x normála desky. d_x tlouška desky

Plochu SOBB lze spočítat dvěma způsoby (Rovnice 3.2 a 3.3) v závislosti na reprezentaci SOBB. Tato implementace používá vždy rovnici 3.3, jelikož jsou v bufferech uložené vždy rozsahy na normále, které odpovídají hodnotě d_x.

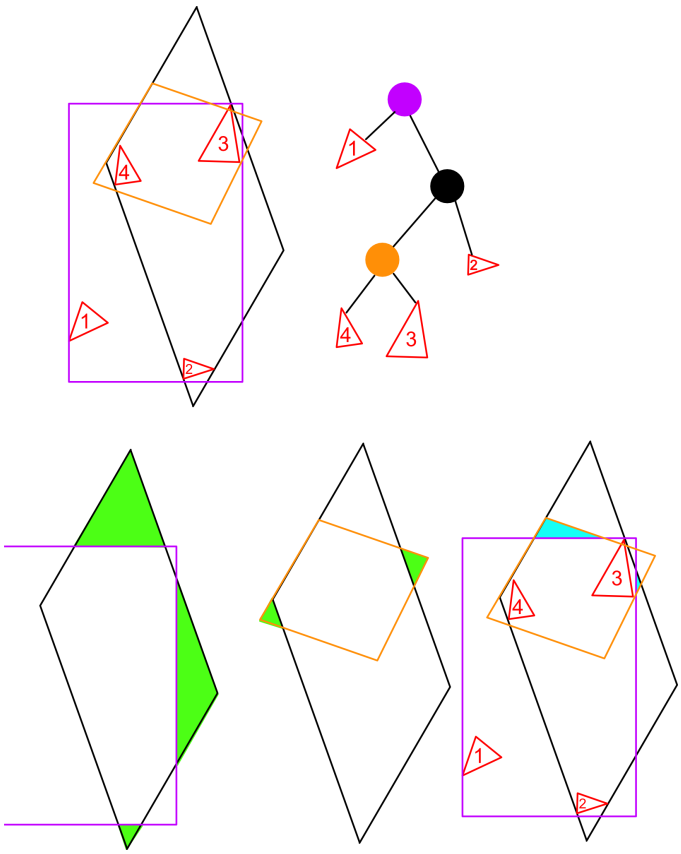
3.3.5 Správný výpočet plochy pro SOBB

Tělesa AABB a 14-DOP jsou definované vždy stejnou množinou os. Nemůže nastat situace, že by potomek vystupoval ven z obalového tělesa předka. Na každé ose je totiž těleso maximálně těsné. To však neplatí pro všechny typy těles. Pro OBB i SOBB existují konfigurace, kdy konfigurace tělesa optimalizující SAH vystupuje ven z tělesa předka (viz Obrázek 3.1). To je způsobeno tím, že SOBB nesdílejí osy, na kterých jsou maximálně těsné. Situaci popíšu na SOBB.

SOBB jsou postaveny nad k-DOPem. Když se z něj vybírají nejlepší osy pro SOBB, jsou nutně některé osy zahozeny a jejich limity ignorovány. Směrem těchto os vznikají rohy SOBB. Tyto oblasti byly dříve "ořezány" k-DOPem, a proto nemohly vystupovat z předka. Z hlediska vrhání paprsků to není problém. Je očividné, že SOBB jsou méně těsné než k-DOP, na kterém byly postaveny.

Když se podíváme na obě zmíněné rovnice pro výpočet SAHu (Rovnice 2.1 a 2.2), je vidět, že ani jedna nepočítá s možností, že by obalové těleso potomka mohlo vystupovat ven z předka. SAH použije plochu k určení pravděpodobnosti, že potomek bude protnut. Pokud by potomek měl identické obalové těleso jako předek, byla by pravděpodobnost 1. Když část předka vystupuje ven, zvětšuje se tím zbytečně SAH. Pokud je tento výstupek protnut paprskem, tak se tento paprsek přesto nedostane přes protnutí s předkem. Je tedy zbytečné tuto část započítávat do SAHu. Teoreticky je i možné, že při použití rovnice 2.1 by mohla pravděpodobnost vstupu do daného potomka $\frac{\mathrm{SA}(N_L)}{\mathrm{SA}(N)}$ být větší než 1, což samozřejmě nedává smysl. Tato sitauce může nastat při použití strategií $O(n)$ nebo $O(n^2)$, které mohou vybrat neoptimální velikost plochy.

Může nastat i situace, kdy první předek obsahuje celého potomka, ale potomek je již v části předka, která vystupuje z prapředka. Tyto části by se také neměly do SAHu počítat. Je proto potřeba ořezat SAH pomocí všech předků, než se vypočte plocha použitá v SAH rovnici.



Obrázek 3.1: První řádek vizualizace SOBB a jejich hierarchie. Druhý řádek zobrazuje oblasti vyčnívající z předků. Zelená oblast vyčnívá z prvního předka uzlu. Tyrkysová oblast vyčnívá až z druhého předka a má být také oříznuta.

Je tedy potřeba definovat novou rovnici pro SAH 3.4, která bere v potaz oblasti mimo předky a zbytečně nebude započítávat oblasti, které nemohou být nikdy protnuty.

$$\mathrm{clip}(N) = \begin{cases} N, & \text{kořen hierarchie,} \\ N \cap \mathrm{clip}(Np), & \text{jiný uzel} \end{cases}$$

$$c(N) = \begin{cases} c_T + \frac{\mathrm{SA}(\mathrm{clip}(N_L))}{\mathrm{SA}(\mathrm{clip}(N))} c(N_L) + \frac{SA(\mathrm{clip}(N_R))}{\mathrm{SA}(\mathrm{clip}(N))} c(N_R), & \text{vnitřní uzel,} \\ c_I |N|, & \text{list.} \end{cases} \tag{3.4}$$

Rovnice 3.4. clip funkce výpočtu skutečné oblasti uzlu. N_p je předek uzlu N, N_R a N_L jsou potomci uzlu N. Funkce SA vypočte plochu vstupní množiny.

Tento výpočet je početně náročný, jelikož je potřeba nejprve SOBB definované třemi deskami obklopit k-DOPem. Proto je potřeba spočítat pozice osmi vrcholů SOBB a následně je promítnout na všechny osy k-DOPu. Poté se musí procházet hierarchie směrem ke kořenu a zároveň k-DOP zmenšovat, pokud některý z předků omezuje rozsah na ose k-DOPu. Nakonec je potřeba spočíst plochu tohoto omezeného k-DOPu. Jelikož se již nejedná o 14-DOP, nemůžeme použít stejný postup a musíme se uchýlit k pomalejšímu face clipping algoritmu popsanému zde [9].

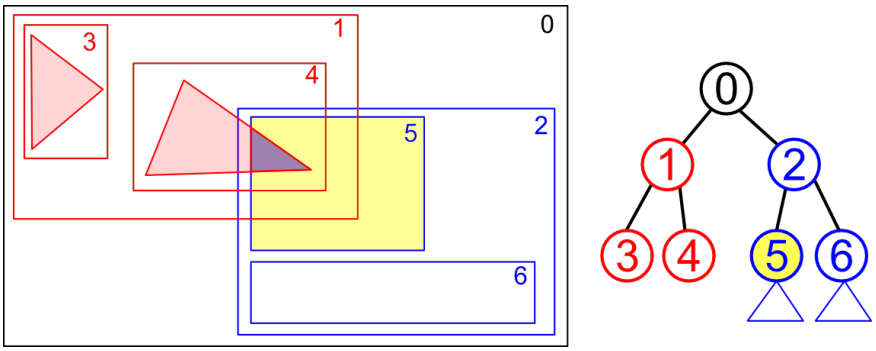
Face clipping algoritmus pro každou možnou stěnu k-DOPu ořízne rovinu, na které stěna teoreticky leží, pomocí páru základních XYZ os, které jsou společně s analyzovanou osou lineárně nezávislé. To vytvoří čtyřhran, který je následně ořezáván dalšími rovinami. Nakonec se sečte suma těchto stran. Je možné, že stěna ve výsledném k-DOPu není a v tom případě bude postupně oříznuta veškerá plocha definovaná původním čtyřstěnem.

3.3.6 Náročnost výpočtu EPO

Pokud máme scénu s N trojúhelníky a předpokládáme jeden trojúhelník v listu, tak výsledná hierarchie bude mít $2N - 1$ uzlů. Naivní metoda výpočtu by provedla $(2N - 1) * N$ kontrol zda trojúhelník patří do podstromu uzlu a pokud ne, tak jaká jeho část se nachází v obalovém tělese uzlu. To by bylo hodně neefektivní nejen z hlediska výpočtu průniku trojúhelníku a obalového tělesa, ale špatně by se i zjišťovalo, zda trojúhelník patří do podstromu uzlu. Většina uzlů se totiž vůbec neprotne, jelikož se nachází v úplně jiné části scény.

Lepším přístupem je provést průchod stromem a ořezávat uzly, které nemůžeme nikdy protnout. Začněme v uzlu 5 (Obrázek 3.2). Víme, že hledáme trojúhelníky, které nepatří do podstromu uzlu, tudíž je zbytečné prohledávat podstrom tohoto uzlu. Posuneme se tedy nahoru do uzlu 2. Ten je vnitřní a neobsahuje žádné trojůhelníky. Víme, že jedním z jeho potomků je uzel 5, odkud jsme přišli. Vydáme se tedy na druhou strana na uzel 6. Ten určitě nepatří do podstromu uzlu 5 a tudíž všechny trojúhelníky, které se v něm nachází, jsou kandidáti na započtení do EPO.

Mohli bychom projít celý podstrom uzlu 6 a otestovat veškeré trojúhelníky v něm, ale je výhodnější nedříve zkontrolovat, že uzly 5 a 6 mají průnik. Pokud by nekolidovaly, tak se určitě nemůže část trojúhelníku spadajícího pod uzel 6 nacházet uvnitř uzlu 5. Tím jsme ušetřili spoustu práce. Čím výše v hierarchi dokážeme provést tento ořez, tím více ušetříme. Pokračujeme dál do uzlu 0, kde stejnou kontrolu provedeme pro uzel 1. Zde již je nějaký překryv a proto provedeme kontrolu rekurzivně nad potomky uzlu 1, až dokud nenarazíme na listy s trojúhelníky, které zkontrolujeme. Uzel 3 bude taky ořezán, ale uzel 4 bude navštíven.



Obrázek 3.2: Ukázka hierarchie obalových těles pro výpočet metriky EPO. Trojúhelníky pod uzly 5 a 6 značí, že jsou pouze kořenem podstromu a ne listy.

Tato optimalizace je kritická pro spočtení EPO v přiměřeném čase na větších scénách. Během experimentu bez využití optimalizace nebylo EPO ani po 10 minutách spočteno. Bez optimalizace je asymptotická složitost výpočtu $O(n^2)$, jelikož každý z $2N - 1$ uzlů musí spočíst průnik s N trojúhelníky.

Kontrola průniku obalových těles

Optimalizace pomocí ořezávání spoléhá na to, že u dvou obalových těles lze snadno určit, zda se protínají. Implementace pracuje s tělesy, která jsou reprezentována rozsahem na normálách (průnik polorovin). Pro AABB a 14-DOP je jednoduché provést kontrolu průniku pomocí Separating axis theorem [34]. Stačí porovnat maxima a minima na jednotlivých normálách. Pokud existuje alespoň jedna normála, na které se poloroviny neprotnou, tělesa se neprotínají.

Problém nastává u SOBB. Dvě SOBB tělesa nemusí sdílet ani jednu normálu, tudíž jednoduché porovnávání hodnot na osách nebude fungovat. Výpočet totiž provádíme až po převedení hierarchie do CudaBVH a tudíž již nemáme přístup k k-DOPu, na kterém je SOBB postaveno. Je tedy zapotřebí spočítat vrcholy jednoho z SOBB a tyto body promítnout na osy druhého SOBB, kde vytvoří rozsah, který jde porovnat s rozsahem druhého SOBB stejně jako u AABB. Každý vrchol leží na průniku tří rovin. Abychom ho našli, je potřeba vyřešit soustavu rovnic o třech neznámých.

$$
\begin{aligned}
n_x x + n_y y + n_z z &= d \\
A &= \begin{bmatrix} n_{1x} & n_{2x} & n_{3x} \\ n_{1y} & n_{2y} & n_{3y} \\ n_{1z} & n_{2z} & n_{3z} \end{bmatrix} \\
p &= A^{-1} \cdot \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}
\end{aligned}
\tag{3.5}
$$

Rovnice 3.5. p jsou souřadnice vrcholu vzniklého průnikem rovin definovaných normálami n a vzdálenostmi d od počátku. Normála a vzdálenost vychází z běžné definice roviny.

V rovnici 3.5 je vidět, že je potřebné, aby byly normály lineárně nezávislé, jinak nebude existovat inverze matice. Vypočítat vrcholy pro každý uzel, se kterým se počítá průnik, je náročné, a proto se vyplatí spočítat vrcholy pro právě zpracovávané SOBB. Ty zůstanou po celou dobu výpočtu stejné a budou se pouze promítat na normály procházených SOBB uzlů.

Průnik obalového tělesa a trojúhelníku

Pro výpočet plochy uvnitř tělesa byl trojúhelník postupně ořezáván polorovinami. Při řezu 2D konvexního útvaru polorovinou je vždy výsledek (pokud existuje) opět konvexní a má maximálně o jeden vrchol navíc. Obalové těleso s nejvíce polorovinami v této implementaci je SOBB s 64 polorovinami. Tudíž může vzniknout maximálně objekt s 67 vrcholy. Při výpočtu na grafické kartě je potřeba někam vrcholy ukládat a jelikož si vlákno nemůže dynamicky alokovat paměť, je potřeba mít předem připravený dostatečný prostor na výsledky. Můžeme použít buď předpřipravený buffer nebo lokální pole s 67 položkami (celkem 804 bytů pro každé vlákno). To je snadnější varianta, ale je možné, že verze s bufferem by umožnila práci více vláken najednou za cenu větší spotřeby paměti. Tato možnost nebyla neimplementována, jelikož už tak SOBB spotřebovává dost paměti a její nároky se ještě zvýší při použití T-SAH.

Měření LCV

Článek [12] měřil LCV pomocí paprsků zastínění okolím (AO[4]). Nepoužívali primární či stínové paprsky, jelikož je považovali za příliš závislé na vybrané pozici kamery. Tato práce generování vylepšuje tím, že paprsky se generují rovnoměrně na ploše všech trojúhelníků. Při generování paprsků postupovalo takto:

1. Nejprve se pro každý trojúhelník spočte jeho plochu a uloží se do bufferu.
2. Následně se spočte suffixová suma[5] pro hodnoty v bufferu. Poslední položka obsahuje součet všech ploch trojúhelníků.
3. Vygeneruje se náhodné číslo od 0 do součtu ploch všech trojúhelníků. Následně se vybere ten trojúhelník, jehož suffixová suma je větší než vygenerované číslo a jeho index je minimální.
4. Pomocí dvou dalších náhodných čísel od 0 do 1 se spočtou barycentrické souřadnice na trojúhelníku, čímž se určí počátek paprsku.
5. Následně se postupuje stejně jako při běžném generování AO paprsků a vygeneruje se náhodný směr paprsku na polokouli kolem normály trojúhelníku. Předpokládá se, že normály ukazují ven z objektu.

[4] AO - ambient oclusion

[5] Suffiová suma - součet všech položek s menším indexem včetně aktuální položky = $v(i) = \sum_{x=0}^{i} v(x)$

Takto generované paprsky jsou naprosto nezávislé na pohledu kamery a tudíž by LCV mělo dávat ještě přesnější výsledky. Problém ale může být v tom, že následné měření neprobíhá ze všech míst ve scéně. Předpoklá se ale, že většina geometrie scény je viditelná a tudíž by i vzorky na ní měly být brány v potaz.

3.3.7 Implementace T-SAH

Jelikož T-SAH pracuje s více vstupními scénami, bylo potřeba upravit načítání. Dříve šlo pouze zadat cestu k 3D souboru, který se má použít, ale jelikož je potřeba jich nyní načíst více, bylo zapotřebí vytvořit něco sofistikovanějšího.

Do projektu byla přidána knihovna pro načítání JSON souborů, které budou držet seznam všech souborů s geometrií scén a indexy těch, které se mají použít jako vzorky pro T-SAH. Byla k tomu použita open-source knihovna `nlohmann/json` [28].

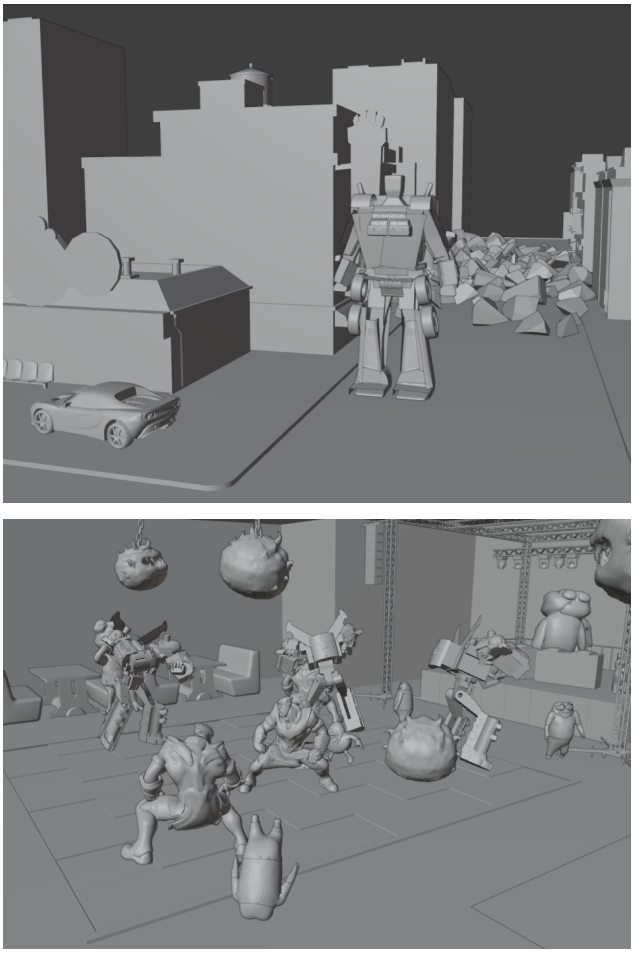
JSON se přečetl a veškerá data (vertex/index buffer) se vložila do stejných bufferů jako předtím, ale scény se poskládaly za sebe. Jelikož víme, kolik vrcholů a trojúhelníků je v každé scéně, můžeme lehce indexovat stejný trojúhelník ve více scénách jen pomocí posunu indexu. Abychom nemuseli upravovat kernely starající se o vytváření výsledné `CudaBVH` třídy, tak jako první scéna byla do bufferů dána ta, která se má nakonec vykreslovat. Je však možné, že se tato scéna nemá použít při výpočtu T-SAHu a s tím se musí počítat, když se předávají data kernelům.

Všechny kernely, které nějak pracují s vrcholy, trojúhelníky nebo obalovými tělesy, bylo potřeba přizpůsobit pro práci s více scénami. Pomocí argumentu `sceneCount` je předáván do kernelu počet scén a ty používají cykly pro zpracování všech dat.

Délka cyklu není pro kernel předem známa a to způsobuje drobnou neefektivitu kernelů. Jelikož nejvíce práce se podle původního článku odvede v kernelu pro hledání optimální pozice uzlu, byla pro tento kernel implementována T-SAH varianta odděleně, aby neovlivňovala výsledky měření bez T-SAHu.

3.3.8 Nově vytvořené scény

Součástí zadání bylo vytvořit nové scény, na kterých by se daly techniky stavby BVH testovat. Byly proto vytvořeny dvě scény (viz Obrázek 3.3). Jedna reprezentující interiér s velkým množstvím animovaných postav a druhá s větší rozlohou, jejíž topologie se značně změní kvůli padajícímu meteoru a transformaci auta. Obě scény byly vytvořeny v programu Blender. Většina postav a modelů je stažena z internetu a jsou volně použitelné. Mým výtvorem je pouze model transformera, meteor a stěny klubu.



Obrázek 3.3: Náhled dvou nově vytvořených scén.

Kapitola 4

Měření

V této kapitole se budeme zabývat měřením hodnot pro implementovaná obalová tělesa. Měření budou porovnána s předchozími pracemi. Analýza bude probíhat pomocí připravených metrik, které by měly určit, která obalová tělesa mají potenciál nahradit AABB.

4.1 Metody měření

Pro měření výsledků byl použit vestavěný benchmark z původního frameworku, který po drobných úpravách umožňuje změřit rychlost vrhání primárních paprsků, sekundárních paprsků s omezenou vzdáleností (AO) a sekundárních paprsků (diffuse), hodnotu SAH pro vykreslovanou scénu, čas vytváření scény a hodnoty metrik EPO a LCV. Výsledky vrhání pro scénu jsou průměrovány napříč definovanými pohledy kamery, aby se minimalizovala lokalita měření. Kód byl rozšířen o možnost aktualizovat geometrii scény. Tím se vyhnulo nutnosti pro každý snímek optimalizovat scénu od začátku. Je umožněno vytvořit nový snímek pomocí interpolace z klíčových snímků nebo si zvolit jakýkoliv čas a vygenerovat scénu z GLTF dat. Následuje refit, aby hierarchie obalových těles odpovídala novým polohám trojúhelníků. Je možné i znovu spustit optimalizaci, aby se dala změřit situace, kdy se optimalizuje každý snímek zvlášť.

Bohužel kód výchozího článku [7] obsahoval pouze 4 scény, z toho pouze jedna byla v článku změřena. Ostatní scény, které byly použity k měření výsledků ve článcích o PRBVH, k-DOPech, SOBB a T-SAH, musely být najity na internetu. Nepovedlo se dohledat všechny scény z původních článků.

Některé byly získány od vedoucího práce a některé staženy z https://casual-effects.com/data/ [35]. Tento zdroj scén byl referován ve článku o k-DOPu [9]. Další statické scény byly nalezeny na internetu. Animované scény pro T-SAH rozdělené na jednotlivé snímky se nachází na https://www.sci.utah.edu/ wald/animrep/ [36] a na https://gamma.cs.unc.edu/DYNAMICB/ [37]

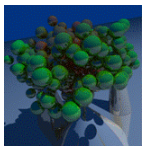
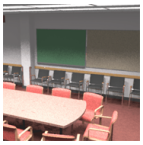
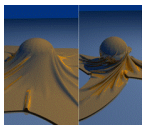
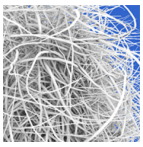
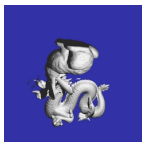
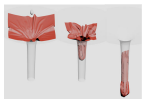
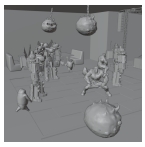
Některé scény jsem měl tedy vícekrát a nebyl jsem si jist, kterou přesně mám použít, jelikož se od sebe jen lehce lišily velikostí a některé dokonce pouze natočením. Natočení přitom může hrát velkou roli při výpočtu LBVH. Při výpočtu Mortonova kódu se osy používají v určitém pořadí a otočení modelu toto pořadí naruší, čímž se uzly srovnají jinak a výsledná hierarchie se pak liší. Podle mých pokusů, kdy jsem vyzkoušel všechny kombinace rotací a inverzí os u scény "conference room", jsem naměřil rozdíl v SAH hodnotě až 6%. Pro měření výsledků je tedy důležité mít stejné vstupní scény.

I kdyby byly všechny scény správné, naměřené výsledky vrhání budou odlišné, jelikož se používá jiná grafická karta a pozice kamery, ze kterých se měřilo, se neshodují. Pár měřených scén u sebe mělo soubor s definovanou orientací kamery, ale zda ho ve článku používali, není známo. V této práci bylo rozhodnuto je použít.

Pro všechna měření byla použita adaptivní velikost listu. Listy se spojují do jednoho většího listu pouze, pokud se tím vylepší SAH. SAH pro SOBB byl měřen bez ořezávání částí mimo potomky. Naměřené časy jsou pouze časy, po které běžely kernely. Práce na CPU do nich není nijak zahrnuta.

Měření byla zautomatizována pomocí Python skriptu, který nechal vypisovat všechna měření do logu, který byl následně analyzován. Výsledky měření byly poté uloženy jako CSV pro další zpracování. Script má připravené testovací sady pro měřené konfigurace. Umožňuje lehce vyměnit scény, které mají být otestovány.

Pokud není uvedeno jinak, probíhala měření na Windows 11 s NVIDIA RTX 4080 16B a Intel Xeon W-2275 3.3 GHz s 128 GB RAM. Animace jsou vždy měřeny 25 snímky a pokud se používá T-SAH, je využilo pět rovnoměrně rozložených vzorků. Před měřením každého snímku proběhne refit.

Statické scény		Animované scény	
	Bistro interior 1043k	24-cell 123k 128 snímků	
	Buddha 1087k	Ball 16 146k 76 snímků	
	Conference 282k	Breaking lion 1604k 44 snímků	
	Fairy forest 174k	Cloth ball 92k 93 snímků	
	Hairball 2880k	Exploding dragon 253k 15 snímků	
	Manuscript 4305k	Fairy forest 174k 20 snímků	
	Powerplant 12759k	Funnel 18k 500 snímků	
	San Miguel 7880k	Hand 16k 43 snímků	
	Sibenik 80k	Meme party 1041k GLTF	
	Vokselia spawn 1875k	Transformer 382k GLTF	

Tabulka 4.1: Přehled použitých scén. GLTF scény si mohou vygenerovat jakýkoliv počet snímků.

Pro obalová tělesa se používaly tyto konstanty pro výpočet SAH, které byly převzaty z původních článků:

- **AABB** - $C_t = 3$ $C_i = 2$
- **14-DOP** - $C_t = 4.5$ $C_i = 2$
- **SOBB** - $C_t = 4.5$ $C_i = 2$

4.1.1 Kernel vrhání paprsků

Kernel `kdop_tesla_persistent_while_while` vznikl jako rozšíření kernelu `tesla_persistent_while_while`, který byl součástí PRBVH repozitáře. Byly k němu přidány parametry potřebné pro k-DOP rozdělený do dvou bufferů a šablonové parametry, aby se daly vygenerovat funkce odpovídající aktuálnímu nastavení.

Kernel používá tzv. perzistentní vlákna. Většině běžných kernelů se nechá vytvořit tolik vláken, kolik je potřebných jednotek ke zpracování (např. jedno vlákno na každý pixel). Takto může vzniknout hodně vláken. CUDA musí rozhodnout, kdy bude jakému vláknu přiřazeno fyzické jádro, aby mohlo běžet. Tato koordinace stojí nějaký výkon. Může také nastat situace, kdy potřebujeme vytvořit více vláken než GPU podporuje. Abychom se těmto problémům vyhnuli, můžeme použít perzistentní vlákna. Ta běží celou dobu programu a vždy, když všechna vlákna v bloku zpracují svou položku, si společně vezmou další položku z fronty. To zajišťuje i rovnoměrné rozložení práce, jelikož pokud se jeden blok zdrží, ostatní mohou převzít jeho položky. Prakticky jediná cena za perzistentní vlákna je drobná práce s indexy a atomická proměnná ukazující na příští položku, která má být zpracována.

Teoreticky bychom mohli spustit tolik perzistentních vláken, kolik je položek ke zpracování, ale tím by tato technika ztrácela smysl. Pokud jich pustíme více, některá vlákna nebudou mít co na práci. Musíme tedy spustit méně vláken než je položek. Ale kolik?

Každá NVIDIA grafická karta s podporou CUDA obsahuje přesně daný počet fyzických jader. Ty se shlukují do "streaming multiprocesssorů"(SM). V závislosti na architekuře má každý SM jiný počet jader. V práci používaná RTX 4080 má 80 SM a každý z nich 128 jader. Tudíž teoreticky může běžet až 10240 výpočtů najednou. Avšak tato vlákna jsou shlukována do warpů a ty musí vykonávat stejnou instrukci. V nějakém bodě potřebuje program načíst nová data. Načítání trvá relativně dlouho, a proto je GPU navrženo tak, aby dokázalo vyměnit aktuálně zpracovávaný warp, který právě načítá data, za nějaký volný. Tím se skryje latence načítání, jelikož fyzické jádro stále něco dělá.

Potřebujeme tedy dostatek perzistentních vláken na to, abychom plně saturovali fyzická vlákna. Spustíme proto přesně tolik warpů, kolik jich může být aktivně zpracováváno na SM. Pro RTX 4080 je to 32 warpů na SM. Tudíž spustíme 2560 warpů perzistentních vláken. To je přibližně 8 vláken na jedno fyzické jádro, což by mělo být dostatečné pro maximální vytížení grafické karty. Počet warpů perzistentních vláken jde manuálně nastavit pomocí parametru programu nebo v JSON souboru.

V tabulce 4.2 můžete vidět porovnání rychlosti vrhání paprsků při využití různého počtu warpů. Je zjevné, že RTX 4080 dokáže využít těchto vláken více a díku tomu značně zvýšit výkon a to klidně až o polovinu.

Scéna	**Počet warpů**	**Primary [Mray/s]**	**AO [Mray/s]**	**Diffuse [Mray/s]**
Hairball	2560	**449.4 (1.00)**	**221.1 (1.00)**	**196.3 (1.00)**
	720	302.7 (0.67)	139.9 (0.63)	123.1 (0.63)
San Miguel	2560	**494.0 (1.00)**	**700.1 (1.00)**	**327.7 (1.00)**
	720	312.3 (0.63)	436.4 (0.62)	201.8 (0.62)
Powerplant	2560	**350.0 (1.00)**	**1507.4 (1.00)**	**463.4 (1.00)**
	720	332.5 (0.95)	1068.7 (0.71)	337.7 (0.73)

Tabulka 4.2: Měření rychlosti vrhání paprsků v závislosti na počtu perzistentních warpů. Měřeno na AABB po optimalizaci s PRBVH.

Pro plné vytížení grafické karty je také zapotřebí, aby blok vláken měl dostatečnou paměť pro registry. Pokud by jich nebyl dostatek, muselo by vlákno sahat do pomalejší globální paměti, což není chtěné, jelikož by se tím zpomaloval celý warp. Proto je důležité zkontrolovat počty registrů, které jednotlivé varianty kernelu pro vrhání paprsků používají.

- AABB - 40 registrů
- 14-DOP bez rozdělení - 46 registrů
- 14-DOP s rozdělením - 51 registrů
- SOBB direct - 40 registrů
- SOBB indirect - 48 registrů

Pomocí kalkulačky obsazenosti, která je vlastně excelovou tabulkou přikládanou NVIDIA k instalaci CUDA, bylo zjištěno, že obsazenost bude úplná pro bloky velikosti 256 a kernely s maximálně 64 registry. Původní implementace používala velikost bloku 192, jelikož hodnota byla optimální pro GTX 250. To však podle kalkulačky neodpovídá ideální hodnotě pro compute capability 7.5. Vyzkoušeny byly tedy obě možnosti. Výsledky je možné vidět v tabulce 4.3. Z ní vyplývá, že je lepší používat bloky o velikosti 192. Proč tomu tak je, nebyla práce schopna objasnit.

Scéna	**Počet vláken**	**Primary [Mray/s]**	**AO [Mray/s]**	**Diffuse [Mray/s]**
Hairball	192	**311.5 (1.00)**	**137.4 (1.00)**	**122.2 (1.00)**
	256	227.6 (0.73)	97.5 (0.71)	87.4 (0.72)
San Miguel	192	**359.0 (1.00)**	**431.9 (1.00)**	**199.5 (1.00)**
	256	298.7 (0.83)	361.7 (0.84)	170.0 (0.85)
Powerplant	192	**331.6 (1.00)**	**572.2 (1.00)**	**327.4 (1.00)**
	256	310.6 (0.94)	505.6 (0.88)	299.1 (0.91)

Tabulka 4.3: Měření rychlosti vrhání paprsků v závislosti na velikosti bloku kernelu. Měřeno na AABB po optimalizaci s PRBVH.

Dále je potřeba mít pro blok vláken i dostatek sdílené paměti. Kernel vrhání paprsků používá 3 proměnné ve sdílené paměti a pro každé vlákno vyžaduje celkem 28 bytů. V tomto případě je blok velký 192 vláken (6 warpů). To je celkem 5376 bytů. RTX 4080 umožňuje spustit až 1536 vláken na jednom SM. Tudíž se na jeden SM vejde 8 bloků, které potřebují celkem 43008 bytů sdílené paměti. To je méně než maximálních 49152 bytů, které SM v RTX 4080 povoluje.

4.2 Analýza výsledků

V tabulkách výsledků měření jsou tučně vyznačena měření, která byla napříč testovanými metodami nejlepší. Některé tabulky obsahují očekávané výsledky dle původních článků. Ty pak slouží k porovnání se články, ze kterých se vycházelo. Není však možné zajistit přesnost tohoto porovnání, a proto by se měly posuzovat pouze orientačně. Nemůžeme si být jisti, že se opravdu jedná o stejné scény nebo zda se používala stejná nastavení optimalizace. Nebyla zároveň použita stejná grafická karta.

4.2.1 Porovnávání variant obalových těles

Nyní porovnáme různé varianty obalových těles, abychom věděli, jaké z jich použít pro měření finální množiny scén. Není totiž možné vyzkoušet a porovnat všechny kombinace obalových těles a jejich nastavení na tak velkém množství scén. Proto každé těleso bude reprezentovat jeho nejlepší varianta.

Typy reprezentací 14-DOPu v paměti

Jak rozložení 14-DOPu v paměti ovlivňuje výkon vrhání paprsků, je možné vidět v tabulce 4.4. Ukazuje se, že rozdělená varianta je o necelé procento pomalejší. Pro testování tedy použijeme nerozdělenou variantu. Pozorování však vůbec neodpovídá původnímu článku 2.2, kde bylo pozorováno zlepšení 3-6%. Předpokladem je, že výsledky jsou rozdílné kvůli typu kernelu vrhání paprsků, který používá uzel obsahující obalová tělesa obou potomků.

Scéna	Rozložení	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Hairball	Nerozděleno	**375.8 (1.00)**	**170.6 (1.00)**	**152.2 (1.00)**
	Rozděleno	369.1 (0.98)	169.4 (0.99)	151.5 (1.00)
San Miguel	Nerozděleno	**439.7 (1.00)**	984.7 (1.00)	**242.7 (1.00)**
	Rozděleno	426.9 (0.97)	**1056.0 (1.07)**	239.1 (0.99)
Poweplant	Nerozděleno	**296.2 (1.00)**	**374.4 (1.00)**	**172.0 (1.00)**
	Rozděleno	286.9 (0.97)	364.1 (0.97)	170.0 (0.99)

Tabulka 4.4: Měření rychlosti vrhání paprsků v závislosti na rozložení 14-DOPu v paměti. Měřeno po optimalizaci s PRBVH.

Typy reprezentace SOBB v paměti

Z naměřených výsledků v tabulce 4.5 je vidět, že přímá reprezentace SOBB, která využívá zakódování normály a rozsahu na ní do čtyřsložkového vektoru, je o více jak 25% rychlejší než nepřímá varianta. To odpovídá výsledkům z článku [10]. Kvůli těmto výsledkům byla u všech následujících měření použita přímá reprezentace SOBB.

Scéna	Rozložení	Primary	AO	Diffuse
Hairball	Přímé	414.3 (1.00)	**185.9 (1.00)**	**171.4 (1.00)**
	Nepřímé	339.9 (0.82)	150.7 (0.81)	131.2 (0.77)
San Miguel	Přímé	305.0 (1.00)	**393.3 (1.00)**	**182.1 (1.00)**
	Nepřímé	249.4 (0.82)	306.0 (0.78)	138.0 (0.76)
Powerplant	Přímé	441.1 (1.00)	**937.2 (1.00)**	**255.0 (1.00)**
	Nepřímé	370.6 (0.84)	556.5 (0.59)	207.1 (0.81)

Tabulka 4.5: Měření rychlosti vrhání paprsků v závislosti na reprezentaci SOBB v paměti. Měřeno po optimalizaci s PRBVH.

Porovnání počtu normál SOBB

Z tabulky 4.6 je zjevné, že SOBB vytvořené z 64-DOPu má nejlepší výkon. Rozdíly však nejsou příliš velké a zrychlení se pohybuje většinou kolem 10-15%. Cenou za toto zrychlení je přibližně sedmkrát delší stavba. Proto si myslím, že SOBB ze 32-DOPu je jediným použitelným tělesem, jelikož než se ostatní hierarchie postavily, byl snímek používající SOBB nad 32-DOPem již dávno vykreslen a změřen.

Scéna	Počet normál	Čas stavby [s]	SAH	EPO	LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Hairball	32	**44.3 (1.00)**	927.8 (1.00)	12.2 (1.00)	33.1 (1.00)	476.2 (1.00)	219.2 (1.00)	203.7 (1.00)
	48	137.0 (3.10)	878.4 (0.95)	10.4 (0.86)	30.6 (0.92)	517.3 (1.09)	251.5 (1.15)	222.6 (1.09)
	64	296.1 (6.69)	**850.0 (0.92)**	**9.5 (0.78)**	**29.2 (0.88)**	**543.3 (1.14)**	**257.0 (1.17)**	**234.2 (1.15)**
San Miguel	32	**107.7 (1.00)**	161.4 (1.00)	**6.4 (1.00)**	6.1 (1.00)	433.8 (1.00)	608.0 (1.00)	272.9 (1.00)
	48	355.7 (3.30)	158.2 (0.98)	7.8 (1.21)	5.7 (0.93)	480.2 (1.11)	623.9 (1.03)	298.6 (1.09)
	64	1020.8 (9.47)	**157.6 (0.98)**	6.8 (1.06)	**5.6 (0.91)**	**502.6 (1.16)**	**642.6 (1.06)**	**307.6 (1.13)**
Powerplant	32	**227.9 (1.00)**	71.7 (1.00)	5.1 (1.00)	20.5 (1.00)	481.1 (1.00)	638.6 (1.00)	422.5 (1.00)
	48	773.9 (3.40)	70.8 (0.99)	4.9 (0.96)	20.2 (0.98)	852.3 (1.77)	643.2 (1.01)	**439.8 (1.04)**
	64	1622.0 (7.12)	**70.5 (0.98)**	**4.8 (0.95)**	**20.1 (0.98)**	**890.7 (1.85)**	**654.3 (1.02)**	423.2 (1.00)

Tabulka 4.6: Měření rychlosti vrhání paprsků pro SOBB v závislosti na počtu normál. Měřeno s $O(n^3)$ strategií a PRBVH optimalizací.

Porovnání strategií tvorby SOBB

Tabulka 4.7 ukazuje, že i když by strategie $O(n^3)$ měla být nejlepší, tak tomu tak není, a to převážně ve větších scénách. V menších scénách $O(n^3)$ má výkon o trochu lepší, ale když vezmeme v potaz dobu optimalizace, tak se její použití nevyplatí. Za nejlepší strategii považuji $O(n)$. I když je naivní a velmi zvyšuje pravděpodobnost výběru neoptimálních normál, dokáže držet s ostatními strategiemi krok a na velkých scénách je hravě předčí. Ve článku zabývajícím se SOBB [10] porovnávali výkon pro sekundární paprsky (v této práci označené jako diffuse) a došli k závěru, že nejlepších výsledků dosahuje strategie $O(n^3)$ následovaná $O(n^2)$. Pro sekundární paprsky je výkon složitějších strategií horší jen na scéně Powerplant. Jinde si drží o $\pm 2 - 3\%$ lepší výkon. Je možné, že strategiím nevyhovuje zrovna tato scéna a kdyby se měřilo na jiných scénách, došlo by se ke stejným závěrům jako v článku.

Těleso	Strategie	Čas stavby [s]	SAH	EPO	LCV	Primary [Mrays/s]	AO [Mrays/s]	Diffuse [Mrays/s]
Hairball								
SOBB 32	$O(n)$	**16.9 (1.00)**	946.9 (1.00)	13.0 (1.00)	33.6 (1.00)	463.6 (1.00)	226.7 (1.00)	200.1 (1.00)
	$O(n^2)$	24.7 (1.46)	937.4 (0.99)	12.7 (0.97)	33.1 (0.98)	469.6 (1.01)	**231.1 (1.02)**	203.1 (1.01)
	$O(n^3)$	44.1 (2.61)	**927.8 (0.98)**	**12.2 (0.93)**	**33.0 (0.98)**	**474.7 (1.02)**	227.4 (1.00)	**206.0 (1.03)**
SOBB 48	$O(n)$	**20.9 (1.00)**	897.5 (1.00)	11.1 (1.00)	**30.4 (1.00)**	516.6 (1.00)	238.0 (1.00)	217.4 (1.00)
	$O(n^2)$	31.3 (1.50)	888.1 (0.99)	11.3 (1.01)	**30.4 (1.00)**	522.0 (1.01)	**238.5 (1.00)**	219.2 (1.01)
	$O(n^3)$	136.8 (6.55)	**878.4 (0.98)**	**10.4 (0.94)**	30.6 (1.01)	**527.5 (1.02)**	238.4 (1.00)	**221.3 (1.02)**
SOBB 64	$O(n)$	**25.5 (1.00)**	869.4 (1.00)	11.0 (1.00)	**28.9 (1.00)**	543.2 (1.00)	259.9 (1.00)	227.3 (1.00)
	$O(n^2)$	40.2 (1.58)	860.2 (0.99)	10.3 (0.93)	**28.9 (1.00)**	543.4 (1.00)	**261.9 (1.01)**	232.0 (1.02)
	$O(n^3)$	297.5 (11.69)	**850.0 (0.98)**	**9.5 (0.86)**	29.3 (1.01)	**546.2 (1.01)**	**261.2 (1.01)**	**234.5 (1.03)**
San Miguel								
SOBB 32	$O(n)$	**77.0 (1.00)**	163.3 (1.00)	**6.4 (1.00)**	6.2 (1.00)	**433.4 (1.00)**	578.4 (1.00)	273.6 (1.00)
	$O(n^2)$	78.8 (1.02)	161.7 (0.99)	7.4 (1.15)	6.2 (0.99)	405.1 (0.93)	577.2 (1.00)	**283.4 (1.04)**
	$O(n^3)$	107.5 (1.40)	**161.4 (0.99)**	**6.4 (1.00)**	**6.1 (0.98)**	419.5 (0.97)	**590.1 (1.02)**	272.9 (1.00)
SOBB 48	$O(n)$	**85.5 (1.00)**	161.1 (1.00)	7.7 (1.00)	5.8 (1.00)	431.6 (1.00)	624.3 (1.00)	277.7 (1.00)
	$O(n^2)$	**85.6 (1.00)**	161.2 (1.00)	**6.6 (0.86)**	**5.7 (0.99)**	396.5 (0.92)	627.9 (1.01)	289.9 (1.04)
	$O(n^3)$	353.5 (4.13)	**158.2 (0.98)**	7.8 (1.02)	**5.7 (0.99)**	**473.2 (1.10)**	**629.8 (1.01)**	**298.4 (1.07)**
SOBB 64	$O(n)$	**91.1 (1.00)**	159.7 (1.00)	7.0 (1.00)	5.7 (1.00)	484.1 (1.00)	630.2 (1.00)	299.8 (1.00)
	$O(n^2)$	124.6 (1.37)	157.9 (0.99)	**6.1 (0.87)**	5.6 (0.99)	**493.3 (1.02)**	625.5 (0.99)	**311.0 (1.04)**
	$O(n^3)$	1019.1 (11.19)	**157.6 (0.99)**	6.8 (0.97)	**5.6 (0.98)**	486.3 (1.00)	**664.6 (1.05)**	307.9 (1.03)
Powerplant								
SOBB 32	$O(n)$	**123.2 (1.00)**	72.5 (1.00)	5.1 (1.00)	**20.4 (1.00)**	**750.8 (1.00)**	739.2 (1.00)	438.8 (1.00)
	$O(n^2)$	140.4 (1.14)	72.4 (1.00)	5.3 (1.03)	20.5 (1.00)	617.3 (0.82)	647.7 (0.88)	**451.5 (1.03)**
	$O(n^3)$	227.2 (1.84)	**71.7 (0.99)**	**5.1 (0.99)**	**20.4 (1.00)**	457.3 (0.61)	**821.7 (1.11)**	435.4 (0.99)
SOBB 48	$O(n)$	**154.0 (1.00)**	**70.8 (1.00)**	5.1 (1.00)	20.2 (1.00)	**877.5 (1.00)**	**693.7 (1.00)**	**523.1 (1.00)**
	$O(n^2)$	189.1 (1.23)	**70.8 (1.00)**	5.2 (1.01)	**20.1 (0.99)**	808.7 (0.92)	640.0 (0.92)	490.6 (0.94)
	$O(n^3)$	771.9 (5.01)	**70.8 (1.00)**	**4.9 (0.95)**	20.2 (1.00)	809.0 (0.92)	667.2 (0.96)	467.1 (0.89)
SOBB 64	$O(n)$	**170.2 (1.00)**	70.8 (1.00)	5.0 (1.00)	**20.1 (1.00)**	788.2 (1.00)	**706.5 (1.00)**	**484.3 (1.00)**
	$O(n^2)$	239.7 (1.41)	**70.4 (1.00)**	**4.8 (0.96)**	**20.1 (1.00)**	637.6 (0.81)	627.4 (0.89)	429.8 (0.89)
	$O(n^3)$	1619.2 (9.51)	70.5 (1.00)	**4.8 (0.96)**	**20.1 (1.00)**	**869.4 (1.10)**	599.8 (0.85)	412.2 (0.85)

Tabulka 4.7: Měření rychlosti vrhání paprsků v závislosti na počtu a strategii výběru normál SOBB.

Ořezávání SOBB

Pro prozkoumání míry, do které obalová tělesa SOBB potomků vystupují ven z předka, byla naimplementována možnost ořezávat SOBB při výpočtu SAHu. Pro každý uzel se postupuje nahoru ke kořenu hierarchie a ořezává se tělesy předků oblast k-DOPu, nad kterým je SOBB postaven. Následně byl využit face clipping algoritmus ze článku o 14-DOPech [9], který byl trochu upravil, aby podporoval ještě více stěn k výpočtu plochy a spočetl plochu výsledného k-DOPu. Rozdíl mezi výsledným SAHem měřeným na LBVH hierarchii je vidět v tabulce 4.8. Při vrhání paprsků se nemusíme o ořezávání nijak starat, jelikož postupný průchod hierarchií jej vlastně simuluje.

Scéna	Typ ořezu	SAH
Sibenik	Bez ořezu	234.3 (1.00)
	Ořez jedním předkem	226.5 (0.97)
	Ořez všemi předky	**223.2 (0.95)**
Fairy forest	Bez ořezu	119.0(1.00)
	Ořez jedním předkem	116.0 (0.98))
	Ořez všemi předky	**115.1 (0.97)**
Hairball	Bez ořezu	1467.0 (1.00)
	Ořez jedním předkem	1381.5 (0.94)
	Ořez všemi předky	**1338.0 (0.91)**

Tabulka 4.8: Porovnání hodnoty SAH pro SOBB 16 pro různé způsoby ořezávání. Měřeno se strategií $O(n^2)$ s hierarchií vytvořenou LBVH.

Na komplexnějších scénách dosahuje ořezávání snížení SAHu o 5% až 10%. To naznačuje, že se metrika, kterou optimalizuje PRBVH, může výrazně lišit od opravdového SAHu. Je tedy možné, že by optimalizace PRBVH, která by počítala s ořezáváním, mohla najít ještě lepší hierarchii. Pokusili jsme se tedy narychlo implementovat tuto optimalizaci, avšak nedá se považovat za úspěšnou. Ořezávání a následný face clipping se musí provádět pro každý uzel, na který se PRBVH snaží přemisťovaný uzel umístit. Tento postup způsobuje velké zpomalení. Je možné, že implementace není korektní, jelikož jí nebylo čas řádně otestovat. Porovnání výsledků je vidět v tabulce 4.9. Optimalizace dosahuje lepších výsledků než optimalizace bez ořezávání, když se ořeže pouze první předek. Při použití ořezu více předky je výsledká hierarchie méně kvalitní. To je možná způsobeno tím, že zapadla do jiného horšího lokálního minima nebo přemístění uzlů způsobí výběr jiných normál nových předků a ty zvětší SAH více než byl zmenšel pomocí přesunu uzlu. To však nebylo dokázáno, protože nebylo změřeno dostatek scén. Hlavní problém je však s trváním optimalizace, která je několikanásobně delší při ořezávání všech předků. Proto tuto techniku můžeme označit za nepoužitelnou pro optimalizaci s PRBVH.

Scéna	Typ ořezu	Čas stavby [s]	SAH
Sibenik	Bez ořezu	**0.8 (1.00)**	150.3 (1.00)
	Ořez jedním předkem	28.7 (37.24)	**146.9 (0.98)**
	Ořez všemi předky	476.2 (618.91)	176.7 (1.18)
Fairy forest	Bez ořezu	**0.9 (1.00)**	98.7 (1.00)
	Ořez jedním předkem	55.6 (60.73)	**96.7 (0.98)**
	Ořez všemi předky	1115.7 (1219.71)	106 (1.08)
Hairball	Bez ořezu	**24.8 (1.00)**	937.3 (1.00)
	Ořez jedním předkem	1481.1 (59.72)	**861.2 (0.92)**
	Ořez všemi předky	xxx	xxx

Tabulka 4.9: Porovnání hodnoty SAH pro SOBB 16 pro optimalizované PRBVH s různými způsoby ořezávání. Měřeno se strategií $O(n^2)$.

4.2.2 Statické scény

S optimální paměťovou reprezentací 14-DOPů a SOBB získanou z předešlých testů byl změřen výkon a metriky pro deset statických scén různé komplexity. Tabulka 4.10 porovnává hierarchie vytvořené pomocí LBVH a hierarchie optimalizované s PRBVH. Až na pár výjimek byl vylepšen jak výkon, tak hodnoty měřených metrik.

Při porovnání časů stavby je vidět, jak je tvorba SOBB pomalá. I při použití $O(n^2)$ strategie trvala optimalizace pro větší scény klidně několik minut. Přestože dokáže SOBB v některých scénách Bistro interior nebo Conference získat náskok oproti ostatním tělesům, není to dost na vykompenzování času stráveného na optimalizaci.

14-DOP se osvědčil na komplexnějších scénách jako je Powerplant nebo Hairball, kde předčí ostatní tělesa.

Velmi zvláštní výsledky jde pozorovat na scéně Vokselia. Ta představuje geometrii Minecraft světa tvořeného z osově zarovnaných bloků. Je zvláštní, že je to jediná scéna, kde PRBVH nebylo schopné vylepšit výkon o tolik jako u ostatních. SOBB a 14-DOP zaznamenaly lehké zlepšení, ale pro AABB optimalizace selhala. Primární paprsky jsou zlepšené pouze o 5% a pro AO s difuzními paprsky klesl výkon o polovinu.

Na všech scénách lze pozorovat, že hodnota metriky LCV pro AABB je podstatně vyšší než u ostatních těles. To je způsobeno menším počtem překryvů obalových těles a také menším počtem listů. 14-DOP a SOBB mají konstantu $C_t = 4.5$ a tudíž může adaptivní slukování listů více zapracovat a snížit počet listů. S nižším počtem listů musí klesnout i LCV. Kvůli adaptivnímu shlukování listů nemůžeme říci, že by složitější obalová tělesa byla lepší právě kvůli metrice LCV. Můžeme však porovnat LCV před a po optimalizaci pomocí PRBVH. Zde se zdá, že nějaká korelace je při porovnání poměry změny LCV a výkonu. LCV se při zkvalitnění hierarchie zmenšuje a výkon naopak zvětšuje. Je tedy potřeba pro porovnání násobků (hodnot v závorkách) nepoužívat hodnotu pro LCV, ale raději $\frac{1}{\mathrm{LCV}}$. Na některých scénách se takto upravené násobky LCV blíží násobkům pro výkon vrhání paprsků.

Při porovnávání hodnot metriky EPO můžeme vidět, že těsnější tělesa dokáží výrazně snížit jeho hodnotu. To platí hlavně pro SOBB, který dokáže být mnohem plošší. Tím se zmenšuje část plochy trojúhelníku, která se nachází v obalovém tělese. Ve většině scén je hodnota EPO větší pro 14-DOP než AABB. 14-DOP dokáže být těsnější než AABB, ale v EPO se to neprojeví, jelikož je počítáno s konstantou C_t, která je o polovinu větší u 14-DOPu. Jeho těsnost však tento rozdíl nedokáže vykompenzovat.

Tabulka 4.10: Soubor měření statických scén. Čas stavby v sekundách a rychlost vrhání paprsků v Mray/s. SOBB měřeno se strategií $O(n^2)$.

Těleso	Algoritmus	Čas stavby [s]	SAH	EPO	LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Bistro interior								
AABB	LBVH	**0.1 (1.00)**	350.3 (1.00)	104.7 (1.00)	14.2 (1.00)	527.5 (1.00)	368.1 (1.00)	202.5 (1.00)
	PRBVH	0.4 (5.90)	**250.8 (0.72)**	**47.7 (0.46)**	**11.6 (0.82)**	**901.6 (1.71)**	**528.8 (1.44)**	**311.7 (1.54)**
14-DOP	LBVH	**0.1 (1.00)**	286.7 (1.00)	102.9 (1.00)	9.4 (1.00)	538.0 (1.00)	383.2 (1.00)	211.5 (1.00)
	PRBVH	0.5 (7.33)	**194.8 (0.68)**	**41.0 (0.40)**	**8.2 (0.88)**	**909.7 (1.69)**	**519.3 (1.36)**	**326.3 (1.54)**
SOBB 32	LBVH	**0.1 (1.00)**	271.5 (1.00)	14.5 (1.00)	7.8 (1.00)	518.3 (1.00)	449.4 (1.00)	238.6 (1.00)
	PRBVH	8.5 (152.94)	**154.9 (0.57)**	**9.5 (0.66)**	**6.9 (0.88)**	**985.1 (1.90)**	**689.0 (1.53)**	**452.0 (1.89)**
SOBB 48	LBVH	**0.1 (1.00)**	263.8 (1.00)	13.3 (1.00)	7.6 (1.00)	523.0 (1.00)	450.6 (1.00)	248.9 (1.00)
	PRBVH	9.6 (110.09)	**147.3 (0.56)**	**8.1 (0.61)**	**6.6 (0.87)**	**1021.2 (1.95)**	**719.4 (1.60)**	**451.4 (1.81)**
SOBB 64	LBVH	**0.1 (1.00)**	265.9 (1.00)	15.7 (1.00)	7.5 (1.00)	534.6 (1.00)	481.1 (1.00)	255.1 (1.00)
	PRBVH	14.0 (109.05)	**151.8 (0.57)**	**9.7 (0.62)**	**6.5 (0.86)**	**1019.3 (1.91)**	**756.3 (1.57)**	**475.9 (1.87)**
Buddha								
AABB	LBVH	**0.05 (1.00)**	191.5 (1.00)	14.5 (1.00)	5.1 (1.00)	1172.9 (1.00)	447.8 (1.00)	472.9 (1.00)
	PRBVH	0.2 (5.10)	**157.2 (0.82)**	**10.2 (0.70)**	**4.6 (0.92)**	**1424.9 (1.21)**	**512.8 (1.15)**	**558.5 (1.18)**
14-DOP	LBVH	**0.1 (1.00)**	203.5 (1.00)	14.9 (1.00)	3.1 (1.00)	1094.1 (1.00)	440.7 (1.00)	429.2 (1.00)
	PRBVH	0.4 (5.53)	**167.6 (0.82)**	**10.8 (0.73)**	**2.8 (0.92)**	**1284.4 (1.17)**	**442.6 (1.00)**	**473.4 (1.10)**
SOBB 32	LBVH	**0.1 (1.00)**	242.7 (1.00)	4.7 (1.00)	3.0 (1.00)	1025.0 (1.00)	391.4 (1.00)	449.4 (1.00)
	PRBVH	5.9 (114.30)	**195.1 (0.80)**	**2.7 (0.58)**	**2.7 (0.90)**	**1230.3 (1.20)**	**446.9 (1.14)**	**498.4 (1.11)**
SOBB 48	LBVH	**0.1 (1.00)**	237.5 (1.00)	4.5 (1.00)	2.8 (1.00)	1179.9 (1.00)	466.1 (1.00)	461.7 (1.00)
	PRBVH	7.0 (97.57)	**191.6 (0.81)**	**2.6 (0.57)**	**2.6 (0.91)**	**1384.8 (1.17)**	**481.0 (1.03)**	**519.2 (1.12)**
SOBB 64	LBVH	**0.1 (1.00)**	233.2 (1.00)	4.4 (1.00)	2.6 (1.00)	1237.1 (1.00)	453.9 (1.00)	502.0 (1.00)
	PRBVH	9.7 (94.74)	**187.8 (0.81)**	**2.5 (0.56)**	**2.4 (0.91)**	**1389.3 (1.12)**	**546.2 (1.20)**	**598.3 (1.19)**
Conference								
AABB	LBVH	**0.02 (1.00)**	146.2 (1.00)	40.0 (1.00)	4.8 (1.00)	1079.7 (1.00)	1278.1 (1.00)	709.0 (1.00)
	PRBVH	0.2 (7.28)	**83.6 (0.57)**	**15.5 (0.39)**	**4.7 (0.98)**	**1555.2 (1.44)**	**1591.7 (1.25)**	**977.7 (1.38)**
14-DOP	LBVH	**0.02 (1.00)**	184.9 (1.00)	47.5 (1.00)	4.0 (1.00)	955.9 (1.00)	1117.5 (1.00)	587.1 (1.00)
	PRBVH	0.3 (14.34)	**105.0 (0.57)**	**19.3 (0.41)**	**4.0 (0.99)**	**1393.3 (1.46)**	**1462.6 (1.31)**	**860.3 (1.47)**
SOBB 32	LBVH	**0.02 (1.00)**	201.5 (1.00)	18.4 (1.00)	4.1 (1.00)	901.0 (1.00)	1271.9 (1.00)	602.5 (1.00)
	PRBVH	2.2 (110.52)	**107.3 (0.53)**	**11.5 (0.63)**	**4.0 (0.96)**	**1465.4 (1.63)**	**1623.8 (1.28)**	**982.8 (1.63)**
SOBB 48	LBVH	**0.02 (1.00)**	199.2 (1.00)	18.5 (1.00)	3.9 (1.00)	967.5 (1.00)	**1158.0 (1.00)**	634.6 (1.00)
	PRBVH	3.4 (177.35)	**105.6 (0.53)**	**12.8 (0.69)**	**3.7 (0.96)**	**1634.1 (1.69)**	834.8 (0.72)	**869.3 (1.37)**
SOBB 64	LBVH	**0.03 (1.00)**	198.7 (1.00)	18.0 (1.00)	3.8 (1.00)	993.0 (1.00)	1246.1 (1.00)	663.2 (1.00)
	PRBVH	4.8 (160.50)	**105.4 (0.53)**	**11.7 (0.65)**	**3.6 (0.95)**	**1792.1 (1.80)**	**1518.6 (1.22)**	**1091.3 (1.65)**
Fairy forest								
AABB	LBVH	**0.02 (1.00)**	89.0 (1.00)	9.3 (1.00)	3.5 (1.00)	1168.3 (1.00)	812.4 (1.00)	711.4 (1.00)
	PRBVH	0.1 (4.06)	**76.0 (0.85)**	**6.4 (0.69)**	**3.2 (0.93)**	**1487.7 (1.27)**	**963.5 (1.19)**	**860.9 (1.21)**
14-DOP	LBVH	**0.02 (1.00)**	107.2 (1.00)	10.4 (1.00)	2.5 (1.00)	1102.3 (1.00)	706.5 (1.00)	596.7 (1.00)
	PRBVH	0.2 (8.52)	**92.4 (0.86)**	**5.6 (0.54)**	**2.1 (0.83)**	**1365.8 (1.24)**	**802.0 (1.14)**	**759.0 (1.27)**
SOBB 32	LBVH	**0.01 (1.00)**	119.0 (1.00)	**2.1 (1.00)**	2.6 (1.00)	1012.7 (1.00)	695.1 (1.00)	571.1 (1.00)
	PRBVH	1.0 (86.15)	**98.1 (0.82)**	2.4 (1.15)	**2.1 (0.83)**	**1278.4 (1.26)**	**906.9 (1.30)**	**766.1 (1.34)**
SOBB 48	LBVH	**0.02 (1.00)**	119.0 (1.00)	**2.1 (1.00)**	2.4 (1.00)	1060.6 (1.00)	663.7 (1.00)	594.9 (1.00)
	PRBVH	1.7 (80.30)	**98.8 (0.83)**	2.3 (1.10)	**2.0 (0.83)**	**1407.5 (1.33)**	**877.5 (1.32)**	**796.0 (1.34)**
SOBB 64	LBVH	**0.02 (1.00)**	118.7 (1.00)	**2.1 (1.00)**	2.4 (1.00)	1063.7 (1.00)	727.8 (1.00)	613.5 (1.00)
	PRBVH	2.4 (111.16)	**97.6 (0.82)**	2.2 (1.03)	**1.9 (0.80)**	**1399.7 (1.32)**	**949.8 (1.30)**	**825.5 (1.35)**
Hairball								
AABB	LBVH	**0.1 (1.00)**	1238.1 (1.00)	112.5 (1.00)	97.8 (1.00)	363.2 (1.00)	169.2 (1.00)	149.4 (1.00)
	PRBVH	0.8 (6.22)	**985.0 (0.80)**	**79.3 (0.71)**	**66.2 (0.68)**	**458.3 (1.26)**	**217.6 (1.29)**	**197.1 (1.32)**
14-DOP	LBVH	**0.2 (1.00)**	1061.8 (1.00)	87.4 (1.00)	49.1 (1.00)	379.7 (1.00)	167.8 (1.00)	153.8 (1.00)
	PRBVH	1.1 (6.29)	**792.7 (0.75)**	**51.1 (0.58)**	**27.6 (0.56)**	**548.6 (1.44)**	**254.6 (1.52)**	**235.2 (1.53)**
SOBB 32	LBVH	**0.1 (1.00)**	1301.5 (1.00)	20.1 (1.00)	48.1 (1.00)	356.8 (1.00)	171.9 (1.00)	150.7 (1.00)
	PRBVH	24.6 (184.40)	**937.4 (0.72)**	**12.7 (0.63)**	**33.1 (0.69)**	**465.2 (1.30)**	**220.6 (1.28)**	**203.8 (1.35)**
SOBB 48	LBVH	**0.2 (1.00)**	1242.9 (1.00)	18.9 (1.00)	42.3 (1.00)	399.7 (1.00)	182.0 (1.00)	161.9 (1.00)
	PRBVH	31.3 (202.00)	**888.1 (0.71)**	**11.3 (0.60)**	**30.6 (0.72)**	**514.3 (1.29)**	**249.5 (1.37)**	**220.2 (1.36)**
SOBB 64	LBVH	**0.2 (1.00)**	1210.6 (1.00)	18.2 (1.00)	39.5 (1.00)	414.2 (1.00)	191.4 (1.00)	166.9 (1.00)
	PRBVH	40.2 (251.15)	**860.2 (0.71)**	**10.3 (0.57)**	**28.9 (0.73)**	**538.3 (1.30)**	**262.2 (1.37)**	**232.6 (1.39)**
Manuscript								
AABB	LBVH	**0.3 (1.00)**	84.1 (1.00)	7.7 (1.00)	2.2 (1.00)	1007.4 (1.00)	1239.6 (1.00)	**1109.1 (1.00)**
	PRBVH	0.3 (1.31)	**81.4 (0.97)**	**7.1 (0.92)**	**2.1 (0.99)**	**1068.1 (1.06)**	**1321.6 (1.07)**	1042.4 (0.94)
14-DOP	LBVH	**0.3 (1.00)**	115.7 (1.00)	7.9 (1.00)	1.6 (1.00)	706.2 (1.00)	864.6 (1.00)	949.6 (1.00)
	PRBVH	0.5 (1.69)	**112.2 (0.97)**	**7.1 (0.90)**	**1.6 (0.98)**	**721.6 (1.02)**	**891.0 (1.03)**	**980.0 (1.03)**
SOBB 32	LBVH	0.2 **(1.00)**	119.5 (1.00)	3.1 (1.00)	**1.6 (1.00)**	707.4 (1.00)	847.7 (1.00)	922.2 (1.00)
	PRBVH	6.8 (30.19)	**115.8 (0.97)**	**2.7 (0.88)**	**1.6 (1.00)**	**736.8 (1.04)**	**872.7 (1.03)**	**963.1 (1.04)**
SOBB 48	LBVH	**0.2 (1.00)**	119.1 (1.00)	2.6 (1.00)	1.6 (1.00)	717.3 (1.00)	**954.1 (1.00)**	856.0 (1.00)
	PRBVH	9.2 (49.21)	**115.1 (0.97)**	**2.2 (0.84)**	**1.6 (0.98)**	**759.5 (1.06)**	890.4 (0.93)	**962.6 (1.12)**
SOBB 64	LBVH	**0.2 (1.00)**	118.8 (1.00)	2.6 (1.00)	1.6 (1.00)	738.3 (1.00)	826.6 (1.00)	**935.1 (1.00)**
	PRBVH	12.6 (56.15)	**114.8 (0.97)**	**2.1 (0.83)**	**1.5 (0.97)**	**760.3 (1.03)**	**1002.2 (1.21)**	907.7 (0.97)

Tabulka 4.10 – Pokračování tabulky z předchozí stránky

Těleso	Algoritmus	Čas stavby [s]	SAH	EPO	LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Powerplant								
AABB	LBVH	**0.4 (1.00)**	114.1 (1.00)	128.3 (1.00)	25.1 (1.00)	301.6 (1.00)	613.2 (1.00)	215.7 (1.00)
	PRBVH	1.0 (2.27)	**68.7 (0.60)**	**42.2 (0.33)**	**24.6 (0.98)**	**356.4 (1.18)**	**825.4 (1.35)**	**464.8 (2.15)**
14-DOP	LBVH	**0.5 (1.00)**	115.7 (1.00)	138.4 (1.00)	21.3 (1.00)	430.6 (1.00)	553.9 (1.00)	239.2 (1.00)
	PRBVH	3.9 (8.09)	**66.6 (0.58)**	**50.6 (0.37)**	**20.9 (0.98)**	**700.7 (1.63)**	**638.6 (1.15)**	**421.8 (1.76)**
SOBB 32	LBVH	**0.3 (1.00)**	135.8 (1.00)	**4.3 (1.00)**	21.3 (1.00)	374.9 (1.00)	**831.7 (1.00)**	217.9 (1.00)
	PRBVH	141.2 (453.04)	**72.4 (0.53)**	5.3 (1.25)	**20.5 (0.96)**	**603.3 (1.61)**	651.6 (0.78)	**468.3 (2.15)**
SOBB 48	LBVH	**0.4 (1.00)**	133.3 (1.00)	**4.1 (1.00)**	20.9 (1.00)	400.0 (1.00)	**660.2 (1.00)**	246.3 (1.00)
	PRBVH	188.9 (465.72)	**70.8 (0.53)**	5.2 (1.26)	**20.2 (0.97)**	**806.7 (2.02)**	643.5 (0.97)	**472.5 (1.92)**
SOBB 64	LBVH	**0.5 (1.00)**	131.8 (1.00)	**4.0 (1.00)**	20.7 (1.00)	419.2 (1.00)	517.1 (1.00)	250.1 (1.00)
	PRBVH	240.8 (478.98)	**70.4 (0.53)**	4.8 (1.21)	**20.0 (0.96)**	**669.5 (1.60)**	**693.5 (1.34)**	**450.5 (1.80)**
San Miguel								
AABB	LBVH	**0.4 (1.00)**	210.8 (1.00)	63.5 (1.00)	10.4 (1.00)	340.5 (1.00)	450.0 (1.00)	206.8 (1.00)
	PRBVH	0.8 (1.90)	**125.5 (0.60)**	**20.5 (0.32)**	**9.1 (0.87)**	**519.4 (1.53)**	**666.5 (1.48)**	**325.7 (1.58)**
14-DOP	LBVH	**0.3 (1.00)**	241.1 (1.00)	69.8 (1.00)	6.9 (1.00)	281.5 (1.00)	367.9 (1.00)	170.4 (1.00)
	PRBVH	2.1 (6.69)	**144.4 (0.60)**	**23.0 (0.33)**	**6.0 (0.87)**	**471.0 (1.67)**	**581.8 (1.58)**	**277.0 (1.63)**
SOBB 32	LBVH	**0.2 (1.00)**	285.4 (1.00)	18.0 (1.00)	7.4 (1.00)	258.1 (1.00)	353.1 (1.00)	163.4 (1.00)
	PRBVH	79.4 (405.49)	**161.7 (0.57)**	**7.4 (0.41)**	**6.1 (0.83)**	**386.9 (1.50)**	**555.6 (1.57)**	**281.7 (1.72)**
SOBB 48	LBVH	**0.3 (1.00)**	281.3 (1.00)	17.1 (1.00)	6.9 (1.00)	283.4 (1.00)	375.1 (1.00)	174.6 (1.00)
	PRBVH	85.3 (328.75)	**161.2 (0.57)**	**6.6 (0.39)**	**5.7 (0.83)**	**378.4 (1.34)**	**599.7 (1.60)**	**288.6 (1.65)**
SOBB 64	LBVH	**0.3 (1.00)**	279.6 (1.00)	17.1 (1.00)	6.8 (1.00)	288.6 (1.00)	385.4 (1.00)	178.8 (1.00)
	PRBVH	125.3 (369.29)	**157.9 (0.56)**	**6.1 (0.36)**	**5.6 (0.83)**	**474.0 (1.64)**	**641.3 (1.66)**	**309.6 (1.73)**
Sibenik								
AABB	LBVH	**0.01 (1.00)**	161.9 (1.00)	26.7 (1.00)	4.3 (1.00)	1836.7 (1.00)	**1478.1 (1.00)**	764.1 (1.00)
	PRBVH	0.1 (9.25)	**113.0 (0.70)**	**10.0 (0.37)**	**3.8 (0.90)**	**2620.2 (1.43)**	891.4 (0.60)	**851.5 (1.11)**
14-DOP	LBVH	**0.02 (1.00)**	185.0 (1.00)	27.2 (1.00)	3.1 (1.00)	1514.4 (1.00)	**844.7 (1.00)**	648.1 (1.00)
	PRBVH	0.2 (9.02)	**134.7 (0.73)**	**10.4 (0.38)**	**2.8 (0.91)**	**1982.9 (1.31)**	749.3 (0.89)	**808.5 (1.25)**
SOBB 32	LBVH	**0.02 (1.00)**	220.1 (1.00)	4.8 (1.00)	3.2 (1.00)	1216.0 (1.00)	**1242.4 (1.00)**	592.0 (1.00)
	PRBVH	0.7 (29.69)	**152.5 (0.69)**	**2.4 (0.50)**	**2.9 (0.89)**	**1822.2 (1.50)**	751.0 (0.60)	**782.6 (1.32)**
SOBB 48	LBVH	**0.03 (1.00)**	219.1 (1.00)	4.6 (1.00)	3.2 (1.00)	1292.1 (1.00)	**1259.6 (1.00)**	610.0 (1.00)
	PRBVH	1.2 (37.74)	**150.2 (0.69)**	**2.3 (0.49)**	**2.8 (0.90)**	**1782.9 (1.38)**	707.3 (0.56)	**813.5 (1.33)**
SOBB 64	LBVH	**0.01 (1.00)**	217.8 (1.00)	4.6 (1.00)	3.1 (1.00)	1288.5 (1.00)	951.2 (1.00)	619.2 (1.00)
	PRBVH	2.0 (159.86)	**150.9 (0.69)**	**2.3 (0.51)**	**2.8 (0.90)**	**1790.7 (1.39)**	**1418.3 (1.49)**	**846.6 (1.37)**
Vokselia								
AABB	LBVH	**0.1 (1.00)**	204.3 (1.00)	8.1 (1.00)	**6.5 (1.00)**	2457.4 (1.00)	**1382.3 (1.00)**	**1170.5 (1.00)**
	PRBVH	0.3 (2.62)	**177.7 (0.87)**	**7.9 (0.98)**	**6.5 (1.00)**	**2579.9 (1.05)**	663.0 (0.48)	652.5 (0.56)
14-DOP	LBVH	**0.1 (1.00)**	270.4 (1.00)	8.7 (1.00)	6.4 (1.00)	1423.3 (1.00)	939.6 (1.00)	969.4 (1.00)
	PRBVH	0.4 (3.02)	**233.8 (0.86)**	**8.2 (0.94)**	**6.3 (0.98)**	**1657.1 (1.16)**	**1005.3 (1.07)**	**1019.1 (1.05)**
SOBB 32	LBVH	**0.1 (1.00)**	297.0 (1.00)	**2.1 (1.00)**	6.4 (1.00)	1378.0 (1.00)	**994.8 (1.00)**	960.2 (1.00)
	PRBVH	8.4 (74.74)	**254.9 (0.86)**	2.2 (1.05)	**6.2 (0.96)**	**1542.3 (1.12)**	687.2 (0.69)	**1011.1 (1.05)**
SOBB 48	LBVH	**0.1 (1.00)**	298.0 (1.00)	**2.2 (1.00)**	6.4 (1.00)	1391.2 (1.00)	959.4 (1.00)	927.9 (1.00)
	PRBVH	10.1 (73.46)	**255.1 (0.86)**	2.2 (1.02)	**6.1 (0.96)**	**1477.0 (1.06)**	**1077.4 (1.12)**	**957.3 (1.03)**
SOBB 64	LBVH	**0.2 (1.00)**	298.8 (1.00)	**2.2 (1.00)**	6.4 (1.00)	1350.2 (1.00)	**941.9 (1.00)**	926.2 (1.00)
	PRBVH	**13.8 (77.23)**	**254.8 (0.85)**	2.2 (1.01)	**6.2 (0.96)**	**1528.6 (1.13)**	695.6 (0.74)	**994.0 (1.07)**

Tabulka 4.11 nabízí pohled na průměrné poměry výkonu pro různé typy tvorby hierarchie a typu obalového tělesa napříč všemi deseti scénami. Můžeme zde vidět, že nejlépe si vede optimalizovaná verze SOBB 64, která získala perfektní skóre. Na druhém místě je SOBB 48 následované s velmi podobnými výsledky optimalizovaným AABB. To dokázalo předčit 14-DOP i SOBB 32. Výhra optimalizovaných variant byla očividná. Je vidět, že značně složitější protnutí 14-DOPu předčilo jeho schopnost se lépe přizpůsobit scéně. SOBB s méně normálami pravděpodobně selhalo kvůli problému s ořezáváním. SOBB 64 dokázalo naopak tento nedostatek vykompenzovat. Pokud by nebyla prováděna optimalizace PRBVH, vyplatí se použít pouze AABB.

Těleso	Algoritmus	Primary	AO	Diffuse
AABB	LBVH	1.00 [6]	1.00 [6]	1.00 [6]
	PRBVH	1.31 [3]	1.13 [2]	1.30 [4]
14-DOP	LBVH	0.92 [8]	0.84 [10]	0.91 [9]
	PRBVH	1.30 [4]	1.03 [5]	1.26 [5]
SOBB 32	LBVH	0.85 [10]	0.93 [7]	0.90 [10]
	PRBVH	1.21 [5]	1.06 [4]	1.34 [3]
SOBB 48	LBVH	0.90 [9]	0.93 [8]	0.94 [8]
	PRBVH	1.34 [2]	1.06 [3]	1.35 [2]
SOBB 64	LBVH	0.93 [7]	0.90 [9]	0.97 [7]
	PRBVH	1.34 [1]	1.20 [1]	1.42 [1]

Tabulka 4.11: Porovnání průměrného zrychlení výkonu napříč deseti scénami. Levá hodnota je zrychlení daného obalového tělesa relativně k AABB hierarchie vytvořené LBVH. Pravá hodnota je pořadí jednotlivých těles a algoritmů (menší je lepší).

4.2.3 Animované scény

Pro porovnání tvorby hierarchií animovaných scén byla vytvořena Tabulka 4.12, která obsahuje nejdůležitější naměřená data. Pro lepší porovnání jednotlivých obalových těles a technik slouží Tabulka 4.13. Zde můžeme vidět jasnou převahu znovupostavení (Rebuild), které překonává výkonem T-SAH s jakoukoliv konstantou K. Z těles si nejlépe při znovupostavení vede SOBB 64, ale prohrává na celé čáře, co se týče rychlosti stavby hierarchie. Při použití T-SAHu dosahuje AABB nejlepších výsledků, které se dokonce blíží k hodnotám znovupostavení.

Při porovnávání časů stavby se může zdát, že T-SAH se vůbec nevyplácí. To je však způsobeno použitím pouze 25 snímků. V praxi se vykreslují stovky snímků a čas tvorby pro znovuvkládání roste lineárně s jejich počtem. Čas strávený na tvorbě hierarchie pomocí T-SAHu však závisí pouze na počtu klíčových snímků. Následně se pro každý snímek provede refit, který je ryhlý.

T-SAH hodnoty scény Breaking lion pro SOBB chybí kvůli nedostatku paměti. Tento nedostatek pramení ze zastaralé implementace optimalizace pomocí T-SAH, která nepředpokládala, že pro práci s obalovým tělesem bude potřeba 300 bytů na jedno těleso, jelikož je potřeba si udržovat hodnoty všech 64 polorovin.

Tabulka 4.12: Soubor změřených animovaných scén. Čas stavby v sekundách a rychlost vrhání paprsků v Mray/s. SOBB měřeno se strategií $O(n^2)$.

Těleso	Alg.	Čas stavby [s]	Průměr SAH	Max SAH	Průměr EPO	Průměr LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
24-cell									
AABB	Rebuild	0.4	115.5	110.1	19.5	5.5	1624.9 (1.00)	1929.1 (1.00)	435.7 (1.00)
	T-SAH$_0$	0.6	144.6	131.5	41.3	5.7	1434.0 (0.88)	1707.4 (0.89)	413.1 (0.95)
	T-SAH$_2$	0.7	147.0	131.9	41.6	5.7	1426.9 (0.88)	1691.6 (0.88)	414.9 (0.95)
	T-SAH$_4$	0.7	147.6	131.6	41.4	5.7	1391.5 (0.86)	1698.0 (0.88)	407.0 (0.93)
14-DOP	Rebuild	1.1	106.6	101.5	14.2	3.1	1713.8 (1.00)	1741.0 (1.00)	445.2 (1.00)
	T-SAH$_0$	0.9	127.3	115.9	30.6	3.2	1438.3 (0.84)	1645.3 (0.95)	432.7 (0.97)
	T-SAH$_2$	0.9	127.8	116.0	31.1	3.2	1394.1 (0.81)	1644.3 (0.94)	423.8 (0.95)
	T-SAH$_4$	0.9	127.7	116.1	30.6	3.2	1431.1 (0.84)	1655.5 (0.95)	430.4 (0.97)
SOBB 32	Rebuild	18.2	134.3	127.9	3.1	3.1	1498.1 (1.00)	1594.3 (1.00)	420.6 (1.00)
	T-SAH$_0$	6.7	159.9	147.9	3.4	3.1	1149.6 (0.77)	1293.3 (0.81)	412.0 (0.98)
	T-SAH$_2$	6.5	162.7	148.4	3.4	3.1	1151.9 (0.77)	1289.1 (0.81)	412.7 (0.98)
	T-SAH$_4$	7.1	160.2	147.6	3.3	3.1	1160.2 (0.77)	1265.5 (0.79)	417.9 (0.99)
SOBB 48	Rebuild	31.2	127.8	122.0	2.6	2.8	1612.0 (1.00)	1673.2 (1.00)	440.2 (1.00)
	T-SAH$_0$	9.1	154.1	140.7	2.7	2.8	1185.2 (0.74)	1342.0 (0.80)	422.6 (0.96)
	T-SAH$_2$	9.3	149.6	138.4	2.7	2.8	1177.4 (0.73)	1297.5 (0.78)	426.0 (0.97)
	T-SAH$_4$	10.5	154.0	139.7	2.7	2.8	1273.1 (0.79)	1390.3 (0.83)	440.8 (1.00)
SOBB 64	Rebuild	54.0	124.3	118.6	2.4	2.7	1688.7 (1.00)	1721.6 (1.00)	447.0 (1.00)
	T-SAH$_0$	14.0	147.2	135.1	2.4	2.7	1270.6 (0.75)	1386.6 (0.81)	432.6 (0.97)
	T-SAH$_2$	14.0	150.1	135.8	2.3	2.7	1290.4 (0.76)	1326.3 (0.77)	431.8 (0.97)
	T-SAH$_4$	13.8	149.5	135.4	2.4	2.7	1223.9 (0.72)	1359.5 (0.79)	430.0 (0.96)
Balls 16									
AABB	Rebuild	0.4	148.6	127.5	6.7	4.6	1965.6 (1.00)	970.0 (1.00)	565.6 (1.00)
	T-SAH$_0$	1.1	299.0	259.0	120.6	4.6	980.9 (0.50)	557.7 (0.57)	378.1 (0.67)
	T-SAH$_2$	1.4	310.2	258.1	120.9	4.6	986.5 (0.50)	566.6 (0.58)	384.9 (0.68)
	T-SAH$_4$	1.4	316.5	260.9	123.5	4.6	967.5 (0.49)	558.1 (0.58)	374.8 (0.66)
14-DOP	Rebuild	1.0	152.3	132.4	7.6	3.1	1894.3 (1.00)	864.0 (1.00)	551.1 (1.00)
	T-SAH$_0$	1.5	298.6	256.0	114.9	3.1	862.7 (0.46)	505.4 (0.58)	336.8 (0.61)
	T-SAH$_2$	1.8	314.9	261.7	120.4	3.1	816.2 (0.43)	491.2 (0.57)	328.0 (0.60)
	T-SAH$_4$	1.6	300.8	259.9	118.1	3.1	833.2 (0.44)	504.4 (0.58)	331.7 (0.60)
SOBB 32	Rebuild	18.2	189.3	164.2	2.2	3.2	1717.8 (1.00)	795.2 (1.00)	519.5 (1.00)
	T-SAH$_0$	15.7	384.2	329.8	38.0	3.1	725.7 (0.42)	463.7 (0.58)	311.2 (0.60)
	T-SAH$_2$	17.3	402.0	331.4	39.7	3.1	720.0 (0.42)	456.5 (0.57)	306.8 (0.59)
	T-SAH$_4$	15.9	395.7	329.0	38.4	3.1	726.7 (0.42)	462.1 (0.58)	308.7 (0.59)
SOBB 48	Rebuild	25.3	187.4	161.8	2.3	2.9	1764.9 (1.00)	830.8 (1.00)	529.4 (1.00)
	T-SAH$_0$	23.1	363.2	322.5	39.6	2.9	777.4 (0.44)	483.9 (0.58)	320.5 (0.61)
	T-SAH$_2$	27.2	383.0	321.6	41.3	2.9	769.7 (0.44)	494.0 (0.59)	324.3 (0.61)
	T-SAH$_4$	26.3	385.7	323.6	41.5	2.9	769.1 (0.44)	506.5 (0.61)	336.1 (0.63)
SOBB 64	Rebuild	41.2	183.4	159.0	2.2	2.8	1814.8 (1.00)	906.5 (1.00)	592.9 (1.00)
	T-SAH$_0$	33.1	364.3	316.4	40.0	2.8	779.1 (0.43)	509.8 (0.56)	337.7 (0.57)
	T-SAH$_2$	46.3	366.6	315.1	40.7	2.8	808.5 (0.45)	506.9 (0.56)	333.8 (0.56)
	T-SAH$_4$	34.6	369.5	317.1	38.6	2.8	767.9 (0.42)	501.0 (0.55)	331.4 (0.56)

Tabulka 4.12 – Pokračování tabulky z předchozí stránky

Těleso	Alg.	Čas stavby [s]	Průměr SAH	Max SAH	Průměr EPO	Průměr LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Breaking lion									
AABB	Rebuild	4.3	607.2	484.3	104.5	30.9	721.2 (1.00)	504.8 (1.00)	204.6 (1.00)
	T-SAH$_0$	9.2	646.6	513.6	122.5	30.2	691.8 (0.96)	490.3 (0.97)	196.2 (0.96)
	T-SAH$_2$	10.4	636.3	512.7	122.7	30.3	664.9 (0.92)	479.8 (0.95)	196.4 (0.96)
	T-SAH$_4$	11.3	627.5	513.7	123.0	30.2	638.7 (0.89)	471.6 (0.93)	192.5 (0.94)
14-DOP	Rebuild	11.5	472.9	377.8	67.6	15.6	888.0 (1.00)	535.5 (1.00)	240.7 (1.00)
	T-SAH$_0$	17.0	504.3	401.1	83.0	15.9	826.9 (0.93)	510.4 (0.95)	235.2 (0.98)
	T-SAH$_2$	15.0	495.8	400.7	83.9	15.8	759.3 (0.86)	499.0 (0.93)	227.8 (0.95)
	T-SAH$_4$	18.3	488.6	401.4	83.7	15.7	810.9 (0.91)	498.3 (0.93)	228.2 (0.95)
SOBB 32	Rebuild	439.9	548.6	441.0	11.5	15.1	889.2 (1.00)	582.9 (1.00)	263.5 (1.00)
	T-SAH$_0$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
	T-SAH$_2$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
	T-SAH$_4$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
SOBB 48	Rebuild	532.2	521.7	419.0	10.0	13.6	947.7 (1.00)	626.4 (1.00)	279.5 (1.00)
	T-SAH$_0$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
	T-SAH$_2$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
	T-SAH$_4$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
SOBB 64	Rebuild	788.7	502.8	403.6	9.2	12.8	1033.3 (1.00)	660.2 (1.00)	292.3 (1.00)
	T-SAH$_0$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
	T-SAH$_2$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
	T-SAH$_4$	xxx	xxx	xxx	xxx	xxx	xxx (xxx)	xxx (xxx)	xxx (xxx)
Cloth ball									
AABB	Rebuild	0.3	88.9	72.1	16.3	5.4	1850.6 (1.00)	1498.6 (1.00)	761.2 (1.00)
	T-SAH$_0$	0.4	138.2	88.1	42.4	6.0	1469.7 (0.79)	1401.3 (0.94)	772.2 (1.01)
	T-SAH$_2$	0.5	127.4	88.1	40.8	5.9	1493.3 (0.81)	1364.8 (0.91)	726.3 (0.95)
	T-SAH$_4$	0.6	120.4	89.2	41.0	5.9	1525.1 (0.82)	1313.6 (0.88)	728.0 (0.96)
14-DOP	Rebuild	0.6	87.0	79.1	15.7	3.2	1920.3 (1.00)	1310.3 (1.00)	731.0 (1.00)
	T-SAH$_0$	0.6	125.8	90.7	36.4	3.5	1620.6 (0.84)	1236.0 (0.94)	712.1 (0.97)
	T-SAH$_2$	0.5	120.2	90.0	34.6	3.4	1634.7 (0.85)	1223.8 (0.93)	701.8 (0.96)
	T-SAH$_4$	0.5	114.5	89.7	33.0	3.4	1586.1 (0.83)	1255.9 (0.96)	733.8 (1.00)
SOBB 32	Rebuild	11.1	106.5	91.5	4.6	3.2	1787.4 (1.00)	1311.9 (1.00)	740.4 (1.00)
	T-SAH$_0$	2.3	160.2	107.0	9.4	3.5	1362.2 (0.76)	1182.4 (0.90)	730.9 (0.99)
	T-SAH$_2$	3.0	150.6	106.5	8.4	3.5	1332.3 (0.75)	1122.3 (0.86)	702.9 (0.95)
	T-SAH$_4$	3.7	141.7	107.2	8.1	3.5	1417.7 (0.79)	1164.9 (0.89)	718.3 (0.97)
SOBB 48	Rebuild	18.5	103.6	88.8	4.4	2.9	2102.1 (1.00)	1384.0 (1.00)	780.0 (1.00)
	T-SAH$_0$	3.3	155.4	103.4	10.2	3.2	1425.8 (0.68)	1201.5 (0.87)	690.4 (0.89)
	T-SAH$_2$	4.0	146.4	102.2	8.9	3.1	1540.3 (0.73)	1122.8 (0.81)	648.1 (0.83)
	T-SAH$_4$	4.9	137.8	103.1	8.8	3.1	1706.0 (0.81)	1198.1 (0.87)	707.7 (0.91)
SOBB 64	Rebuild	31.7	101.0	87.8	4.3	2.8	2104.6 (1.00)	1432.5 (1.00)	777.7 (1.00)
	T-SAH$_0$	5.1	151.9	101.2	10.3	3.1	1536.1 (0.73)	1243.1 (0.87)	712.2 (0.92)
	T-SAH$_2$	5.8	143.2	100.4	8.9	3.0	1539.7 (0.73)	1145.9 (0.80)	665.5 (0.86)
	T-SAH$_4$	7.1	135.4	100.8	8.8	3.0	1585.2 (0.75)	1232.2 (0.86)	727.8 (0.94)

Tabulka 4.12 – Pokračování tabulky z předchozí stránky

Těleso	Alg.	Čas stavby [s]	Průměr SAH	Max SAH	Průměr EPO	Průměr LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Exploding dragon									
AABB	Rebuild	0.5	133.3	58.5	14.7	4.8	2280.7 (1.00)	411.3 (1.00)	322.9 (1.00)
	T-SAH$_0$	1.9	360.3	136.9	143.1	4.9	1063.7 (0.47)	250.2 (0.61)	190.0 (0.59)
	T-SAH$_2$	2.1	220.0	104.6	159.0	4.9	900.6 (0.39)	262.8 (0.64)	217.5 (0.67)
	T-SAH$_4$	2.2	191.8	106.6	212.7	4.9	843.5 (0.37)	266.7 (0.65)	215.9 (0.67)
14-DOP	Rebuild	1.2	134.2	58.4	14.6	2.8	1915.6 (1.00)	353.7 (1.00)	309.6 (1.00)
	T-SAH$_0$	3.5	359.7	134.7	149.4	2.9	822.6 (0.43)	203.6 (0.58)	154.4 (0.50)
	T-SAH$_2$	3.1	213.8	101.2	160.1	2.9	715.5 (0.37)	225.2 (0.64)	182.2 (0.59)
	T-SAH$_4$	3.2	184.3	102.4	216.0	2.9	668.7 (0.35)	234.6 (0.66)	190.7 (0.62)
SOBB 32	Rebuild	28.6	162.9	74.0	5.9	2.9	1830.6 (1.00)	346.4 (1.00)	289.4 (1.00)
	T-SAH$_0$	29.7	452.1	173.0	25.9	2.9	771.4 (0.42)	207.2 (0.60)	158.8 (0.55)
	T-SAH$_2$	27.8	267.3	129.7	52.2	2.9	670.5 (0.37)	229.0 (0.66)	187.2 (0.65)
	T-SAH$_4$	26.2	233.2	131.5	68.0	2.9	651.7 (0.36)	233.2 (0.67)	192.3 (0.66)
SOBB 48	Rebuild	40.5	158.5	71.9	5.6	2.6	1952.8 (1.00)	374.8 (1.00)	315.3 (1.00)
	T-SAH$_0$	43.0	446.1	170.4	30.2	2.6	815.9 (0.42)	223.8 (0.60)	168.7 (0.54)
	T-SAH$_2$	43.2	258.2	126.2	50.6	2.6	722.4 (0.37)	241.5 (0.64)	197.3 (0.63)
	T-SAH$_4$	39.1	223.0	126.0	71.6	2.6	713.9 (0.37)	249.5 (0.67)	205.4 (0.65)
SOBB 64	Rebuild	64.7	154.3	70.4	5.5	2.5	2009.7 (1.00)	393.6 (1.00)	320.9 (1.00)
	T-SAH$_0$	51.8	428.4	164.3	27.2	2.5	859.8 (0.43)	233.8 (0.59)	179.6 (0.56)
	T-SAH$_2$	56.1	253.1	122.5	49.3	2.5	778.0 (0.39)	260.8 (0.66)	206.9 (0.64)
	T-SAH$_4$	55.2	218.3	123.5	73.0	2.5	753.3 (0.37)	261.1 (0.66)	214.2 (0.67)
Fairy forest									
AABB	Rebuild	0.4	76.5	76.0	6.2	3.2	1470.0 (1.00)	1019.5 (1.00)	884.8 (1.00)
	T-SAH$_0$	0.7	76.7	76.1	6.3	3.2	1396.6 (0.95)	999.6 (0.98)	848.0 (0.96)
	T-SAH$_2$	0.7	76.7	76.1	6.3	3.2	1377.4 (0.94)	1009.5 (0.99)	849.8 (0.96)
	T-SAH$_4$	0.7	76.7	76.1	6.2	3.2	1369.4 (0.93)	997.7 (0.98)	854.4 (0.97)
14-DOP	Rebuild	1.0	93.0	92.4	5.6	2.1	1358.1 (1.00)	904.0 (1.00)	795.4 (1.00)
	T-SAH$_0$	1.0	93.4	92.6	5.8	2.1	1352.2 (1.00)	896.4 (0.99)	782.6 (0.98)
	T-SAH$_2$	1.0	93.3	92.5	5.8	2.1	1377.7 (1.01)	896.0 (0.99)	781.6 (0.98)
	T-SAH$_4$	1.0	93.4	92.6	5.8	2.1	1365.9 (1.01)	891.3 (0.99)	783.7 (0.99)
SOBB 32	Rebuild	20.5	99.3	98.5	2.5	2.1	1238.9 (1.00)	911.8 (1.00)	786.1 (1.00)
	T-SAH$_0$	8.2	99.3	98.4	2.4	2.1	1189.8 (0.96)	922.8 (1.01)	785.5 (1.00)
	T-SAH$_2$	7.8	99.3	98.5	2.5	2.1	1196.7 (0.97)	910.9 (1.00)	779.6 (0.99)
	T-SAH$_4$	7.8	99.4	98.5	2.5	2.1	1207.1 (0.97)	919.8 (1.01)	785.8 (1.00)
SOBB 48	Rebuild	33.6	99.6	98.8	2.3	2.1	1345.8 (1.00)	976.8 (1.00)	838.1 (1.00)
	T-SAH$_0$	9.3	99.8	99.0	2.1	2.1	1309.1 (0.97)	958.1 (0.98)	816.6 (0.97)
	T-SAH$_2$	9.7	99.7	98.9	2.1	2.1	1290.0 (0.96)	956.2 (0.98)	825.5 (0.99)
	T-SAH$_4$	9.2	99.7	99.0	2.1	2.1	1275.9 (0.95)	959.0 (0.98)	824.1 (0.98)
SOBB 64	Rebuild	58.7	98.3	97.6	2.2	1.9	1339.0 (1.00)	996.9 (1.00)	863.7 (1.00)
	T-SAH$_0$	13.0	98.5	97.6	2.1	1.9	1352.5 (1.01)	1000.6 (1.00)	855.1 (0.99)
	T-SAH$_2$	12.7	98.6	97.6	2.1	1.9	1349.2 (1.01)	997.1 (1.00)	858.1 (0.99)
	T-SAH$_4$	14.9	98.4	97.5	2.1	1.9	1347.5 (1.01)	1000.9 (1.00)	860.2 (1.00)

Tabulka 4.12 – Pokračování tabulky z předchozí stránky

Těleso	Alg.	Čas stavby [s]	Průměr SAH	Max SAH	Průměr EPO	Průměr LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Funnel									
AABB	Rebuild	0.3	28.8	25.5	4.1	4.1	1571.4 (1.00)	1967.6 (1.00)	1392.8 (1.00)
	T-SAH$_0$	0.3	29.7	27.5	9.2	4.2	2427.9 (1.55)	2722.6 (1.38)	1439.0 (1.03)
	T-SAH$_2$	0.3	29.5	27.3	9.2	4.2	2199.9 (1.40)	2790.9 (1.42)	1562.1 (1.12)
	T-SAH$_4$	0.2	29.6	27.3	9.2	4.2	2439.5 (1.55)	2856.1 (1.45)	1503.4 (1.08)
14-DOP	Rebuild	0.4	34.1	30.7	3.6	2.8	1894.9 (1.00)	2309.5 (1.00)	1629.3 (1.00)
	T-SAH$_0$	0.3	35.0	32.8	9.2	2.7	2022.8 (1.07)	2403.5 (1.04)	1363.2 (0.84)
	T-SAH$_2$	0.3	35.1	32.8	9.3	2.7	2068.3 (1.09)	2432.7 (1.05)	1487.9 (0.91)
	T-SAH$_4$	0.3	35.0	32.7	9.2	2.7	2267.6 (1.20)	2371.7 (1.03)	1479.9 (0.91)
SOBB 32	Rebuild	6.2	38.2	34.1	0.9	2.6	2620.1 (1.00)	2711.0 (1.00)	1666.0 (1.00)
	T-SAH$_0$	1.1	38.7	36.3	1.1	2.6	1565.3 (0.60)	1929.4 (0.71)	1253.2 (0.75)
	T-SAH$_2$	1.1	38.8	36.4	1.0	2.6	1483.5 (0.57)	1747.2 (0.64)	1146.3 (0.69)
	T-SAH$_4$	1.1	38.8	36.4	1.0	2.6	1377.0 (0.53)	1975.6 (0.73)	1286.0 (0.77)
SOBB 48	Rebuild	11.7	38.1	33.8	0.9	2.4	2762.4 (1.00)	2793.1 (1.00)	1692.2 (1.00)
	T-SAH$_0$	1.9	38.2	35.9	0.9	2.4	1603.5 (0.58)	2010.8 (0.72)	1272.3 (0.75)
	T-SAH$_2$	1.9	38.1	35.9	1.0	2.4	1714.3 (0.62)	2004.0 (0.72)	1289.6 (0.76)
	T-SAH$_4$	2.0	38.3	36.0	1.0	2.4	1452.7 (0.53)	1768.0 (0.63)	1190.7 (0.70)
SOBB 64	Rebuild	18.4	37.4	33.5	0.8	2.3	3081.4 (1.00)	2882.1 (1.00)	1736.4 (1.00)
	T-SAH$_0$	2.7	37.6	35.4	0.9	2.3	1700.0 (0.55)	2056.9 (0.71)	1285.7 (0.74)
	T-SAH$_2$	2.8	37.7	35.5	0.9	2.3	1646.4 (0.53)	2008.4 (0.70)	1312.7 (0.76)
	T-SAH$_4$	3.0	37.7	35.5	1.0	2.3	1759.8 (0.57)	1835.9 (0.64)	1150.1 (0.66)
Hand									
AABB	Rebuild	0.4	96.2	89.7	12.0	2.7	1746.0 (1.00)	530.5 (1.00)	495.2 (1.00)
	T-SAH$_0$	0.2	97.0	92.1	13.9	2.8	1423.7 (0.82)	608.1 (1.15)	463.8 (0.94)
	T-SAH$_2$	0.4	97.1	92.2	14.0	2.8	1507.6 (0.86)	557.1 (1.05)	463.2 (0.94)
	T-SAH$_4$	0.3	96.9	92.1	13.9	2.8	1777.6 (1.02)	575.0 (1.08)	481.0 (0.97)
14-DOP	Rebuild	0.6	97.1	91.2	10.8	1.8	2149.0 (1.00)	758.4 (1.00)	547.6 (1.00)
	T-SAH$_0$	0.3	97.4	92.4	11.8	1.8	2238.2 (1.04)	724.5 (0.96)	535.4 (0.98)
	T-SAH$_2$	0.4	97.8	92.7	11.9	1.8	2337.9 (1.09)	685.5 (0.90)	501.0 (0.91)
	T-SAH$_4$	0.3	97.7	92.7	11.9	1.8	2122.9 (0.99)	671.2 (0.89)	513.6 (0.94)
SOBB 32	Rebuild	6.9	115.4	109.2	3.0	1.8	3479.0 (1.00)	1010.7 (1.00)	635.3 (1.00)
	T-SAH$_0$	1.2	117.2	111.6	3.2	1.8	1771.4 (0.51)	423.5 (0.42)	444.2 (0.70)
	T-SAH$_2$	1.3	116.9	111.2	3.3	1.8	1441.0 (0.41)	334.8 (0.33)	342.3 (0.54)
	T-SAH$_4$	1.5	116.5	111.1	3.3	1.8	1395.4 (0.40)	337.8 (0.33)	357.8 (0.56)
SOBB 48	Rebuild	13.5	108.4	102.5	2.4	1.6	3860.7 (1.00)	1087.9 (1.00)	553.1 (1.00)
	T-SAH$_0$	2.3	109.4	104.5	2.5	1.6	1436.9 (0.37)	400.4 (0.37)	387.0 (0.70)
	T-SAH$_2$	2.2	109.3	104.5	2.5	1.6	1177.2 (0.30)	356.6 (0.33)	363.6 (0.66)
	T-SAH$_4$	2.3	109.4	104.5	2.5	1.6	1295.9 (0.34)	382.1 (0.35)	394.0 (0.71)
SOBB 64	Rebuild	20.0	105.9	100.5	2.3	1.5	3583.9 (1.00)	1055.6 (1.00)	588.4 (1.00)
	T-SAH$_0$	3.6	106.4	102.0	2.4	1.5	1319.5 (0.37)	387.2 (0.37)	411.2 (0.70)
	T-SAH$_2$	3.7	106.7	102.1	2.4	1.5	1296.8 (0.36)	409.0 (0.39)	401.0 (0.68)
	T-SAH$_4$	3.7	106.6	102.0	2.4	1.5	1442.6 (0.40)	364.2 (0.34)	386.8 (0.66)

Tabulka 4.12 – Pokračování tabulky z předchozí stránky

Těleso	Alg.	Čas stavby [s]	Průměr SAH	Max SAH	Průměr EPO	Průměr LCV	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
Meme party									
AABB	Rebuild	1.6	131.8	128.5	16.9	6.5	632.4 (1.00)	2927.4 (1.00)	622.7 (1.00)
	T-SAH$_0$	4.6	135.9	132.7	19.1	6.5	627.4 (0.99)	2818.0 (0.96)	600.5 (0.96)
	T-SAH$_2$	5.6	136.2	132.9	18.3	6.6	592.0 (0.94)	2810.6 (0.96)	590.0 (0.95)
	T-SAH$_4$	4.4	133.8	130.8	18.8	6.4	617.3 (0.98)	2842.3 (0.97)	596.0 (0.96)
14-DOP	Rebuild	5.1	140.7	138.0	16.6	4.6	918.7 (1.00)	3108.3 (1.00)	744.4 (1.00)
	T-SAH$_0$	7.5	145.5	141.1	18.3	4.5	860.6 (0.94)	2986.7 (0.96)	707.2 (0.95)
	T-SAH$_2$	7.3	143.2	140.5	18.1	4.6	910.0 (0.99)	3071.0 (0.99)	692.4 (0.93)
	T-SAH$_4$	9.5	144.6	141.2	18.7	4.5	790.4 (0.86)	3001.1 (0.97)	699.2 (0.94)
SOBB 32	Rebuild	183.1	154.7	150.6	5.5	4.3	920.4 (1.00)	3002.4 (1.00)	758.7 (1.00)
	T-SAH$_0$	65.4	155.5	152.2	6.0	4.2	909.1 (0.99)	2924.2 (0.97)	711.6 (0.94)
	T-SAH$_2$	60.0	156.5	153.5	5.5	4.3	875.5 (0.95)	2906.3 (0.97)	704.7 (0.93)
	T-SAH$_4$	63.3	155.2	151.7	4.5	4.2	892.8 (0.97)	2935.8 (0.98)	717.5 (0.95)
SOBB 48	Rebuild	229.1	153.2	149.1	5.5	4.0	1007.4 (1.00)	3113.4 (1.00)	800.4 (1.00)
	T-SAH$_0$	86.8	153.4	150.4	6.6	4.0	975.9 (0.97)	3066.5 (0.98)	742.2 (0.93)
	T-SAH$_2$	70.7	153.4	150.2	5.7	3.9	1020.6 (1.01)	3064.3 (0.98)	757.0 (0.95)
	T-SAH$_4$	86.0	155.5	151.9	5.2	4.0	993.3 (0.99)	3039.1 (0.98)	754.8 (0.94)
SOBB 64	Rebuild	332.3	152.3	148.9	5.7	3.9	1047.4 (1.00)	3205.8 (1.00)	822.5 (1.00)
	T-SAH$_0$	297.9	155.0	151.0	5.0	3.9	1018.7 (0.97)	3111.1 (0.97)	789.4 (0.96)
	T-SAH$_2$	393.4	153.0	150.4	6.0	3.9	1047.5 (1.00)	3084.6 (0.96)	776.6 (0.94)
	T-SAH$_4$	289.3	155.7	151.4	5.7	3.9	1065.1 (1.02)	3113.6 (0.97)	771.4 (0.94)
Transformer									
AABB	Rebuild	0.9	50.1	49.5	6.3	7.1	1901.3 (1.00)	3612.8 (1.00)	1146.5 (1.00)
	T-SAH$_0$	2.1	51.1	50.3	7.9	7.3	1646.7 (0.87)	3558.8 (0.99)	1091.4 (0.95)
	T-SAH$_2$	2.2	51.0	50.2	7.9	7.4	1436.1 (0.76)	3570.4 (0.99)	1065.9 (0.93)
	T-SAH$_4$	2.8	51.0	50.2	7.8	7.4	1639.5 (0.86)	3464.3 (0.96)	1075.1 (0.94)
14-DOP	Rebuild	2.2	67.2	66.4	7.0	3.9	1595.6 (1.00)	2822.7 (1.00)	1050.0 (1.00)
	T-SAH$_0$	3.4	68.2	67.0	8.7	4.1	1256.6 (0.79)	2715.7 (0.96)	924.8 (0.88)
	T-SAH$_2$	3.9	68.2	67.0	8.8	4.1	1105.2 (0.69)	2677.1 (0.95)	869.2 (0.83)
	T-SAH$_4$	4.2	68.2	67.0	8.6	4.1	1235.5 (0.77)	2693.7 (0.95)	911.9 (0.87)
SOBB 32	Rebuild	63.2	70.1	69.0	1.9	3.5	1574.3 (1.00)	2696.8 (1.00)	1004.5 (1.00)
	T-SAH$_0$	23.8	71.8	70.3	1.9	3.6	1118.4 (0.71)	2574.4 (0.95)	871.6 (0.87)
	T-SAH$_2$	26.8	71.6	70.2	1.8	3.6	1155.0 (0.73)	2568.0 (0.95)	884.1 (0.88)
	T-SAH$_4$	31.4	71.0	69.9	1.9	3.6	989.3 (0.63)	2510.8 (0.93)	857.6 (0.85)
SOBB 48	Rebuild	85.9	69.9	68.9	1.9	3.1	1740.1 (1.00)	2745.4 (1.00)	1014.9 (1.00)
	T-SAH$_0$	32.7	71.5	69.9	1.9	3.3	1299.8 (0.75)	2606.1 (0.95)	915.9 (0.90)
	T-SAH$_2$	35.3	71.4	69.9	1.8	3.3	1293.7 (0.74)	2553.5 (0.93)	916.1 (0.90)
	T-SAH$_4$	39.9	71.1	69.7	1.7	3.3	1455.8 (0.84)	2646.9 (0.96)	931.9 (0.92)
SOBB 64	Rebuild	130.4	69.9	68.8	1.9	3.0	1830.2 (1.00)	2824.2 (1.00)	1031.4 (1.00)
	T-SAH$_0$	43.6	71.1	69.5	1.8	3.1	1346.6 (0.74)	2773.0 (0.98)	927.8 (0.90)
	T-SAH$_2$	49.2	70.9	69.6	1.9	3.1	1381.6 (0.75)	2714.2 (0.96)	921.5 (0.89)
	T-SAH$_4$	64.3	70.6	69.3	2.0	3.1	1457.3 (0.80)	2694.9 (0.95)	939.7 (0.91)

Těleso	Algoritmus	Čas stavby	Primary	AO	Diffuse
AABB	Rebuild	1.00 [1]	1.00 [5]	1.00 [4]	1.00 [5]
	T-SAH$_0$	2.07 [2]	0.87 [10]	0.94 [6]	0.90 [8]
	T-SAH$_2$	2.33 [3]	0.83 [12]	0.94 [8]	0.91 [6]
	T-SAH$_4$	2.36 [5]	0.88 [9]	0.94 [7]	0.90 [7]
14-DOP	Rebuild	2.35 [4]	1.06 [4]	0.98 [5]	1.02 [4]
	T-SAH$_0$	3.19 [7]	0.90 [7]	0.88 [9]	0.88 [10]
	T-SAH$_2$	3.16 [6]	0.90 [6]	0.88 [10]	0.87 [12]
	T-SAH$_4$	3.29 [8]	0.88 [8]	0.88 [11]	0.89 [9]
SOBB 32	Rebuild	51.69 [15]	1.15 [3]	1.03 [3]	1.02 [3]
	T-SAH$_0$	24.65 [9]	0.78 [18]	0.76 [18]	0.84 [18]
	T-SAH$_2$	24.77 [10]	0.74 [19]	0.73 [20]	0.81 [20]
	T-SAH$_4$	25.33 [11]	0.73 [20]	0.75 [19]	0.83 [19]
SOBB 48	Rebuild	76.49 [19]	1.25 [2]	1.09 [2]	1.04 [2]
	T-SAH$_0$	34.33 [12]	0.80 [16]	0.78 [15]	0.84 [17]
	T-SAH$_2$	35.17 [13]	0.80 [17]	0.77 [17]	0.85 [16]
	T-SAH$_4$	36.16 [14]	0.81 [15]	0.78 [16]	0.86 [15]
SOBB 64	Rebuild	122.00 [20]	1.28 [1]	1.12 [1]	1.08 [1]
	T-SAH$_0$	58.90 [16]	0.83 [14]	0.80 [12]	0.88 [11]
	T-SAH$_2$	71.05 [18]	0.83 [13]	0.80 [13]	0.87 [13]
	T-SAH$_4$	63.54 [17]	0.85 [11]	0.79 [14]	0.87 [14]

Tabulka 4.13: Průměrné hodnoty poměrů metrik pro 9 animovaných scén z tabulky 4.12. Scéna breaking lion se nepoužívá kvůli chybějícím hodnotám pro SOBB. Levá hodnota je poměr k obalovému tělesu AABB optimalizovanému každý snímek. Pravá hodnota je pořadí kombinace tělesa a algoritmu pro danou metriku.

4.2.4 Stavba pomocí refitu z AABB

AABB má díky své jednoduchosti nejrychlejší optimalizaci pomocí PRBVH a to při porovnání s SOBB klidně více než stokrát. Komplexnější tělesa však stále mají výhodu ve své těsnosti, jen vázne jejich optimalizace. Při každé iteraci PRBVH se po přesunutí uzlů musí provést refit, aby byla tělesa platná. To stejné se provádí při změně snímku při použití T-SAH. Hierarchie se zde nemění, mění se jen geometrie scény. V tento moment můžeme místo refitování AABB zkusit použít jiné obalové těleso. Naměřené hodnoty po refitu s jiným tělesem ukazuje tabulka 4.14.

Zde jsou porovnány hierarchie vytvořené optimalizací daného obalového tělesa a hierarchie AABB, na kterou bylo těleso nafitováno. Můžeme zde vidět, že refitované SOBB značně předčí svou vlastní optimalizaci, i přestože metriky ukazují, že optimalizovaná varianta by měla být lepší. To si vysvětluji nepoužitím ořezávání při optimalizaci SOBB. Má implementace vlastně optimalizuje špatnou SAH metriku. Dalším faktorem je, že každá SOBB varianta dokáže napodobit AABB, a tudíž i když se neoptimalizovalo pro SOBB, tak optimalizace AABB je dostačující a předčí špatně optimalizovanou hierarchii. Dá se říci, že optimalizace AABB představuje něco jako heuristiku pro SOBB. Tento výsledek jsem očekával.

Překvapilo mě, že pro scénu Powerplant nastala podobná situace i pro 14-DOP. Optimalizovaný 14-DOP u ostatních scén předčí refitovanou variantu. U Powerplant je výkon primárních a AO paprsků o více jak 10% lepší při refitu. Může nastat situace, že optimalizování AABB dosáhne jiného lokálního minima pro 14-DOP, kam se běžná optimalizace 14-DOPu nedostane kvůli greedy vybírání pozice uzlů. Tento rozdíl by ale, podle mého názoru, neměl být tak velký. Větší rozdíl u primárních paprsků může být také způsoben ještě plně "nenahřátým"GPU, jelikož primární paprsky se počítají jako první. Tento test byl prováděn až skoro jako poslední, což tuto variantu vylučuje.

Těleso	**SAH**	**EPO**	**LCV**	Primary [Mray/s]	AO [Mray/s]	Diffuse [Mray/s]
			Hairball			
AABB	985.0 (xxx)	79.3 (xxx)	67.8 (xxx)	465.2 (XXX)	218.3 (xxx)	194.5 (xxx)
14-DOP	**792.7 (1.00)**	**51.1 (1.00)**	**27.6 (1.00)**	**548.6 (1.00)**	**254.6 (1.00)**	**235.2 (1.00)**
AABB → 14-DOP	815.2 (1.03)	53.9 (1.05)	30.3 (1.10)	546.1 (1.00)	235.3 (0.92)	226.7 (0.96)
SOBB 32	**937.4 (1.00)**	**12.7 (1.00)**	33.1 (1.00)	465.2 (1.00)	220.6 (1.00)	203.8 (1.00)
AABB → SOBB 32	955.3 (1.02)	13.0 (1.02)	**29.7 (0.90)**	**482.8 (1.04)**	**228.2 (1.03)**	**209.6 (1.03)**
SOBB 48	**888.1 (1.00)**	**11.3 (1.00)**	30.6 (1.00)	514.3 (1.00)	249.5 (1.00)	220.2 (1.00)
AABB → SOBB 48	909.7 (1.02)	11.5 (1.02)	**26.7 (0.87)**	**526.6 (1.02)**	**253.3 (1.02)**	**225.1 (1.02)**
SOBB 64	**860.2 (1.00)**	**10.3 (1.00)**	28.9 (1.00)	538.3 (1.00)	**262.2 (1.00)**	232.6 (1.00)
AABB → SOBB 64	882.7 (1.03)	10.7 (1.04)	**24.6 (0.85)**	**552.0 (1.03)**	257.5 (0.98)	**236.1 (1.01)**
			San Miguel			
AABB	125.5 (xxx)	20.5 (xxx)	9.1 (xxx)	536.6 (xxx)	688.5 (xxx)	326.6 (xxx)
14-DOP	**144.4 (1.00)**	23.0 (1.00)	**6.0 (1.00)**	**471.0 (1.00)**	581.8 (1.00)	**277.0 (1.00)**
AABB → 14-DOP	147.7 (1.02)	**22.9 (1.00)**	**6.0 (1.00)**	470.3 (1.00)	**583.2 (1.00)**	274.8 (0.99)
SOBB 32	**161.7 (1.00)**	7.4 (1.00)	**6.1 (1.00)**	386.9 (1.00)	555.6 (1.00)	**281.7 (1.00)**
AABB → SOBB 32	162.7 (1.01)	**7.2 (0.97)**	6.3 (1.03)	**436.6 (1.13)**	**578.8 (1.04)**	269.4 (0.96)
SOBB 48	**161.2 (1.00)**	**6.6 (1.00)**	**5.7 (1.00)**	378.4 (1.00)	599.7 (1.00)	288.6 (1.00)
AABB → SOBB 48	160.8 (1.00)	6.7 (1.01)	6.0 (1.04)	**475.7 (1.26)**	**614.1 (1.02)**	**288.7 (1.00)**
SOBB 64	**157.9 (1.00)**	**6.1 (1.00)**	**5.6 (1.00)**	474.0 (1.00)	**641.3 (1.00)**	**309.6 (1.00)**
AABB → SOBB 64	160.2 (1.01)	6.7 (1.09)	5.8 (1.03)	**487.8 (1.03)**	596.1 (0.93)	295.9 (0.96)
			Powerplant			
AABB	68.7 (xxx)	42.2 (xxx)	24.5 (xxx)	375.7 (xxx)	746.5 (xxx)	479.2 (xxx)
14-DOP	**66.6 (1.00)**	50.6 (1.00)	20.9 (1.00)	700.7 (1.00)	638.6 (1.00)	421.8 (1.00)
AABB → 14-DOP	68.2 (1.02)	**50.1 (0.99)**	**20.5 (0.98)**	**871.1 (1.24)**	**726.6 (1.14)**	**449.8 (1.07)**
SOBB 32	**68.2 (1.00)**	**5.3 (1.00)**	**20.5 (1.00)**	603.3 (1.00)	**651.6 (1.00)**	468.3 (1.00)
AABB → SOBB 32	72.6 (1.00)	**5.3 (0.99)**	**20.5 (1.00)**	**810.6 (1.34)**	616.0 (0.95)	**492.0 (1.05)**
SOBB 48	**70.8 (1.00)**	5.2 (1.00)	**20.2 (1.00)**	806.7 (1.00)	643.5 (1.00)	472.5 (1.00)
AABB → SOBB 48	71.8 (1.01)	**5.2 (0.99)**	20.3 (1.01)	**832.7 (1.03)**	**654.0 (1.02)**	**497.9 (1.05)**
SOBB 64	**70.4 (1.00)**	**4.8 (1.00)**	**20.0 (1.00)**	669.5 (1.00)	693.5 (1.00)	450.5 (1.00)
AABB → SOBB 64	71.3 (1.01)	5.0 (1.05)	20.2 (1.01)	**834.2 (1.25)**	**710.3 (1.02)**	**489.6 (1.09)**
			Buddha			
AABB	157.2 (xxx)	10.2 (xxx)	4.6 (xxx)	1427.5 (xxx)	470.2 (xxx)	532.8 (xxx)
14-DOP	**167.6 (1.00)**	10.8 (1.00)	**2.8 (1.00)**	**1284.4 (1.00)**	442.6 (1.00)	473.4 (1.00)
AABB → 14-DOP	172.7 (1.03)	**10.7 (0.99)**	2.9 (1.01)	1275.5 (0.99)	**446.4 (1.01)**	**481.5 (1.02)**
SOBB 32	**195.1 (1.00)**	**2.7 (1.00)**	**2.7 (1.00)**	1230.3 (1.00)	446.9 (1.00)	**498.4 (1.00)**
AABB → SOBB 32	201.9 (1.03)	2.9 (1.06)	2.8 (1.02)	**1270.2 (1.03)**	**442.2 (0.99)**	487.7 (0.98)
SOBB 48	**191.6 (1.00)**	**2.6 (1.00)**	**2.6 (1.00)**	1384.8 (1.00)	**481.0 (1.00)**	**519.2 (1.00)**
AABB → SOBB 48	197.8 (1.03)	2.7 (1.06)	2.6 (1.02)	**1443.6 (1.04)**	454.6 (0.95)	516.2 (0.99)
SOBB 64	**187.8 (1.00)**	**2.5 (1.00)**	**2.4 (1.00)**	1389.3 (1.00)	**546.2 (1.00)**	**598.3 (1.00)**
AABB → SOBB 64	193.9 (1.03)	2.6 (1.06)	2.4 (1.01)	**1438.1 (1.04)**	478.1 (0.88)	527.3 (0.88)
			Conference			
AABB	83.6 (xxx)	15.5 (xxx)	4.7 (xxx)	1549.9 (xxx)	1582.3 (xxx)	979.0 (xxx)
14-DOP	105.0 (1.00)	19.3 (1.00)	4.0 (1.00)	**1393.3 (1.00)**	1462.6 **(1.00)**	**860.3 (1.00)**
AABB → 14-DOP	**103.6 (0.99)**	**17.6 (0.91)**	**4.0 (0.99)**	1334.4 (0.96)	1007.5 (0.69)	856.1 (1.00)
SOBB 32	**107.3 (1.00)**	**11.5 (1.00)**	**4.0 (1.00)**	**1465.4 (1.00)**	1623.8 **(1.00)**	**982.8 (1.00)**
AABB → SOBB 32	107.9 (1.01)	11.9 (1.03)	**4.0 (1.00)**	1383.2 (0.94)	1348.9 (0.83)	930.2 (0.95)
SOBB 48	**105.6 (1.00)**	12.8 (1.00)	**3.7 (1.00)**	**1634.1 (1.00)**	834.8 (1.00)	869.3 (1.00)
AABB → SOBB 48	106.6 (1.01)	**11.8 (0.92)**	3.7 (1.01)	1437.6 (0.88)	**1417.6 (1.70)**	**992.9 (1.14)**
SOBB 64	**105.4 (1.00)**	11.7 (1.00)	**3.6 (1.00)**	**1792.1 (1.00)**	1518.6 (1.00)	**1091.3 (1.00)**
AABB → SOBB 64	106.2 (1.01)	**11.6 (0.99)**	3.7 (1.02)	1564.0 (0.87)	**1702.9 (1.12)**	1031.9 (0.95)
			Fairy forest			
AABB	76.0 (xxx)	6.4 (xxx)	3.2 (xxx)	1465.5 (xxx)	981.0 (xxx)	865.1 (xxx)
14-DOP	**92.4 (1.00)**	**5.6 (1.00)**	**2.1 (1.00)**	**1365.8 (1.00)**	**802.0 (1.00)**	**759.0 (1.00)**
AABB → 14-DOP	93.0 (1.01)	5.7 (1.03)	2.1 (1.01)	1401.3 (1.03)	788.7 (0.98)	750.6 (0.99)
SOBB 32	**98.1 (1.00)**	**2.4 (1.00)**	2.1 (1.00)	**1278.4 (1.00)**	**906.9 (1.00)**	**766.1 (1.00)**
AABB → SOBB 32	99.3 (1.01)	2.5 (1.03)	**2.2 (1.02)**	1192.6 (0.93)	877.2 (0.97)	737.9 (0.96)
SOBB 48	**98.8 (1.00)**	**2.3 (1.00)**	**2.0 (1.00)**	**1407.5 (1.00)**	**877.5 (1.00)**	**796.0 (1.00)**
AABB → SOBB 48	99.4 (1.01)	2.6 (1.15)	2.1 (1.02)	1246.3 (0.89)	847.4 (0.97)	764.0 (0.96)
SOBB 64	**97.6 (1.00)**	**2.2 (1.00)**	**1.9 (1.00)**	**1399.7 (1.00)**	**949.8 (1.00)**	**825.5 (1.00)**
AABB → SOBB 64	99.0 (1.01)	2.7 (1.22)	2.0 (1.03)	1350.1 (0.96)	831.2 (0.88)	780.5 (0.95)

Tabulka 4.14: Porovnání tvorby obalových těles vytvořených optimalizací PRVBH oproti optimalizaci AABB a následném refitu na dané těleso. Pro SOBB použita strategie $O(n^2)$.

4.2.5 Porovnání výsledků s články

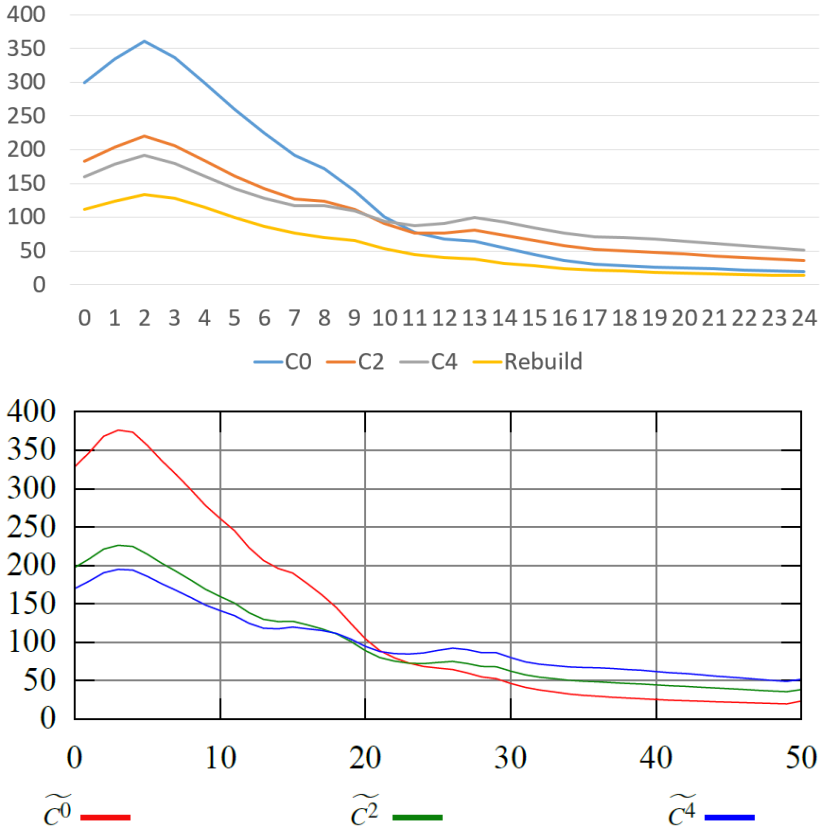
Abych ověřil své výsledky, porovnal jsem naměřené hodnoty SAH v tabulce 4.15 s hodnotami z původních článků. Výsledky se převážně shodují. Drobné ochylky se dají vysvětlit jiným natočením scény, které může značně ovlivnit LBVH metodu, jenž poté přesune PRBVH do jiného lokálního minima. SOBB nemohu porovnat, jelikož článek [10] obsahuje pouze výkon vrhání paprsků. Článek 14-DOPu [9] neoptimalizoval hierarchii pomocí PRBVH, ale využil PLOC. Můžeme vidět, že PRBVH dokázalo vylepšit SAH více. To je očekávatelné, jelikož PRBVH nespolehá na určení vzdálenosti Mortonovými kódy. Výsledky pro scénu San Miguel pro AABB nedpovídají hodnotám z článku, ale ostatní scény se zdají být v pořádku. Tato scéna nebyla součástí výchozího repozitáře. Stránka, ze které jsem ji stahoval, uváděla, že se jedná o verzi 2.0. Je tedy možné, že článek využíval ještě starší verzi, a proto se výsledky tolik liší. I po spuštění původního kódu PRBVH na GTX 1080, což je nejbližší dostupná karta k architektuře Maxwell, na které byly výsledky ve článku měřeny, byly naměřené hodnoty stejné. Z těchto důvodů se dá předpokládat, že problém je se scénou samotnou a ne s iimplementací.

Scéna	Těleso	Porovnávané metody	Naměřeno	Očekáváno
Budha	AABB	LBVH	191.5	204.00
Budha	AABB	PRBVH	157.2	159
Hairball	AABB	LBVH	1238.1	1237
Hairball	AABB	PRBVH	985.0	985
San Miguel	AABB	LBVH	210.8	299
San Miguel	AABB	PRBVH	125.5	100
San Miguel	14-DOP	PRBVH vs PLOC	144.4	172.8
Powerplant	14-DOP	PRBVH vs PLOC	66.6	74.9
Bistro interior	14-DOP	PRBVH vs PLOC	194.8	211.6

Tabulka 4.15: Porovnání naměřených hodnot SAHu s původními články.

Správnost implementace T-SAHu byla ověřena pomocí ukázkového grafu nacházejícího se v článku [24], který ho popisuje. Jedná se o naměřené hodnoty SAH pro scénu Exploding dragon s použitím AABB. Na obrázku 4.1 je vidět, že naměřené hodnoty a hodnoty článku jsou skoro totožné. Drobné rozdíly jsou přisuzovány většímu počtu naměřených snímků v grafu pocházejícím z článku.

Jelikož hodnoty naměřené pro PRBVH na statických scénách se zdají v pořádku a implementace T-SAH se nijak zvlášt neliší pro jednotlivá obalová tělesa, lze předpokládat, že implementace je korektní.



Obrázek 4.1: Hodnota SAH obalového tělesa AABB napříč snímky pro scénu Exploding dragon. Nahoře mnou naměřené hodnoty. Dole hodnoty z původního článku.

Porovnání AABB, 14-DOPu a SOBB 3

Obalové těleso SOBB umožňuje plně simulovat obalové těleso AABB. Jde tedy porovnat, jak velký rozdíl je mezi rychlostmi jejich výpočtu průsečíků s paprsky. K tomu stačí, aby množina tří normál odpovídala osám XYZ. To zajistí, že se budou vytvářet osově zarovnané kvádry, jelikož nebude existovat jiná kombinace normál. 14-DOP také na sebe dokáže vzít podobu AABB. Stačí při vytváření finální struktury `CudaKdopBVHNode` přepočítat vzdálenosti na diagonálách, aby odpovídaly rohům AABB. V tabulce 4.16 jsou tato obalová tělesa porovnána. Pokud je pro 14-DOP i SOBB konstanta C_t rovna 4.5 a SAH by byl ideální metrikou, pak by poměry SAHu a výkonů jednotlivých typů paprsků mezi AABB a ostatními tělesy měly být stejné. Je vidět, že poměry přibližně odpovídají. To svědčí o tom, že konstanty C_t byly zvoleny dobře. Může se zdát, že rozdíl mezi konstantami SOBB a AABB je až moc velký. Nesmíme však zapomínat na všechnu práci potřebnou k vypočtení průsečíku SOBB. Nejenže se musí načíst více dat, ale také je potřeba vypočítat skalární součiny k promítnutí paprsku na normálu. Tyto hodnoty jsou u AABB předpočtené, ale SOBB si to nemůže dovolit, jelikož se pokaždé použije jiná normála a ukládat tolik dat v registrech by snižovalo výkon. 14-DOP zase potřebuje otestovat více jak dvojnásobný počet os.

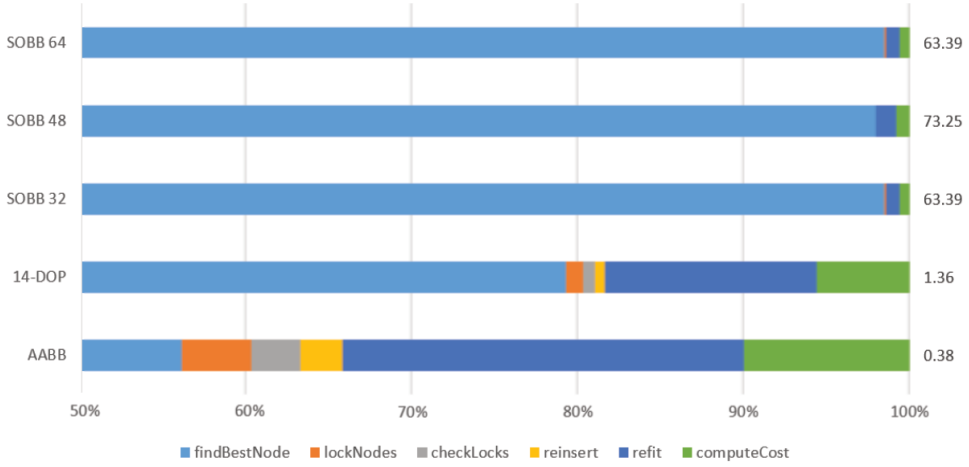
Scéna	**Těleso**	**SAH**	**Primary [Mray/s]**	**AO [Mray/s]**	**Diffuse [Mray/s]**	C_t
Hairball	AABB	1238.1 (0.77)	364.0 (1.00)	165.0 (1.00)	148.07 (1.00)	3
	SOBB 3	1606.5 (1.00)	247.0 (0.68)	116.4 (0.71)	103.57 (0.70)	4.5
	14-DOP	1606.5 (1.00)	174.86 (0.48)	85.96 (0.52)	76.32 (0.52)	4.5
San Miguel	AABB	210.8 (0.71)	343.8 (1.00)	445.3 (1.00)	205.68 (1.00)	3
	SOBB 3	298.7 (1.00)	195.12 (0.57)	269.67 (0.61)	118.32 (0.58)	4.5
	SOBB 3	298.7 (1.00)	204.3 (0.59)	268.1 (0.60)	123.73 (0.60)	4.5
Powerplant	AABB	114.1 (0.73)	311.5 (1.00)	1066.8 (1.00)	224.58 (1.00)	3
	SOBB 3	156.9 (1.00)	180.66 (0.58)	545.84 (0.51)	129.97 (0.58)	4.5
	SOBB 3	156.9 (1.00)	234.3 (0.75)	689.1 (0.65)	153.59 (0.68)	4.5

Tabulka 4.16: Porovnání výkonu AABB, 14-DOPu a SOBB 3 s totožným tvarem těles. Měřeno s přímým rozložením SOBB v paměti a bez rozděleného 14-DOPu.

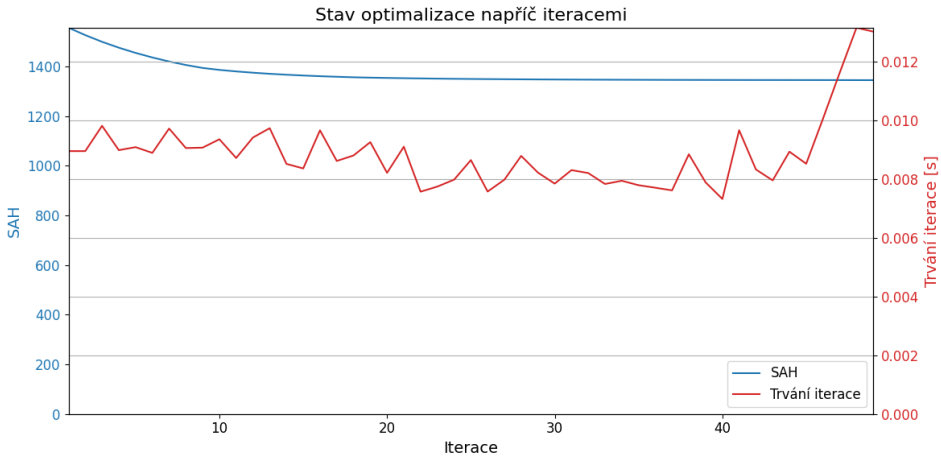
PRBVH

Podle měření PRBVH splnilo svůj účel a dokázalo značně vylepšit SAH i reálný výkon scén oproti LBVH. Při pohledu na změnu SAHu během optimalizace na obrázcích 4.3 a 4.4, můžeme usoudit, že ukončovací podmínka, která vyžaduje mít rozdíl po přesunu uzlů menší než 0.1, je vcelku přísná a podobných výsledků by se dalo dosáhnout i při použití jen poloviny iterací. Tato konstanta by mohla být nastavitelná například podle počáteční hodnoty SAHu, pokud by se PRBVH používalo ve více časově kritických situacích.

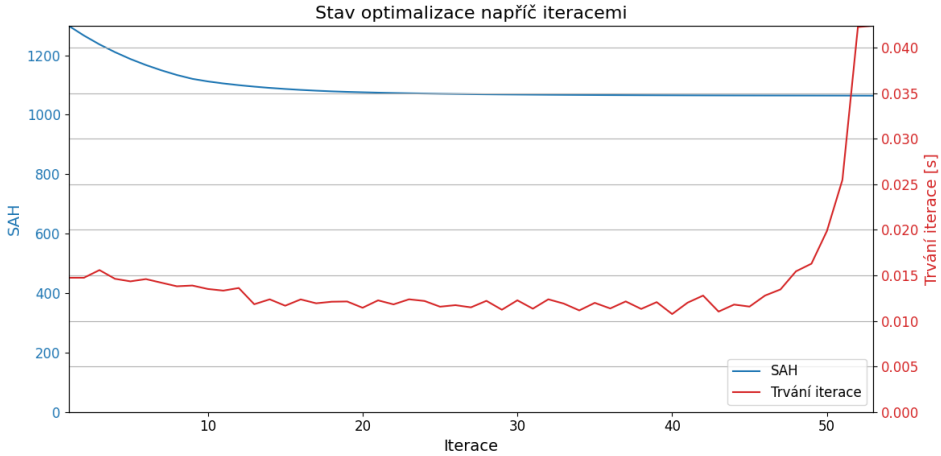
V grafu na obrázku 4.2 můžeme vidět poměry časů strávených v jednotlivých kernelech PRBVH. Nejvíce práce se odvede v kernelu findBestNode. SOBB stráví prakticky veškerý čas v tomto kernelu, jelikož jeho slučování SOBB a následný výpočet plochy vyžaduje výběr tří finálních normál, což je znatelně pomalejší než jednoduché min/max funkce pro AABB a 14-DOPu.



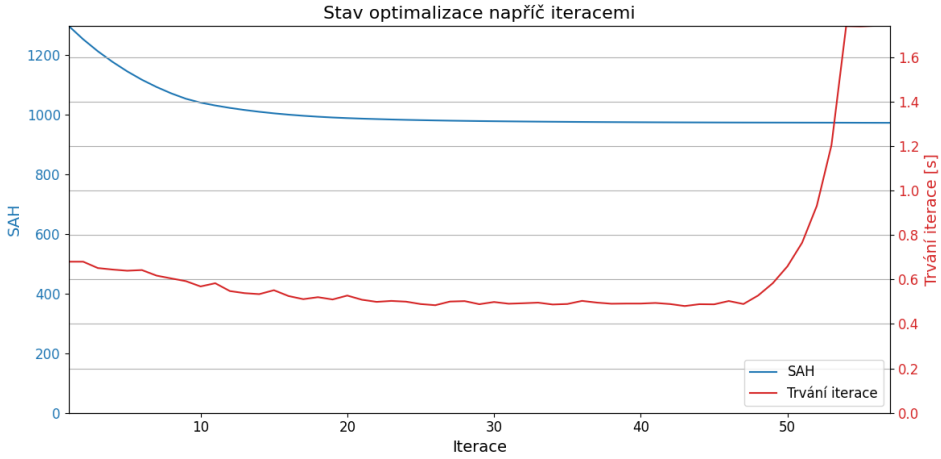
Obrázek 4.2: Poměr času stráveného v kernelech pro scénu San Miguel. Vpravo celkový čas v sekundách. Levý limit horizontální osy je 50%.



(a) : AABB

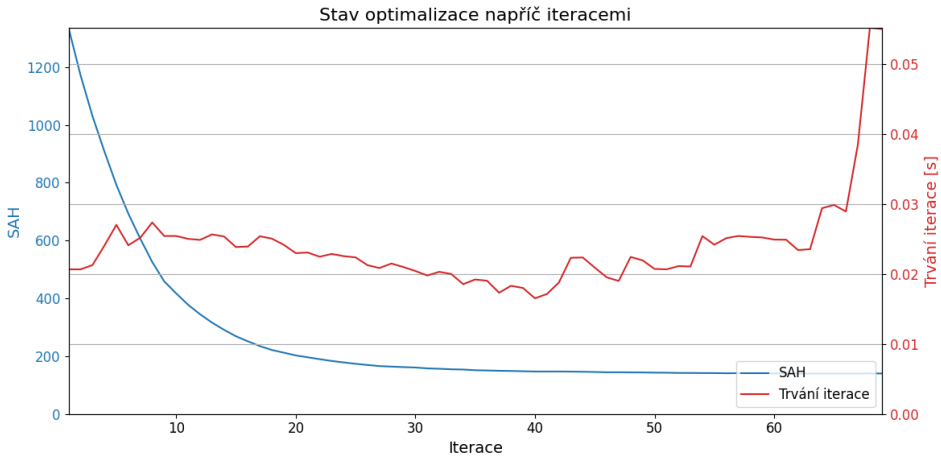


(b) : 14-DOP



(c) : SOBB 64

Obrázek 4.3: Vývoj hodnoty SAH během optimalizace pomocí PRBVH a trvání jednotlivých iterací pro scénu Hairball



Obrázek 4.4: Vývoj hodnoty T-SAH a trvání jednotlivých iterací během optimalizace scény Exploding dragon. Měřeno pro AABB a T-SAH C_0.

- **Parametr mod**

  PRBVH umožňuje nastavit parametr *mod*, který určuje, jak hustě se mají vybírat uzly k optimalizaci. V tabulce 4.17 můžeme vidět, že na větší scéně může mít parametr znatelný vliv na výkon a trvání optimalizace, a to především pro SOBB. Můžeme předpokládat, že při použití $mod = 1$ je kvůli greedy strategii vybírání optimalizace více hnáno do lokálního minima. Tomu naznačuje i počet iterací, který neroste lineárně s hodnotou *mod*. Naopak přeskakování některých uzlů zaručí, že se ne vždy provede právě optimální krok, což připomíná simulované žíhání. Volba parametru nemá skoro žádný vliv na výsledný SAH.

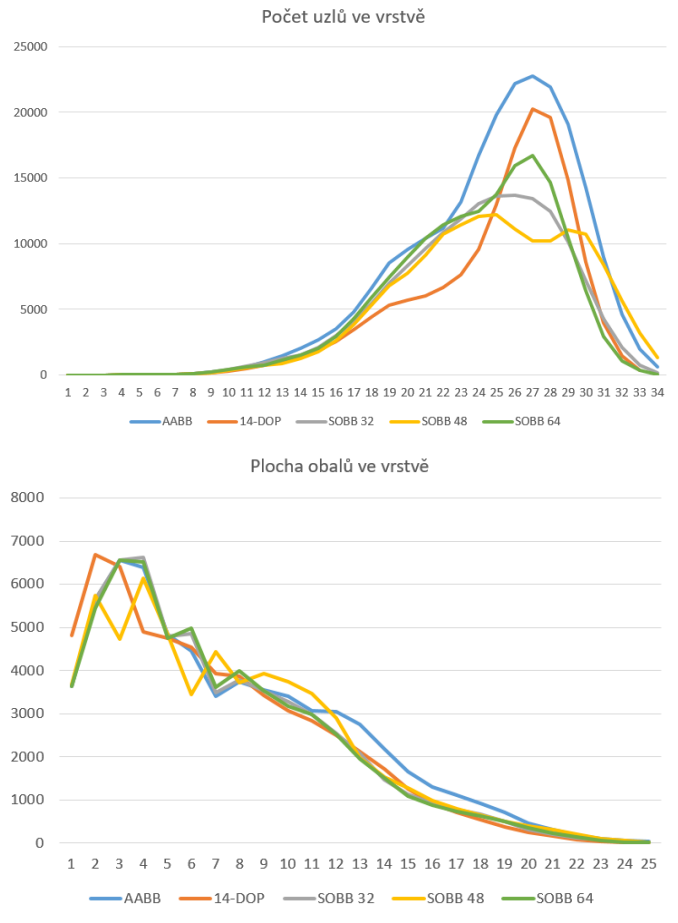
Scéna	*mod*	**Čas stavby [s]**	**SAH**	**Iterace**	**Primary [Mray/s]**	**AO [Mray/s]**	**Diffuse [Mray/s]**
Hairball	1	0.52 (1.00)	985.3	17	467.6 (1.00)	219.0 (1.00)	197.0 **(1.00)**
	4	0.63 (1.23)	**985.2**	29	468.9 (1.00)	**221.0 (1.01)**	196.3 (1.00)
	8	0.63 (1.22)	**985.2**	45	**469.0 (1.00)**	211.5 (0.97)	196.8 (1.00)
San Miguel	1	0.90 (1.00)	**126.3**	21	**528.1 (1.00)**	692.3 (1.00)	317.3 (1.00)
	4	0.78 (0.86)	126.9	27	526.4 (1.00)	**703.5 (1.02)**	321.5 (1.01)
	8	**0.72 (0.80)**	126.6	40	504.0 (0.95)	672.0 (0.97)	**322.7 (1.02)**
Powerplant	1	1.69 (1.00)	69.2	21	287.7 (1.00)	**1249.2 (1.00)**	352.8 (1.00)
	4	**0.97 (0.58)**	**68.7**	26	**334.8 (1.16)**	834.8 (0.67)	380.4 (1.08)
	8	1.04 (0.62)	69.0	35	323.6 (1.12)	899.3 (0.72)	**382.2 (1.08)**

Tabulka 4.17: Porovnání SAHu a výkonu AABB pro různé hodnoty parametru *mod*.

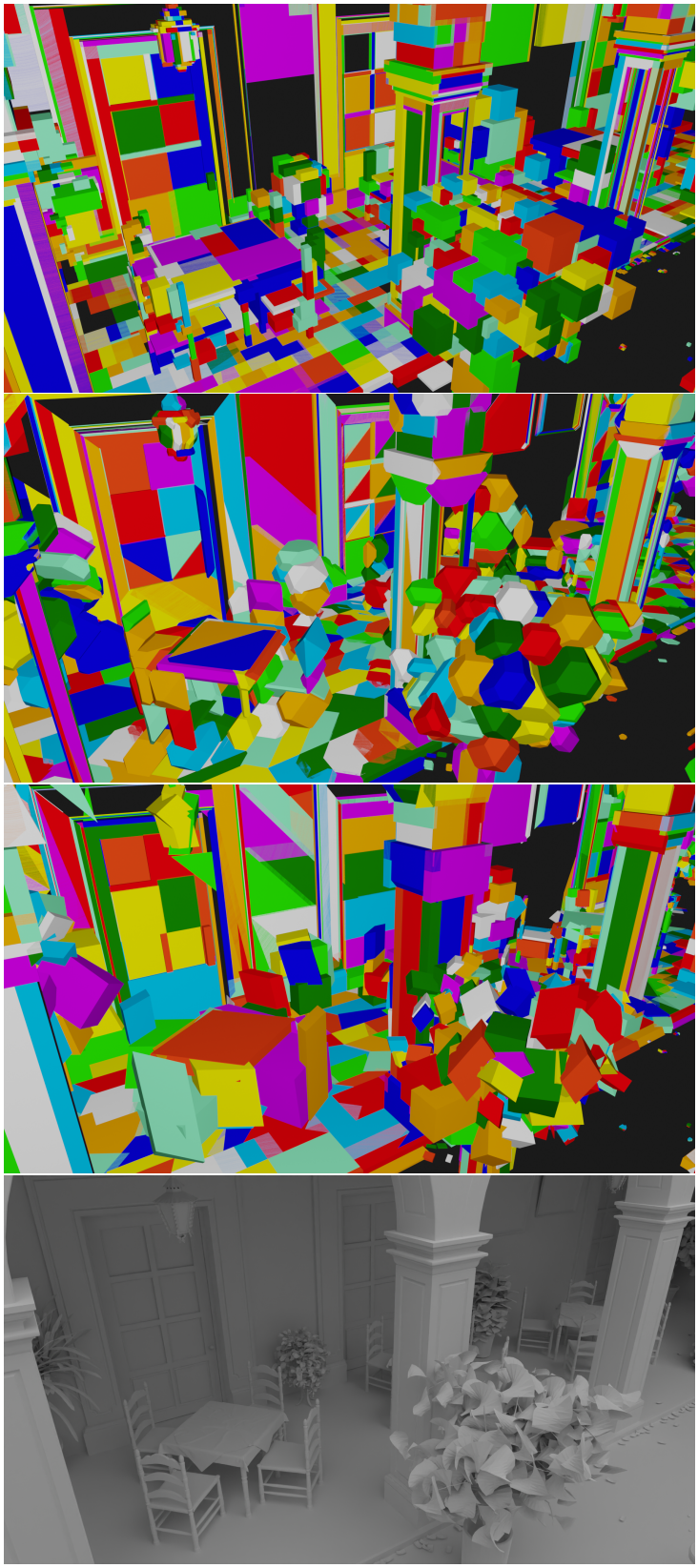
Další vizualizace naměřených dat

Pohled na vizualizace uzlů v prostoru (viz Obrázek 4.6) nám umožní částečně posoudit, jak dobře abaly obklopují geometrii a jak se chovají. Můžeme zde například vidět, že 14-DOP si neumí dobře poradit s předměty natočenými o 45° kolem některé z hlavních os. Lze to pozorovat na židlích, které se nachází v levé části.

Grafy 4.5 ukazují, že jednotlivé typy obalových těles se chovají podobně z hlediska plochy. Rozdíly lze pozorovat na počtu uzlů ve vrstvách hierarchie. SOBB neobsahuje tolik uzlů v hlubších vrstvách. To přisuzuji lepší schopnosti se přizpůsobit geometrii a vyšší konstantě C_t. Ta způsobuje, že dochází k většímu shlukování listů kvůli použitému adaptivnímu spojování listů, které má zlepšit SAH.



Obrázek 4.5: Vizualizace počtu uzlů hierarchie (nahoře) a součtu ploch těles (dole) v závislosti na hloubce uzlu.



Obrázek 4.6: Vizualizace 19. úrovně obalových těles na scéně San Miguel. Horní AABB, uprostřed 14-DOP, dole SOBB 64.

Kapitola 5

Závěr

Prostudoval jsem implementaci paralelní optimalizace hierarchií obalových těles na grafické kartě za použití technologie CUDA.

Rozšířil jsem základní framework o možnost používat obalová tělesa 14-DOP a SOBB a obě možnosti jejich uložení v paměti. Naimplementoval jsem možnost kombinovat různé typy těles s technikou T-SAH.

Provedl jsem měření kombinací naimplementovaných algoritmů na 10 statických a 10 animovaných scénách. Porovnal jsem kvalitu stromu vzhledem k metrice SAH, EPO, LCV i vzhledem k naměřenému výkonu při vrhání.

Připravil jsem dvě nové animované scény a systém na jejich načítání vhodný pro další práce týkající se animací.

Z prováděných měření je vidět, že i když 14-DOP a SOBB dokáží znatelně zlepšit SAH, záleží i na implementaci kernelu vrhání paprsků a na topologii scény, zda se je vyplatí používat. T-SAH považuji za velmi dobrou náhradu neustálého znovuvytváření scény a to především pro obalová tělesa SOBB, jejichž optimalizace je pomalá.

Pokud bych měl projekt vytvořit znovu, nerozděloval bych hodnoty os 4-7 14-DOPu do samostatných bufferů. Jednodušší by bylo mít vše ve dvou bufferech podobně jako u SOBB. Neučinil jsem tak, jelikož jsem během tvorby semestrálního projektu, kterým je tato práce pokračováním, nevěděl, že budu implementovat další typ obalového tělesa.

Možná vylepšení do budoucna:

- Převedení frameworku na CUDA runtime API a vypuštění abstrakční vrstvy vytvářené frameworkem.
- Vylepšení ořezávání SOBB, aby se mohlo optimalizovat podle opravdové hodnoty SAH a aby i výběr normál a adaptivní shlukování listů respektovaly tuto hodnotu.
- Přepsání ukládání dat během optimalizace, aby se ušetřila paměť. Použít perzistentní vlákna ke snížení potřebné paměti pro uložení obalového tělesa při optimalizaci T-SAH.

Literatura

[1] T. Foley a J. Sugerman, "KD-tree acceleration structures for a GPU raytracer", in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ř. HWWS '05, Los Angeles, California: Association for Computing Machinery, 2005, 15–22, ISBN: 1595930868. DOI: 10.1145/1071866.1071869. URL: https://doi.org/10.1145/1071866.1071869.

[2] J. H. Clark, "Hierarchical geometric models for visible surface algorithms", *Commun. ACM*, roč. 19, č. 10, 547–554, říj. 1976, ISSN: 0001-0782. DOI: 10.1145/360349.360354. URL: https://doi.org/10.1145/360349.360354.

[3] I. Wald, W. Usher, N. Morrical, L. Lediaev a V. Pascucci, "RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location", in *High-Performance Graphics - Short Papers*, 2019. DOI: 10.2312/hpg.20191189.

[4] K. Vaidyanathan, S. Woop a C. Benthin, "Wide BVH traversal with a short stack", in *Proceedings of the Conference on High-Performance Graphics*, 2019, s. 15–19.

[5] H. Ylitie, T. Karras a S. Laine, "Efficient incoherent ray traversal on GPUs through compressed wide BVHs", in *Proceedings of High Performance Graphics*, ř. HPG '17, Los Angeles, California: Association for Computing Machinery, 2017, ISBN: 9781450351010. DOI:

10.1145/3105762.3105773. URL: https://doi.org/10.1145/3105762.3105773.

[6] I. Wald, "On fast Construction of SAH-based Bounding Volume Hierarchies", in *2007 IEEE Symposium on Interactive Ray Tracing*, 2007, s. 33–40. DOI: 10.1109/RT.2007.4342588.

[7] D. Meister a J. Bittner, "Parallel reinsertion for bounding volume hierarchy optimization", in *Computer Graphics Forum*, Wiley Online Library, sv. 37, 2018, s. 463–473.

[8] T. Larsson a L. Källberg, "Fast computation of tight-fitting oriented bounding boxes", *Game Engine Gems 2*, s. 1, 2011.

[9] M. Káčerik a J. Bittner, "SAH-Optimized k-DOP Hierarchies for Ray Tracing", *Proc. ACM Comput. Graph. Interact. Tech.*, roč. 7, č. 3, srp. 2024. DOI: 10.1145/3675391. URL: https://doi.org/10.1145/3675391.

[10] M Káčerik a J Bittner, "SOBB: Skewed Oriented Bounding Boxes for Ray Tracing", in *Computer Graphics Forum*, Wiley Online Library, 2025, e70062.

[11] S. Dinas a J. M. Bañón, "A literature review of bounding volumes hierarchy focused on collision detection", *Ingeniería y competitividad*, roč. 17, č. 1, s. 49–62, 2015.

[12] T. Aila, T. Karras a S. Laine, "On quality metrics of bounding volume hierarchies", in *Proceedings of the 5th High-Performance Graphics Conference*, ř. HPG '13, Anaheim, California: Association for Computing Machinery, 2013, 101–107, ISBN: 9781450321358. DOI: 10.1145/2492045.2492056. URL: https://doi.org/10.1145/2492045.2492056.

[13] B. Fabianowski, C. Fowler a J. Dingliana, "A Cost Metric for Scene-Interior Ray Origins.", in *Eurographics (Short Papers)*, 2009, s. 49–52.

[14] M. Vinkler, V. Havran a J. Sochor, "Visibility driven BVH build up algorithm for ray tracing", *Computers Graphics*, roč. 36, č. 4, s. 283–296, 2012, Applications of Geometry Processing, ISSN: 0097-8493. DOI:

https://doi.org/10.1016/j.cag.2012.02.013. URL: https://www.sciencedirect.com/science/article/pii/S0097849312000362.

[15] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe a J. Bittner, "A survey on bounding volume hierarchies for ray tracing", in *Computer Graphics Forum*, Wiley Online Library, sv. 40, 2021, s. 683–712.

[16] I. Wald, "On fast Construction of SAH-based Bounding Volume Hierarchies", in *2007 IEEE Symposium on Interactive Ray Tracing*, 2007, s. 33–40. DOI: 10.1109/RT.2007.4342588.

[17] J. Pantaleoni a D. Luebke, "HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry", in *Proceedings of the Conference on High Performance Graphics*, 2010, s. 87–95.

[18] D. Meister a J. Bittner, "Parallel locally-ordered clustering for bounding volume hierarchy construction", *IEEE transactions on visualization and computer graphics*, roč. 24, č. 3, s. 1345–1353, 2017.

[19] A. Kensler, "Tree rotations for improving bounding volume hierarchies", in *2008 IEEE Symposium on Interactive Ray Tracing*, 2008, s. 73–76. DOI: 10.1109/RT.2008.4634624.

[20] L. R. Domingues a H. Pedrini, "Bounding volume hierarchy optimization through agglomerative treelet restructuring", in *Proceedings of the 7th Conference on High-Performance Graphics*, 2015, s. 13–20.

[21] J. Bittner, M. Hapala a V. Havran, "Fast insertion-based optimization of bounding volume hierarchies", in *Computer Graphics Forum*, Wiley Online Library, sv. 32, 2013, s. 85–100.

[22] M. Vinkler, J. Bittner a V. Havran, "Extended Morton codes for high performance bounding volume hierarchy construction", *Proceedings of High Performance Graphics*, 2017. URL: https://api.semanticscholar.org/CorpusID:5758143.

[23] D. Meister a J. Bittner, "Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing", *Journal of Computer Graphics Techniques (JCGT)*, roč. 11, č. 4, s. 1–19, 2022.

[24] J. Bittner a D. Meister, "T-SAH: Animation Optimized Bounding Volume Hierarchies", in *Computer Graphics Forum*, Wiley Online Library, sv. 34, 2015, s. 527–536.

[25] T. Karras a T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies", in *Proceedings of the 5th High-Performance Graphics Conference*, ř. HPG '13, Anaheim, California: Association for Computing Machinery, 2013, 89–99, ISBN: 9781450321358. DOI: 10.1145/2492045.2492055. URL: https://doi.org/10.1145/2492045.2492055.

[26] T. Aila a S. Laine, "Understanding the efficiency of ray traversal on GPUs", in *Proceedings of the Conference on High Performance Graphics 2009*, ř. HPG '09, New Orleans, Louisiana: Association for Computing Machinery, 2009, 145–149, ISBN: 9781605586038. DOI: 10.

1145/1572769.1572792. URL: https://doi.org/10.1145/1572769.1572792.

[27] S. Woop, "A ray tracing hardware architecture for dynamic scenes", *Fachrichtung 6.2-Informatik Computer Graphik, Saarland Univ., Saarbriicken, Germany*, 2004.

[28] N. Lohmann, *json*, https://github.com/nlohmann/json, 2013.

[29] P. Bourke, *Polygon File Format (PLY)*, https://paulbourke.net/dataformats/ply/, Accessed: 2025-05-21, n.d.

[30] S. Yoyo, *TinyGLTF: Header only C++11 glTF 2.0 loader and writer*, https://github.com/syoyo/tinygltf, Accessed: 2025-04-11, 2015.

[31] N. D. Blog, *CUDA Refresher: The CUDA Programming Model*, Accessed: 2025-01-19, 2023. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[32] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, Accessed: 2025-02-04, 2021. URL: https://docs.nvidia.com/cuda/archive/11.4.4/cuda-c-programming-guide/index.html.

[33] NVIDIA Corporation, *CUDA Toolkit Documentation*, Accessed: 2025-01-19, 2025. URL: https://docs.nvidia.com/cuda/index.html.

[34] J. Huynh, "Separating axis theorem for oriented bounding boxes", *URL: jkh. me/files/tutorials/Separating% 20Axis% 20Theorem% 20for% 20Oriented% 20Bounding% 20Boxes. pdf*, 2009.

[35] M. McGuire. "Computer Graphics Archive". (2017), URL: https://casual-effects.com/data.

[36] I. Wald, *Animation Representation*, Accessed: 2025-01-19, 2023. URL: https://www.sci.utah.edu/~wald/animrep/.

[37] GAMMA Group, *UNC Dynamic Scene Benchmarks - GAMMA*, https://gamma.cs.unc.edu/DYNAMICB/, Accessed: 2024-04-16, 2012.

Příloha A

Návod na implementaci nového typu obalového tělesa

Zde popíšu kroky potřebné k rozšíření aplikace o další typ obalového tělesa. Budu se zabývat částí, která je společná pro všechny typy těles. Některá tělesa vyžadují speciální zacházení jako je tomu například pro SOBB, které potřebuje vygenerovat normály polorovin. Rozšíření je hlavně zaměřeno na tělesa tvořená průniky polorovin. Postup lze upravit i pro jiná tělesa.

První krok je přidání nastavení, které bude načteno z JSON souboru, aby šlo poznat, že uživatel chce vytvořit nový typ obalového tělesa. Ve třídě `SceneData` přidejte proměnnou určující, že se používá nový typ tělesa. Následně do ní načtěte data ve funkci `App::ParseJson`.

Přidejte identifikační číslo nového typu tělesa do `volumeTypes.h` za pomoci makra. To bude použito při rozhodování, jaký typ tělesa se právě zpracovává.

Dále je potřeba vytvořit třídu reprezentující nové obalové těleso. Ta se bude používat jak na CPU, tak na GPU a je tedy potřeba, aby se nepoužívaly funkce patřící jen k jedné platformě. Pro matematické operace použijte namespace `FW`. Každá funkce musí mít před sebou makro `FW_CUDA_FUNC`. Inspirujte se třídou SOBB v `sobb.h`. Je zapotřebí, aby byla implementována funkce `area()`, která spočte plochu obalového tělesa. Dále se tělesa musí umět slučovat, aby bylo možné je použít v PRBVH při optimalizaci SAHu.

Nová třída je potřeba i pro reprezentaci uzlu ve finálním CudaBVH. Vytvořte třídu podobající se `CudaBVHNode`. Ta musí obsahovat základní informace uložené v proměnných `begin, end, size, parent` určujících topologii hierarchie. Ve zbytku třídy musí být uložena data obalových těles dvou potomků.

Nyní se můžete přesunout do tříd `HybridPRBVHBuilder` a `HybridPRBVHCollapser`. Zde je potřeba v každé funkci přidat několik `else if` podmínek a v nich provést danou operaci podle potřeb nového tělesa. Většinou se jedná o načtení správného jména kernelu nebo alokování dat podle potřeb daného tělesa. Tyto rozcestníky můžete najít pomocí vyhledání použití proměnných `use_sobb` nebo `use_kdop`.

Nejvíce změn bude probíhat v `hybridPRBVHBuilderKernels.cu` a `hybridPRBVHColalpserKernels.cu`. Zde se nachází všechny kernely, které potřebují individuální zacházení pro každé těleso. Provedu Vás úpravou refit kernelu. Kernel je vždy tvořen šablonovou funkcí `hybrid_refit(...)`. Šablonový parametr volumeType určuje, pro jaký typ obalového tělesa se má funkce vytvořit. Použijte ho dohromady s vámi definovaným identifikátorem nového tělesa k oddělení kódu specifického pro nové těleso od kódu, který tělesa sdílí. Pokud váše těleso vyžaduje více vstupních bufferů, budete muset upravit argumenty funkce. Nyní je potřeba se podívat pod upravovanou funkci. Naleznete tam funkce se stejnými vstupy a názvem jako předešlá upravovaná funkce, ale tentokrát mají na konci označení typu tělesa (např. `hybrid_refit_AABB(...)`). Tyto funkce jsou opravdové kernely a vždy volají šablonovou funkci se specifickým nastavením. Vytvořte takovýto kernel a označte ho názvem svého tělesa. Pokud si to situace vyžaduje, nezapomeňte na T-SAH variantu. Pokud jste přidávali argumenty funkce, musíte změnit i ostatní kernely. Jinak na Vás IDE bude "křičet". Pro kernel `findBestNode` jsou již připravené čtyři argumenty `additionalData`, které můžete využít.

Tento postup je potřeba provést u všech funkcí výše zmíněných dvou souborů. Postupujte opatrně a dávejte si pozor, ať použijete `else if()` a ne jenom `if()`.

Poslední krok je upravení kernelu vrhání paprsků v `kdop_tesla_persistent_while_while.cu`. Je potřeba provést změny podobného ražení jako v předchozím odstavci. Zde jsou však sekce obalových těles více oddělené, jelikož se vše hodně liší. Upravte funkci `HybridPRBVH::getTraceKernelName` tak, aby vracela správný název kernelu pro váše obalové těleso.

Příloha B

Návod na implementaci nové metriky optimalizované pomocí PRBVH

Pro implementaci nové metriky jsou zapotřebí pouze dva kroky.

Zaprvé je potřeba vytvořit kernel, který dokáže vypočítat danou metriku. Doporučuji vyjít z kernelu `ComputeCost`, který počítá SAH. Postupujte podobně jako při implementaci nového obalového tělesa (viz Příloha A) a využijte šablonu pro podporu více typů těles. Pokud Vás zajímá i výsledná hodnota metriky po ukončení optimalizace a aplikování Collapseru, je potřeba implementovat výpočet metriky i z finální reprezentace uzlů hierarchie. Inspirujte se kernelem `ComputeCostBVH`. Ten používá i šablonu se třídou uzlu podle typu obalového tělesa. Po jeho přetypování na specifický typ můžete přistupovat k hodnotám tělesa. Výpočet metriky bude potřeba značně upravit, jelikož uzel neobsahuje své obalové těleso, ale tělesa potomků.

Zadruhé je potřeba vytvořit variantu funkce `FindBestNode` využívající novou metriku. Zkopírujte si tuto funkci a vyhledejte v ní funkci `area()`. Ta se nachází na místech, kde se vypočítává změna SAHu. Zde je potřeba vypočítat vaši metriku pro dané uzly. Tato implementace předpokládá, že je možné metriku vypočítat bez nutnosti spouštění dalšího kernelu. Pokud nechcete vytvářet další funkci specificky pro vaší metriku, můžete přidat další šablonový parametr a oddělit vaši metriku od SAHu pomocí něj. Kvůli přehlednosti tuto variantu nedoporučuji.

Nové kernely je třeba správně načíst a zavolat ze třídy `HybridPRBVHBuilder` popřípadě z vašeho vlastního Builderu.

Příloha C

Struktura odevzdaných zdrojových kódů

```
root
├── SLN_project
│   ├── cub - knihovna komponent pro CUDA
│   ├── gpu-ray-traversal
│   │   ├── scenes
│   │   │   ├── rt
│   │   │   │   ├── animations - složka se soubory animovaných scén
│   │   │   │   ├── static - složka se soubory statických scén
│   │   │   │   └── scene_template.json - ukázkový konfigurační soubor
│   │   │   │       scény
│   │   ├── src
│   │   │   ├── framework - zdrojové soubory NVIDIA frameworku
│   │   │   ├── libs - soubory použitých knihoven třetích stran
│   │   │   ├── rt - zdrojové soubory PRBVH a mnou naimplementovaných
│   │   │   │   tříd
│   │   ├── benchmark.py - skript pro spouštění testů
│   ├── rt.sln - Visual studio řešení
├── README.md - návod na spuštění projektu
```