

Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Akcelerační datové struktury pro 3D zobrazování

Bc. Pavel Lokvenc

Diplomová práce
2016

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2015/2016

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Pavel Lokvenc**
Osobní číslo: **I14267**
Studijní program: **N2646 Informační technologie**
Studijní obor: **Informační technologie**
Název tématu: **Akcelerační datové struktury pro 3D vizualizaci**
Zadávající katedra: **Katedra softwarových technologií**

Z á s a d y p r o v y p r a c o v á n í :

Cílem práce je popis, implementace, testování a vzájemné porovnání vybraných akceleračních struktur, používaných při vizualizaci 3D scény. Dané struktury budou použity ve spojení s metodou path tracingu.

V teoretické části budou popsány dané akcelerační struktury. Dále bude popsána samotná metoda path tracingu.

V praktické části bude provedena implementace vybraných struktur a implementace samotné vizualizační metody. Implementace bude provedena v jazyce C++.

Dále bude vytvořen ukázkový testovací nástroj, který bude schopen spouštět proces vizualizace vybrané scény s použitím jednotlivých akceleračních struktur. Testovací nástroj bude produkovat výsledný obraz scény a příslušné statistické údaje o procesu vizualizace. Získané statistiky budou následně zpracovány v odpovídající přehledné formě.

Hlavním přínosem práce by mělo být, na základě výsledných statistik, doporučení ohledně výkonu a vhodnosti použití dané akcelerační struktury dle charakteru scény.

Rozsah grafických prací:

Rozsah pracovní zprávy: cca 60 stran

Forma zpracování diplomové práce: tištěná

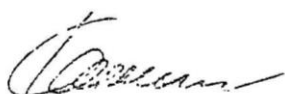
Seznam odborné literatury:

PHARR, Matt a Greg HUMPHREYS. Physically based rendering: from theory to implementation. 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2010, xxvii,1167 p. ISBN 01-237-5079-2.

SHIRLEY, Peter. Realistic ray tracing. Vyd. 1. Massachusetts: Natick, 2003, 225 s.ISBN 15-688-1198-5.

Vedoucí diplomové práce: **Ing. Petr Veselý**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2015**
Termín odevzdání diplomové práce: **13. května 2016**



prof. Ing. Simeon Karamazov, Dr.
děkan



L.S.



prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2015

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Polici nad Metují dne 13. 5. 2016

podpis autora
Bc. Pavel Lokvenc

PODĚKOVÁNÍ

Poděkování bych chtěl věnovat především rodičům, kteří mě po celou dobu studia podporovali.

ANOTACE

Diplomová práce se zabývá akceleračními datovými strukturami určenými pro optimalizaci 3D zobrazování. V práci jsou popsány principy datových struktur grid, bounding volume hierarchy a kd-tree. Dále je v práci popsána metoda path tracing. Tato metoda a uvedené datové struktury jsou implementovány v jazyce C++ a jsou provedena měření pro porovnání výkonnosti struktur na několika vybraných scénách.

KLÍČOVÁ SLOVA

3D grafika, sledování cest, mřížka, bvh, hierarchie obalové geometrie, kd-tree, optimalizace, akcelerační datové struktury

TITLE

Acceleration data structures for a 3D rendering

ANNOTATION

This thesis is focused on the acceleration data structures for a 3D rendering optimization. The principles of the grid, bounding volume hierarchy and kd-tree data structures are described, altogether with the path tracing rendering method. The latter method and the corresponding data structures are implemented in C++ language and the performance of data structures is compared on a few testing scenes.

KEYWORDS

3D graphics, path tracing, grid, bvh, bounding volume hierarchy, kd-tree, optimization, acceleration structures

OBSAH

Úvod.....	12
1 Principy renderovacích metod.....	13
1.1 Přehled renderovacích metod.....	14
1.1.1 Ray tracing	14
1.1.2 Distributed ray tracing	14
1.1.3 Path tracing.....	14
1.1.4 Photon mapping	15
1.2 Základní princip renderovacích metod	15
1.3 Objekty ve scéně	16
1.3.1 Tělesa.....	16
1.3.2 Materiály	17
1.3.3 Světla	17
2 Path tracing	18
2.1 Matematický popis.....	20
2.2 Monte Carlo integrace	21
2.3 Vzorkování polokoule.....	22
2.4 Ukončení algoritmu	25
3 Optimalizace renderingu	26
3.1 Grid.....	27
3.1.1 Vytváření struktury	28
3.1.2 Průchod strukturou.....	29
3.2 Bounding volume hierarchy	32
3.2.1 Reprezentace binárního stromu	33
3.2.2 Obalová geometrie.....	34
3.2.3 Vytváření struktury	35
3.2.4 Průchod strukturou.....	37
3.3 Kd-tree.....	39
3.3.1 Vytváření struktury	40
3.3.2 Průchod strukturou.....	42

4	Implementace.....	46
4.1	Architektura.....	46
4.2	Základní třídy	47
4.2.1	Generátor náhodných čísel	48
4.2.2	Nástroje pro správu paměti	49
4.3	Implementace path tracingu	50
4.4	Akcelerační datové struktury	52
4.4.1	Grid	53
4.4.2	Bounding volume hierarchy	54
4.4.3	Kd-tree	55
4.5	Statistiky	57
5	Měření	59
5.1	Jednotlivé scény	60
5.2	Závěry měření	65
6	Závěr.....	69
7	Použitá literatura	71
8	Seznam příloh	73

SEZNAM ILUSTRACÍ A TABULEK

Obrázek 1 – Ilustrace zobrazovací rovnice	14
Obrázek 2 – Princip <i>renderovacích</i> metod vycházejících z <i>ray tracingu</i>	15
Obrázek 3 – Možnosti průsečíku paprsku s koulí	16
Obrázek 4 – Ilustrace algoritmu Möller-Trumbore [1].....	17
Obrázek 5 – Odraz světla od povrchu tělesa	17
Obrázek 6 – Osvětlení ploch, kde přímo nedopadá sluneční světlo.....	18
Obrázek 7 – Ukázka cest pro jeden pixel při použití metody Monte Carlo	19
Obrázek 8 – Ukázka metody <i>path tracing</i>	20
Obrázek 9 – Náhodná procházka scénou s koncem ve zdroji světla.....	21
Obrázek 10 – Sférický prostor	23
Obrázek 11 – Vygenerovaný náhodný vektor	23
Obrázek 12 – Transformace podle <i>LCS</i> v místě dopadu	25
Obrázek 13 – Nerovnoměrně distribuovaná scéna.....	28
Obrázek 14 – Přechody mezi <i>voxely</i> , barevně odlišené v jednotlivých osách ...	31
Obrázek 15 – Průběh algoritmu <i>DDA</i> , přechody barevně odlišeny	31
Obrázek 16 – Ukázka jednoduché <i>BVH</i>	32
Obrázek 17 – Dynamická reprezentace binárního stromu v paměti.....	33
Obrázek 18 – Reprezentace binárního stromu „do šířky“ na poli	33
Obrázek 19 – Reprezentace binárního stromu „do hloubky“ na poli	34
Obrázek 20 – Určení osy podle vzdálenosti mezi těžišti krajních těles.....	36
Obrázek 21 – Scéna a pořadí průchodu scénou nad binárním stromem	38
Obrázek 22 – Ilustrace postupného vkládání a odebírání do zásobníku	39
Obrázek 23 – Vizualizace <i>kd-tree</i> . Zdroj: http://blog.yiningkarlli.com/	40
Obrázek 24 – Vytváření <i>kd-tree</i>	41
Obrázek 25 – Ilustrace průchodu přes <i>kd-tree</i> [13]	42
Obrázek 26 – Ukázky případů, kdy je jeden z potomků vynechán [13]	42
Obrázek 27 – Určení pořadí podle osy řezu a počátku	43
Obrázek 28 – Scéna pro ukázkou průchodu strukturou <i>kd-tree</i>	43
Obrázek 29 – Průchod přes <i>kd-tree</i>	44
Obrázek 30 – Zjednodušený náčrt architektury	47
Obrázek 31 – Diagram tříd pro tělesa a akcelerační struktury.....	52
Obrázek 32 – <i>Happy Buddha</i> , výstup testu.....	60
Obrázek 33 – <i>Hairball</i> , výstup testu.....	61
Obrázek 34 – <i>Dragon</i> , výstup testu.....	62
Obrázek 35 – Sponza, výstup testu.....	63
Obrázek 36 – Katedrála sv. Jakuba, výstup testu	64
Obrázek 37 – Graf propustnosti datových struktur	65
Obrázek 38 – Graf paměťové náročnosti datových struktur	66

Obrázek 39 – Graf doby trvání vytváření datových struktur	67
Obrázek 40 – Graf časů tvorby datových struktur nad scénou <i>Hairball</i>	67
Obrázek 41 – Graf časů potřebných pro vykreslení scén	68
Tabulka 1 – Výsledky měření pro scénu <i>Happy Buddha</i>	60
Tabulka 2 – Výsledky měření pro scénu <i>Hairball</i>	61
Tabulka 3 – Výsledky měření pro scénu <i>Dragon</i>	62
Tabulka 4 – Výsledky měření pro scénu atrium Sponza.....	63
Tabulka 5 – Výsledky měření pro scénu atrium katedrála sv. Jakuba.....	64

SEZNAM ZKRATEK A ZNAČEK

BVH	bounding volumes hierarchy
QBVH	quad bounding bolumes hierarchy
GCS	global coordinates system
LCS	local coordinates system
SSE	streaming SIMD extensions
MMX	MultiMedia eXtensions
AVX	Advanced Vector Extensions
NURBS	non uniform rational bezier surface
SIMD	single instruction multiple data
DDA	digital differential analyzer
RGB	barevny prostor red, green, blue
PNG	Portable Network Graphics
STL	standard template library
BSP	binary space partitioning
SAH	surface area heuristic
BRDF	bidirectional reflection distribution function
HDR	high dynamic range

ÚVOD

Výsledky všech možných metod pro fotorealistické zobrazování v dnešní době vídáme každý den. Od celovečerních filmů, přes reklamní kampaně až po věci technického a třeba lékařského charakteru. S tím jde ruku v ruce také vývoj algoritmů a metod pro jejich implementaci.

Přestože v poslední době výkon výpočetní techniky prudce stoupá, fotorealistické zobrazování je i při použití moderních algoritmů pro výpočty reálného osvětlení stále velice výpočetně náročná záležitost. Tyto datové struktury a algoritmy řádově snižují časovou náročnost výpočtu fotorealistického zobrazení trojrozměrné scény. Jejich využití je nutné i v případě použití rozsáhlých distribuovaných *renderovacích* systémů.

Toto téma diplomové práce bylo zvoleno z toho důvodu, že moje bakalářská práce se zabývala metodou *ray tracing* a pojetí diplomové práce jako pokračování byla logická volba, zvláště když bylo zváženo, kolik možností a prostoru se v tomto tématu skrývá.

V diplomové práci budou popsány datové struktury *grid*, *bounding volume hierarchy* a *kd-tree*, které redukuje počet výpočtů typu paprsek-objekt. Tyto struktury budou také implementovány, některé několika různými způsoby. Nakonec bude provedeno měření klíčových vlastností těchto datových struktur na několika testovacích scénách, které pokrývají co nejširší škálu zkoumaných problémů. Měřena je jak časová náročnost jednotlivých fází výpočtu, tak jejich paměťová náročnost.

Pro testování bude použita metoda *path tracing*. Tato metoda byla zvolena kvůli své podstatě, kdy umožňuje testovat nejen koherentní paprsky, které jdou z kamery do scény, ale také paprsky, které vznikají v celé scéně a mají náhodný směr.

Testovací prostředí bude napsáno v příkazové řádce, aby došlo k co nejmenším ztrátám na výkonu, které by mohly nastat v případě nutnosti vykreslování v reálném čase. Dále bude celý algoritmus paralelizován, což s sebou nese některá úskalí, která budou v rámci implementace řešena.

Cílem celého měření bude porovnání akceleračních datových struktur a vhodnost jednotlivých typů akceleračních datových struktur pro několik různých druhů scén. Z tohoto měření pak bude možné vyvodit, který typ datové struktury je vhodný pro daný typ scény.

1 PRINCIPY RENDEROVACÍCH METOD

První kapitola této diplomové práce je stručným shrnutím základů realistických zobrazovacích metod. Pro pochopení problematiky diplomové práce je nutné znát problematiku realistického 3D *renderingu* alespoň v rozsahu [1]. Jako další literaturu je možné doporučit [2] a [3].

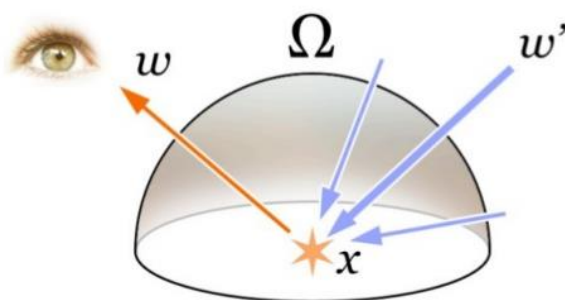
Realistické *renderovací* metody založené na metodě *ray tracing* jsou více či méně přesnou aproximací zobrazovací rovnice, kterou v roce 1989 publikoval James T. Kayija [4]. Zobrazovací rovnice popisuje, jakým způsobem probíhá distribuce světla ve scéně:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i \quad (1.1)$$

kde:

- λ – je konkrétní vlnová délka světla (každá vlnová délka se může jinak odrazet),
- t – je čas,
- \mathbf{x} – je konkrétní místo ve scéně,
- ω_o – je směr odchozího světla,
- ω_i – je obrácený směr příchozího světla,
- $L_o(\mathbf{x}, \omega_o, \lambda, t)$ – je všechna odchozí záře ve směru ω_o , o vlnové délce λ , v t a v \mathbf{x} ,
- $L_e(\mathbf{x}, \omega_o, \lambda, t)$ – emitovaná záře,
- Ω – je jednotková polokoule obsahující všechny hodnoty ω_i ,
- $\int_{\Omega} \dots d\omega_i$ – integrál příchozí záře přes jednotkovou polokouli Ω ,
- $f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)$ – *BRDF*, pravděpodobnost, že se světlo odrazí daným směrem, více v kapitole 1.3.2, podrobněji potom v [1],
- $L_i(\mathbf{x}, \omega_i, \lambda, t)$ – světlo přicházející ze směru ω_i ,
- $(\omega_i \cdot \mathbf{n})$ – faktor zeslabení světla v závislosti na úhlu mezi vektorem ω_i a normálou.

V zobrazovací rovnici je vidět integrál přes všechny směry příchozí záře, a je zřejmé, že přesné analytické vyjádření lze získat pouze u těch nejjednodušších případů. Zobrazovací metody se pokoušejí tento integrál aproximovat pomocí rozdílných technik, jako je výpočet integrálu pomocí metody Monte Carlo, metoda konečných elementů, *photon mapping* a jiné.



Obrázek 1 – Ilustrace zobrazovací rovnice

1.1 Přehled renderovacích metod

V této podkapitole je stručný přehled *renderovacích* metod, aby se čtenář lépe zorientoval v problematice diplomové práce.

1.1.1 Ray tracing

Metoda sledování paprsků byla představena T. Whittedem. Vychází z geometrické optiky a umožňuje vykreslit libovolný objekt, pro který je možné vypočítat průsečík s paprskem. Jako výhody této metody můžeme považovat to, že vychází z fyzikálních zákonů, její poměrně snadnou implementaci a možnost poměrně snadné paralelizace.

Naopak nevýhodami je značná výpočetní náročnost, nemožnost zobrazovat měkké stíny a rozostřené odrazy, rozklad světla pod povrchem materiálu, a to, že výsledný obrázek je zatížený silným *aliasingem*. Jedná se ovšem o velmi hrubou aproximaci zobrazovací rovnice, která vůbec nepočítá globální osvětlení a kaustické efekty.

1.1.2 Distributed ray tracing

Vylepšuje možnosti klasického *ray tracingu* tím, že vzorku paprsky jak v prostoru, tak v čase a díky tomu eliminuje *aliasing* a dokáže pracovat s hloubkou ostroty, měkkými stíny, neostrý odraz a lom světla a rozmazání pohybem.

1.1.3 Path tracing

Algoritmus publikoval v roce 1986 James Kajiya spolu se zobrazovací rovnicí [4]. Věrně simuluje globální osvětlení v trojrozměrné scéně. Zobrazovací rovnici řeší pomocí Monte Carlo integrace. Pro každý pixel sleduje možné cesty paprsků světla, vycházející ze světelných zdrojů, které na něj po interakci se scénou dopadají.

Sledování cest fyzikálně přesně simuluje efekty globálního osvětlení, jako jsou lomy a odrazy světla, *color bleeding*, kaustika, měkké stíny, ale také hloubku ostroty a rozmazání pohybem.

K dokonale vypadajícímu obrázku je však třeba provést velké množství simulací, jinak je obrázek zatížen viditelným šumem.

Tato metoda byla dále rozšířena metodami *bidirectional path tracing* a *Metropolis light transport*. V současné době také existují implementace na grafické kartě, které podávají poměrně solidní výsledky v reálném čase.

Tato metoda bude podrobně popsána v další části diplomové práce.

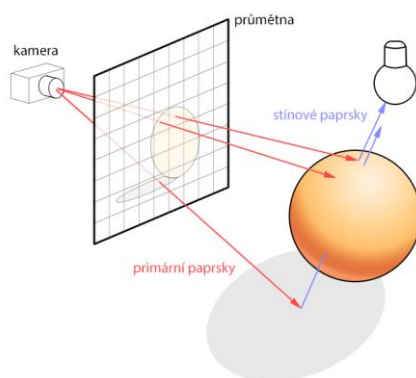
1.1.4 Photon mapping

Metodu publikoval Henrik Wann Jensen. *Rendering* a probíhá ve dvou částech. V první se předpočítá fotonová mapa a ve druhé části se obrázek vykreslí pomocí distribuovaného *ray tracingu*, kde se k výsledné záři přičítá ještě příspěvek nejbližších fotonů.

Pomocí této metody lze počítat veškeré efekty zmiňované v části o *path tracingu*. Mapování fotonů je silné především při výpočtu kaustiky. Při nedostatečném počtu fotonů jsou výsledné obrázky flekaté, takže je nutné tuto metodu rozšířit o další části výpočtu.

1.2 Základní princip renderovacích metod

Všechny výše uvedené zobrazovací metody tak, že vrhají z kamery skrz jednotlivé pixely paprsky do scény a je třeba vypočítat nejbližší průsečík s objektem ve scéně. Náročnost výpočtu se pak úměrně roste s velikostí výsledného obrázku a počtem těles ve scéně.



Obrázek 2 – Princip *renderovacích* metod vycházejících z *ray tracingu*

V některých metodách je třeba také počítat, jestli je průsečík osvětlen světelným zdrojem. To se řeší pomocí stínových paprsků, které jsou vystřeleny z průsečíku směrem ke světlu, a zjišťuje se, jestli se po cestě nacházejí překážky.

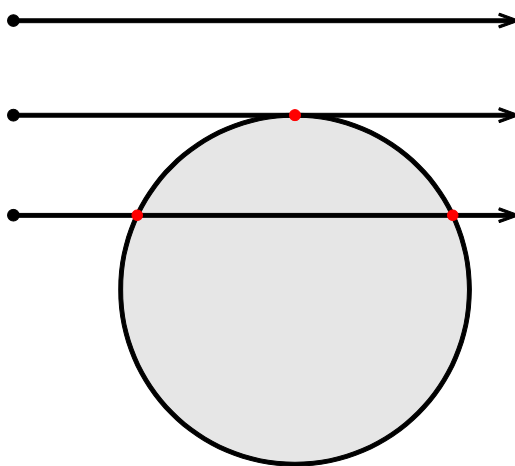
Posledním typem jsou paprsky, které vznikají při ostrém odrazu a lomu světla. Ty začínají v průsečíku a mají směr vypočítaný pomocí zákona odrazu a Snellova zákona lomu světla.

1.3 Objekty ve scéně

Ve scéně se zpravidla uchovává velké množství informací, které slouží k jejich definici. Na první pohled nejdůležitější jsou informace o tělesech, materiálech a jejich *BRDF* funkcích a světlech. Další věci uchovávané ve scéně jsou textury, *cachované* informace o částech výpočtů, akcelerační datové struktury atd.

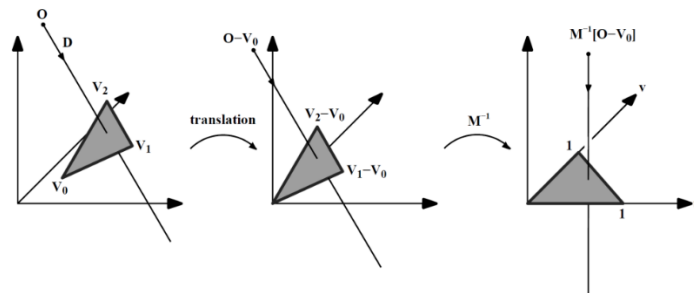
1.3.1 Tělesa

Hlavní podmínkou pro to, aby výše uvedené *renderovací* metody zobrazit tělesa, je, že je možné vypočítat průsečík tělesa a paprsku. Z tohoto důvodu je možné zobrazit pouze poměrně omezenou škálu objektů. Mezi ně se řadí koule, trojúhelník, kužel, plocha, *NURBs* plochy a parametricky definované plochy.



Obrázek 3 – Možnosti průsečíku paprsku s koulí

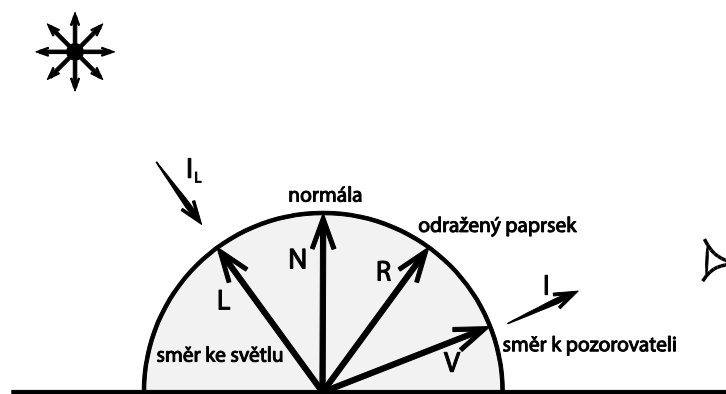
V této diplomové práci jsou zahrnuty pouze tvary koule a trojúhelníku. U koule se průsečík počítá odvozeným vzorcem z parametrické rovnice koule a parametrické rovnice polopřímky. U trojúhelníku je použit algoritmus *Möller-Trumbore*, který k výpočtu průsečíku využívá barycentrické souřadnice, které je možné využít k interpolaci normál a mapovacích souřadnic [1].



Obrázek 4 – Ilustrace algoritmu Möller-Trumbore [1]

1.3.2 Materiály

Materiály jsou definované pomocí *BRDF* funkcí, které říkají, kolik světla o jaké vlnové délce se má odrazit v daném směru. Materiál dále může definovat věci jako je průhlednost, odrazivost, průsvitnost. To vše je možné řešit jak konstantními barvami, tak třeba procedurálními texturami nebo texturou načtenou z externího souboru.



Obrázek 5 – Odraz světla od povrchu tělesa

Pro metodu *path tracing* je vhodné do materiálu také zahrnout, jestli daný materiál emituje světlo a jeho barvu resp. vlnovou délku. To je možné využít jak při klasickém *path tracingu* tak při jeho modifikaci, kde se využívá také explicitního svícení z tělesa, které je navzorkováno bodovými světly.

1.3.3 Světla

Světla je možné definovat klasickým způsobem jako světla bodová, směrová, kuželová nebo plošná tak jak je uvedeno v [1].

Tento přístup však v některých případech není úplně vhodný, jak pro zde použitou výpočetní metodu, tak pro uživatele, který může chtít definovat světlo jako libovolný tvar ve scéně. Z toho důvodu se informace o emitované záři často ukládají jako vlastnost materiálů.

2 PATH TRACING

Jak již bylo zmíněno v předcházející části diplomové práce, *path tracing* [4], česky sledování cest, je *renderovací* metoda, která pomocí metody Monte Carlo přesně řeší integrál zobrazovací rovnice.

V první řadě je nutné zmínit, jaký je vlastně rozdíl mezi přímým a nepřímým osvětlením. Pokud je počítáno pouze s tím, že světlo se odráží pouze jednou a to přímo k pozorovateli, jedná se o přímé osvětlení. V reálném světě se však světlo odráží od povrchu vícekrát. Během této cesty dochází v závislosti na materiálu povrchu, od kterého se světlo odráží, k útlumu frekvencí ve světle. Takové světlo je tak vnímáno jako barevné. V tomto případě pak mluvíme o osvětlení nepřímém. Díky tomu pak dochází k efektům jako je *color bleeding*, kaustika atd.



Obrázek 6 – Osvětlení ploch, kde přímo nedopadá sluneční světlo

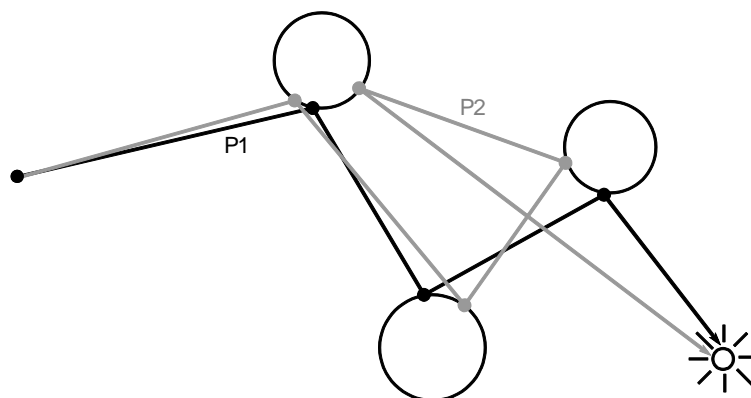
Existují dva možné přístupy k tomu jak provádět výpočet při metodě *path tracingu*. Přístupem je simulace skutečného chování světla v reálném světě. To jde směrem od světelného zdroje k pozorovateli. Obrovskou nevýhodou tohoto přístupu je, že velké množství cest, které budou tímto přístupem generovány, nebudou směřovat k pozorovateli, ale skončí kdesi ve scéně. Tento přístup se nazývá dopředné sledování (*forward tracing*).

Druhý přístup, který se nazývá zpětné sledování cest (*backward tracing*), funguje obráceně. Cesty jsou generovány od pozorovatele se snahou dostat se ke zdroji světla. I tady však dochází k tomu, že některé cesty „uléhnu“ ze scény, nebo jsou ukončeny, přestože ještě nedorazily ke zdroji světla. Zvláště výrazné

je to v případě, že zdroje světla mají malou plochu a je tak malá pravděpodobnost, že náhodně generovaná cesta k nim dojde.

V metodě *path tracing* se zavádí pojem cesta pro jeden průchod scénou náhodný průchod scénou. Při každém odrazu je vygenerován náhodný směr nového paprsku. Takto se rekurzivně pokračuje až do chvíle, kdy se cesta dostane ke zdroji světla. V každém bodě odrazu se pak aktualizuje světelný příspěvek pro danou cestu. Pokud se cesta nebude končit ve zdroji světla, pak je vrácen světelný příspěvek pozadí. Toho je možné využít a namapovat na nekonečně velkou kouli kolem scény *HDR* obrázek prostředí, který je díky své barevné hloubce možné použít k osvětlení scény.

Přestože řešení tohoto problému je intuitivní a technicky velmi jednoduché, tak je velice výpočetně náročné a při použití malého počtu vzorků také náchylné k chybám. Hlavním důvodem vzniku těchto chyb je, že není možné v konečném čase vypočítat všechny cesty světla. Zde se dostává ke slovu metoda Monte Carlo, která vybírá náhodné vzorky, které sledujeme. Pro každý pixel tak bude vypočteno N vzorků, které budou zprůměrovány. Metoda Monte Carlo integrace bude více popsána v kapitole 2.2.



Obrázek 7 – Ukázka cest pro jeden pixel při použití metody Monte Carlo

Klasická metoda *path tracing*, tak jak byla publikována v [4], využívá pouze zpětného sledování cest. Existují také modifikace této metody, například *bidirectional path tracing*, které využívají kombinaci obou přístupů. Tento přístup pak řeší pomalou konvergenci scény a problémy při výpočtu kaustických efektů. Kaustické efekty vznikají při lámání světla na průhledných materiálech, jako je sklo. Při stejném počtu vzorku pak vznikne kvalitnější výstup.

V této diplomové práci je implementována metoda klasického *path tracingu*, který využívá zpětné sledování cest ve scéně.



Obrázek 8 – Ukázka metody *path tracing*

2.1 Matematický popis

Metoda *path tracing* přímo vychází ze zobrazovací rovnice. Přestože už byla v diplomové práci jednou uvedena, pro názornost bude lepší ji uvést znovu:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i \quad (2.1)$$

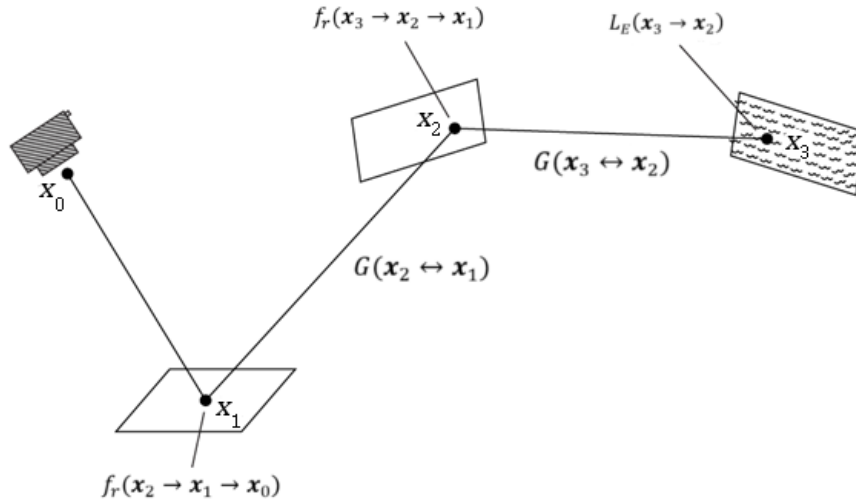
Nyní tuto rovnici bude rovnice zjednodušena, aby byly zkráceny následující zápisy:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) (\omega_i \cdot \mathbf{n}) d\omega_i \quad (2.2)$$

Je bráno, že čas je v celé cestě konstantní a pracuje se s celým spektrem najednou.

Tato rovnice ale popisuje chování pouze v jednom bodě scény. Při aplikaci *path tracingu* je třeba zobrazovací rovnici aplikovat na všechny body celé cesty.

Pro potřeby integrování celé cesty je nutné zavést nové další značení. Body cesty tedy označíme jako $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_n$. Výsledky *BRDF* funkcí v bodech odrazu mezi jednotlivými částmi cesty pak budou značeny jako $f_r(\mathbf{x}_2 \rightarrow \mathbf{x}_1 \rightarrow \mathbf{x}_0)$, obecně jako $f_r(\mathbf{x}_{i+1} \rightarrow \mathbf{x}_i \rightarrow \mathbf{x}_{i-1})$, koeficient zeslabení mezi jednotlivými částmi cesty bude značen jako $G(\mathbf{x}_2 \leftrightarrow \mathbf{x}_1)$, obecně potom značeno jako $G(\mathbf{x}_{i+1} \leftrightarrow \mathbf{x}_i)$.



Obrázek 9 – Náhodná procházka scénou s koncem ve zdroji světla

Bod \mathbf{x}_0 leží na „filmu“ kamery, bod \mathbf{x}_n leží na zdroji světla. Body $\mathbf{x}_1, \dots, \mathbf{x}_{n-1}$ leží na povrchu objektů, na kterých cesta mění směr. Zobrazovací rovnici je nutné aplikovat na každý bod $\mathbf{x}_1, \dots, \mathbf{x}_{n-1}$. Nechť $P(\bar{\mathbf{x}}_n)$ je označením světelného příspěvku celé cesty. Tento příspěvek je pak roven:

$$P(\bar{\mathbf{x}}_n) = \underbrace{\int_{\Omega} \int_{\Omega} \dots \int_{\Omega}}_{n-1} L_e(\mathbf{x}_n \rightarrow \mathbf{x}_{n-1}) T(\bar{\mathbf{x}}_n) d\Omega(\mathbf{x}_1) \dots d\Omega(\mathbf{x}_n), \quad (2.3)$$

$$T(\bar{\mathbf{x}}_n) = \prod_{i=1}^{n-1} f_r(\mathbf{x}_{i+1} \rightarrow \mathbf{x}_i \rightarrow \mathbf{x}_{i-1}) G(\mathbf{x}_{i+1} \leftrightarrow \mathbf{x}_i)$$

Takto definovaný příspěvek jedné světelné cesty je již možné aplikovat metodu Monte Carlo, kterou je možné výsledek rovnice aproximovat.

Tento postup se aplikuje na každý pixel ve výsledném obrázku.

2.2 Monte Carlo integrace

K vyřešení rovnice, která definuje světelný příspěvek cesty, je třeba dokázat vyřešit komplikovaný integrál. Analytické řešení nepřichází v tomto případě v úvahu, protože zpravidla není možné nalézt jeho analytické řešení. Je tedy třeba integrál vyřešit numericky. K tomu se zpravidla využívá metody Monte

Carlo [5], případně její modifikace Quasi-Monte Carlo [6]. Pomocí této metody lze dostat pouze přibližný odhad nikoliv však přesný výsledek.

Pro ukázkou Monte Carlo integrace mějme následující příklad. V n -rozměrném prostoru mějme integrál:

$$I = \int_{\Omega} f(\mathbf{x}) d\mathbf{x}, \quad (2.4)$$

kde Ω je podmnožinou \mathbb{R}^n a má objem:

$$V = \int_{\Omega} d\mathbf{x} \quad (2.5)$$

Dále potom mějme sadu N vzorků $\mathbf{x}_1, \dots, \mathbf{x}_N \in \Omega$, které jsou distribuovány rovnoměrným náhodným rozdělením. Potom je možné I aproximovat jako:

$$I \approx Q_N \equiv V \frac{1}{N} \sum_{i=1}^N f(\mathbf{x}_i) = V \langle f \rangle \quad (2.6)$$

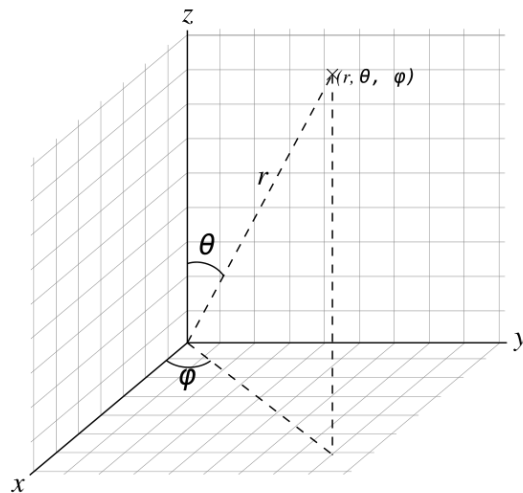
Díky zákonu velkých čísel je pak možné tvrdit že:

$$\lim_{N \rightarrow \infty} Q_N = I \quad (2.7)$$

Jedou z modifikací této metody je Quasi-Monte Carlo. Tato metoda nevyužívá náhodně generované vzorky, ale vzorky jsou deterministické. Tento přístup zamezí negativním vlastnostem náhodných vzorků jako je například shlukování.

2.3 Vzorkování polokoule

Jedním z hlavních problému, které je třeba během *path tracingu* řešit, je problém jakým způsobem najít vzorky, které vhodně pokryjí polokouli. To je třeba řešit při vytváření každého nového paprsku, který vzniká při odrazu, kdy tento paprsek je vystřelen do scény s náhodným směrem.

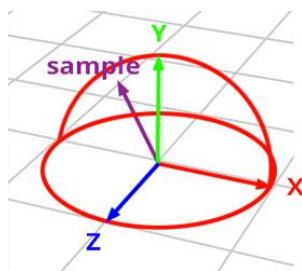


Obrázek 10 – Sférický prostor

Pro vzorkování polokoule je nejvhodnější zvolit sférickou souřadnou soustavu. V té je poloha bodu dána pomocí úhlů φ, θ a poloměru r (obrázek 10). Pro převod do kartézské souřadné soustavy se využívá následujících vzorců:

$$\begin{aligned} x &= r \cdot \sin \theta \cos \varphi \\ y &= r \cdot \sin \theta \sin \varphi \\ z &= r \cdot \cos \theta \end{aligned} \quad (2.8)$$

Hodnoty úhlů se volí v rozmezí $0 \leq \theta \leq \pi$ a $0 \leq \varphi \leq 2\pi$. Hodnota poloměru $r = 1$ pro potřeby vzorkování. Díky tomu získaný vektor bude vektor jednotkový. To je vhodné v další části, kdy bude vektor třeba natočit. Vygenerovaný vektor je umístěn v GCS, viz obrázek 11, a je třeba pak provést transformaci do vypočteného LCS v místě průsečíku.



Obrázek 11 – Vygenerovaný náhodný vektor

Dalším krokem je již zmíněné natočení generovaného vektoru. Vektor je třeba transformovat do lokálního souřadnicového systému, kde osa směřující vzhůru (v globálním souřadném systému je to osa Y) odpovídá normála \mathbf{N} v místě dopadu. V místě dopadu je k dispozici pouze normála \mathbf{N} , která je kolmá k ploše tělesa v místě dopadu paprsku.

Pokud je k dispozici pouze normála, není úplně triviální dopočítat k ní dva kolmé vektory, se kterými vytváří ortonormální prostor. Vyjděme z rovnice plochy:

$$Ax + By + Cz + D = 0 \quad (2.9)$$

Proměnnou D je možné zanedbat, protože ta slouží k posunu plochy oproti počátku souřadné soustavy a v tomto případě je žádoucí, aby plocha procházela středem souřadného systému. Koeficienty A, B, C jsou složky normály a ty se rovnají:

$$A = N_x, B = N_y, C = N_z \quad (2.10)$$

Další známou věcí je tvrzení, že libovolný vektor ležící na ploše, ke které je N normálou, je vektor kolmý k normále N . Takový vektor označme jako N_P . Nyní dosadíme do rovnice plochy $y = 0$, není třeba, aby vektor N_P směřoval tímto směrem:

$$N_x x + N_y \cdot 0 + N_z z = 0, \quad (2.11)$$

po úpravě dostaneme:

$$N_x x = -N_z z \quad (2.12)$$

Tato rovnice platí právě v případě, že $x = N_z$ a $z = -N_x$ nebo $x = -N_z$ a $z = N_x$. Z toho pak dostaneme, že:

$$N_P = (N_z, 0, -N_x) \text{ nebo } N_P = (-N_z, 0, N_x) \quad (2.13)$$

Třetí vektor, který je kolmý k oběma předchozím pak spočteme jako:

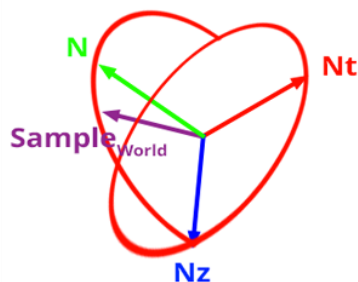
$$N_K = N \times N_P \quad (2.14)$$

Tento algoritmus má jednu nevýhodu v podobě špatné numerické stability, za předpokladu, že $N_x > N_y$. Potom je vhodné do rovnice místo $y = 0$ dosadit $x = 0$.

Výpočet vektoru N_P by potom vypadal takto:

$$N_P = \begin{cases} (N_z, 0, -N_x), & N_x < N_y \\ (0, N_z, -N_y), & N_x > N_y \end{cases} \quad (2.15)$$

Posledním krokem je provést transformaci navzorkovaného vektoru S z globálního souřadného systému do vypočteného lokálního souřadného systému $L_N = \{N, N_P, N_K\}$ jak je vidět na obrázku 12.



Obrázek 12 – Transformace podle *LCS* v místě dopadu

To lze provést pomocí prostého maticového násobení:

$$\mathbf{S}_G = \mathbf{S} \cdot \begin{pmatrix} N_P \\ \mathbf{N} \\ N_K \end{pmatrix} \quad (2.16)$$

Se zde popsaným aparátem je možné vytvořit základní implementaci algoritmu *path tracing*.

2.4 Ukončení algoritmu

Protože se jedná o rekurzivní algoritmus, je nutné stanovit jeho ukončovací podmínku. Je možné zvolit pevnou hloubku zanoření, adaptivní ukončení podle významnosti příspěvku dalšího kroku, nebo použít metodu, která se označuje jako „ruská ruleta“.

„Ruská ruleta“ spočívá v tom, že od jisté hloubky zanoření se provádí náhodné ukončení rekurze. V každé iteraci se volí pravděpodobnost ukončení rekurze, zpravidla se použije pravděpodobnost podle *BRDF* funkce v daném bodě. Ukončovací podmínka pak vypadá následovně:

$$\begin{array}{ll} L_e, & \rho < \alpha \\ \text{další iterace}, & \rho \geq \alpha \end{array} \quad (2.17)$$

3 OPTIMALIZACE RENDERINGU

Na úvod kapitoly bude uveden malý motivační příklad, proč je třeba u zobrazovacích metod založených na metodě *ray tracingu* klást důraz na optimalizaci. Mějme scénu o 1000 těles, kterou chceme vykreslit v rozlišení 800×800 px. Při použití úplně základní metody *ray tracingu* bude pouze pro primární paprsky počet výpočtů průsečíku typu paprsek-těleso roven:

$$800 \cdot 800 \cdot 1000 = 6400 \cdot 10^5 \quad (3.1)$$

Pokud by se jednalo o reálné nasazení, tak by vykreslení probíhalo nejspíše v rozlišení *4K* a počet vypočtených průsečíků by pak byl roven:

$$3840 \cdot 2160 \cdot 1000 = 82944 \cdot 10^5 \quad (3.2)$$

Z této úvahy tak vychází, že kritickým místem celého výpočtu je výpočet průsečíku tělesa s paprskem. Toto tvrzení potvrzují také provedená měření z *profileru*. Je tak třeba se pokusit tyto výpočty co nejvíce optimalizovat.

Optimalizaci lze provádět dvěma způsoby:

- optimalizace samotných výpočetních metod,
- minimalizace počtu volání kritických metod.

Optimalizace výpočetních metod se dá provádět pomocí volby, případně vhodného návrhu samotného algoritmu výpočtu průsečíku paprsek-těleso. To zahrnuje například nahrazování některých matematických (především goniometrických funkcí) vektorovým přístupem vedoucím k řešení stejného problému.

Další možností jak vylepšit samotné výpočty je některé operace provádět za pomoci speciálních instrukcí procesoru. Tyto funkce a k nim příslušné datové struktury je možné explicitně volat v jazyce C++, kde se nacházejí ve standardní knihovně v hlavičkových souborech *<mmintrin.h>*, kde začínají prefixem *_mm_*. Tyto funkce jsou součástí rozšíření základní instrukční sady, které dodávají výrobci procesorů (*MMX*, *SSE*, *AVX*). Nevýhodou tohoto přístupu je, že program se poté stává závislý na volbě kompilátoru a hardwaru. Pokud tedy chceme využívat tyto funkce a zároveň nepřijít o možnosti kompatibility, je nutné tak provádět podmíněný překlad a zároveň produkovat více druhů sestavení programu pro konkrétní hardware.

Dále je možné program vytvářet tak, aby měl kompilátor lepší představu o tom, jak provádět optimalizace, aby se využívalo *SIMD* (*single instruction, multiple data*) přístupu. Tento přístup vychází z toho, že je možné do několika registrů

nahrát různá data a nad nimi zavolat jednu instrukci, která se najednou vykoná nad všemi registry.

Všechny výše uvedené metody přinášejí nějaké možnosti urychlení, nejedná se ovšem o řádové zrychlení výpočtu. Jsou tedy vhodná až pro finální optimalizaci často používaných funkcí, jako jsou vektorové operace typu sčítání, odčítání, vektorové a skalární součin.

K minimalizaci počtu volání kritických funkcí se potom využívá akceleračních datových struktur. Ty zajišťují, aby se prováděly výpočty průsečíků typu paprsek-těleso pouze s tělesy, která leží v blízkosti paprsku a mají tak vysokou pravděpodobnost, že paprsek těleso opravdu protne.

Správně implementovaná akcelerační datová struktura je taková struktura, která poskytuje naprosto stejné výsledky jako výpočet provedený bez ní a to v co nejkratším čase.

Akcelerační datové struktury je možné rozdělit podle toho, jak přistupují k řešení problému na takové, které:

- **dělí objekty** na logické kusy (židle → nohy, opěradlo, sedadlo), nebo na části, které se nachází poblíž sebe. Typickým zástupcem této akcelerační datové struktury je *bounding volume hierarchy* a její modifikace. Tato akcelerační datová struktura se také používá v počítačových hrách, kde ji využívají algoritmy pro detekci kolizí.
- **dělí prostor** scény na menší celky a do nich poté přiřazuje tělesa. Typickými zástupci tohoto přístupu jsou datové struktury *grid*, *kdTree*, *octree* a jejich úplné zobecnění *binary space partitioning*. Tyto datové struktury se využívají v *realtime* zobrazovacích metodách rovněž pro urychlení vykreslení scény.

3.1 Grid

Princip této akcelerační struktury spočívá v tom, že scéna se umístí do trojrozměrné mřížky tvořené z takzvaných *voxelů* (analogie pixelů ve 2D). K traverzování skrze datovou strukturu se potom využívá modifikovaný *DDA* (využívá se pro kreslení úsečky), který je modifikovaný tak, aby dokázal pracovat v 3D prostřední a různými rozměry *voxelů* v každém směru.

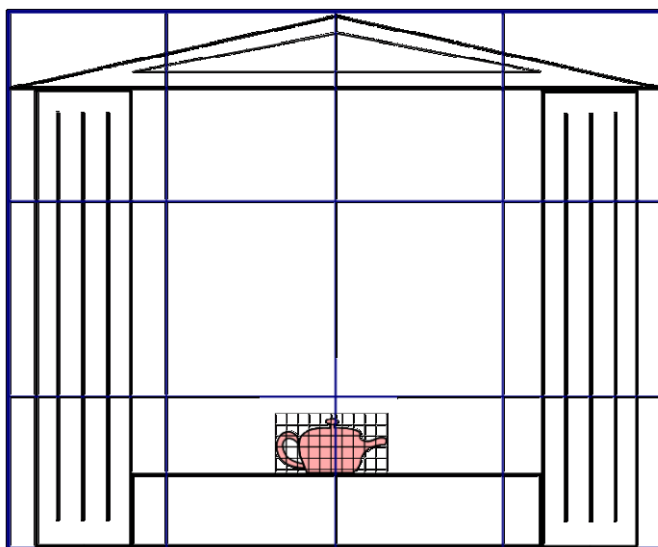
Výhodou této akcelerační datové struktury je její poměrně snadná implementace a poměrně rychlé vybudování celé struktury.

Naopak horší je to s pamětovou náročností, kde je třeba mít v každém *voxelu* poměrně hodně informací a zároveň je *voxelů* velký počet, takže je struktura

poměrně paměťově náročná. To lze řešit například tak, že při budování struktury budou alokovány pouze ty *voxely*, které budou obsahovat tělesa. Při průchodu strukturou toto řešení nijak neomezuje, jen je potřeba kontrolovat, zda byl *voxel* alokovan.

Vzhledem k vysokému počtu alokovaných *voxelů*, je vhodné využít pro alokaci *memory pool*, který nealokuje každý *voxel* zvlášť, ale naráz alokuje větší množství paměti, které potom postupně přiděluje. Sníží se tak počet nutných systémových volání při vytváření datové struktury.

Tato akcelerační datová struktura se hodí pro scény, které mají rovnoměrné rozložení objektů ve scéně. Naopak ve scénách, kde je distribuce objektů nepravidelná tak akcelerační datová struktura podává velmi špatné výsledky.



Obrázek 13 – Nerovnoměrně distribuovaná scéna

Na obrázku 13 je vidět typický problém, který lze v literatuře nalézt pod pojmem *teapot in stadium*, případně *teapot in temple*. Ve scéně velkých rozměrů se vyskytuje malý, ale detailní objekt, který se celý vejde do jednoho *voxelu*. *Renderování* tohoto objektu tak probíhá, stejně, jako kdyby datová struktura neexistovala.

Tento problém lze řešit buď volbou jiné, vhodnější akcelerační datové struktury, anebo vytvořením *nested grid*. To znamená, že pro objekt vytvoříme vlastní akcelerační strukturu, kterou poté vložíme do rodičovské datové struktury.

3.1.1 Vytváření struktury

Prvním krokem je výpočet obalové geometrie (v tomto případě kvádrů) pro všechna tělesa ve scéně. Tím je získáno obalové těleso datové struktury. To je

nalezeno tak, že kolem všech těles ve scéně je nalezeno obalové těleso a ke všem těmto obalovým tělesům je hledáno obalové těleso.

V dalším kroku je třeba zjistit, kolik *voxelů* je třeba alokovat v každém rozměru. Zde je třeba zvolit vhodnou strategii. V diplomové práci je zvolena následující metrika představena v [7]:

$$\begin{aligned} N_x &= d_x \sqrt[3]{\frac{\lambda N}{V}} \\ N_y &= d_y \sqrt[3]{\frac{\lambda N}{V}} \\ N_z &= d_z \sqrt[3]{\frac{\lambda N}{V}} \end{aligned} \quad (3.3)$$

kde N_x, N_y, N_z jsou počet *voxelů* v každém směru, d_x, d_y, d_z jsou rozměry mřížky ve směru osy, N je počet těles ve scéně, V je objem obalové krychle datové struktury. Parametr λ je pak uživatelsky definovaný parametr, který slouží k ladění počtu *voxelů* v datové struktuře. Jak je zkoumáno v [7], je vhodné zde dosazovat hodnoty v intervalu $\langle 3, 5 \rangle$.

V tomto stádiu je již možné začít přiřazovat *voxelům* tělesa, která v nich leží. Těleso s přiřadí všem *voxelům*, které do kterých zasahuje obal tělesa. Toto řešení není ideální, protože mohou nastat případy, kdy těleso může být přiřazeno i do *voxelů*, do kterých sice obalová geometrie zasahuje, nicméně těleso samotné už ne.

3.1.2 Průchod strukturou

Jak již bylo zmíněno, k procházení této datové struktury se využívá modifikovaného algoritmu *DDA*, který je doplněn o třetí rozměr. Při určování nejbližšího průsečíku je nutné určit, kterými *voxely* paprsek prochází. V postupně se v těchto *voxlech* zkouší nalézt nejbližší průsečík. Pokud jsou *voxely* procházeny ve směru paprsku, je možné výpočet ukončit ve chvíli, kdy je nalezen nejbližší průsečík v aktuálním *voxelu*.

Algoritmus *DDA* vychází z předpokladu podobnosti trojúhelníků. Hlavní z vlastností dvou podobných trojúhelníků je, že poměry odpovídajících si stran jsou stejné:

$$\frac{AB}{DE} = \frac{BC}{EF} = \frac{AC}{DF} \quad (3.4)$$

Hlavním problémem *DDA* je určit, v jaké hodnotě parametru t dojde ke změně aktuálního voxelu a ve směru které osy k tomu dojde. Vzdálenost mezi hodnotami parametru t pro přechod do dalšího voxelu je pro paprsek se středem O a směrem d pro každý směr vždy stejná. Označme tedy průsečíky paprsku s jednotlivými hranicemi *voxelů* jako t_x, t_y, t_z . Vzdálenosti mezi jednotlivými průsečíky v každém směru potom označme $\Delta t_x, \Delta t_y, \Delta t_z$. Počáteční hodnoty parametru ve směru konkrétní osy, kdy bude třeba přejít do dalšího *voxelu* nechtě jsou rovny:

$$\begin{aligned}
 t_x &= \begin{cases} \frac{(C_x(t_{min}) + 1)w_x - O_x}{d_x}, & d_x \geq 0 \\ \frac{(C_x(t_{min}))w_x - O_x}{d_x}, & d_x < 0 \end{cases} \\
 t_y &= \begin{cases} \frac{(C_y(t_{min}) + 1)w_y - O_y}{d_y}, & d_y \geq 0 \\ \frac{(C_y(t_{min}))w_y - O_y}{d_y}, & d_y < 0 \end{cases} \\
 t_z &= \begin{cases} \frac{(C_z(t_{min}) + 1)w_z - O_z}{d_z}, & d_z \geq 0 \\ \frac{(C_z(t_{min}))w_z - O_z}{d_z}, & d_z < 0 \end{cases}
 \end{aligned} \tag{3.5}$$

$$O_x = 0 - Gmin_x, O_y = 0 - Gmin_y, O_z = 0 - Gmin_z$$

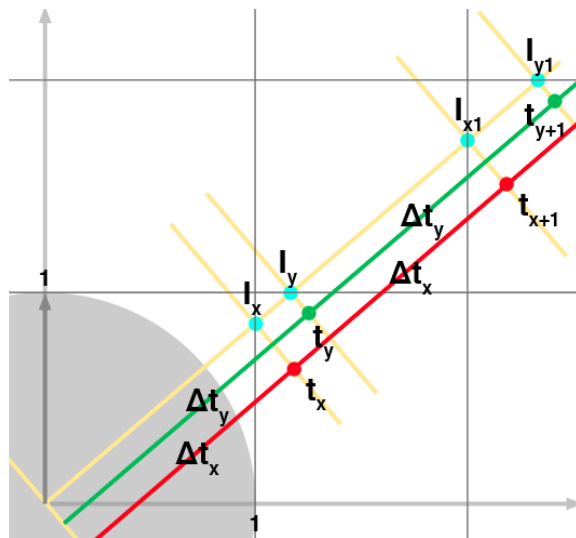
kde $C_x(t), C_y(t), C_z(t)$ jsou funkce, které zjišťují *voxel*, ve kterém se nachází bod na paprsku s parametrem t , ve směru konkrétní osy, w_x, w_y, w_z jsou rozměry *voxelu* ve směru konkrétní osy, d_x, d_y, d_z jsou hodnoty směrového vektoru paprsku ve směrech os souřadného systému a $Gmin_x$ je bod, který reprezentuje nejmenší hodnoty obalového kvádru mřížky.

Vzdálenosti $\Delta t_x, \Delta t_y, \Delta t_z$ nechtě se potom rovnají v závislosti na směru paprsku:

$$\Delta t_x = \pm \frac{w_x}{d_x}, \Delta t_y = \pm \frac{w_y}{d_y}, \Delta t_z = \pm \frac{w_z}{d_z} \tag{3.6}$$

Hodnotu parametru t , kdy dojde k přechodu do dalšího *voxelu*, lze poté vypočítat jako:

$$\begin{aligned}
 t_{x+1} &= t_x + \Delta t_x, \\
 t_{y+1} &= t_y + \Delta t_y, \\
 t_{z+1} &= t_z + \Delta t_z
 \end{aligned} \tag{3.7}$$



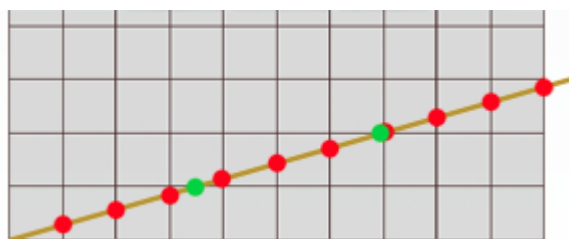
Obrázek 14 – Přechody mezi *voxely*, barevně odlišené v jednotlivých osách

Dále je potom vhodné definovat ukončovací podmínku pro každý směr, když paprsek dokončí průchod strukturou:

$$\begin{aligned} s_x &= \begin{cases} NV_x, & d_x \geq 0 \\ -1, & d_x < 0 \end{cases} \\ s_y &= \begin{cases} NV_y, & d_y \geq 0 \\ -1, & d_y < 0 \end{cases} \\ s_z &= \begin{cases} NV_z, & d_z \geq 0 \\ -1, & d_z < 0 \end{cases} \end{aligned} \quad (3.8)$$

kde NV_x, NV_y, NV_z jsou počty *voxelů* ve směru konkrétní osy.

Nyní je možné realizovat průchod datovou strukturou. *Voxely* jsou poté procházeny v cyklu, který ukončujeme, pokud najdeme nejbližší průsečík, nebo se dostaneme do místa, kde paprsek opouští datovou strukturu.



Obrázek 15 – Průběh algoritmu *DDA*, přechody barevně odlišený

Hledání průsečíku uvnitř tělesa probíhá metodou hrubé síly, kdy je hledán nejbližší průsečík paprsku a tělesa se všemi tělesy, které obsahuje seznam těles v aktuálním *voxelu*.

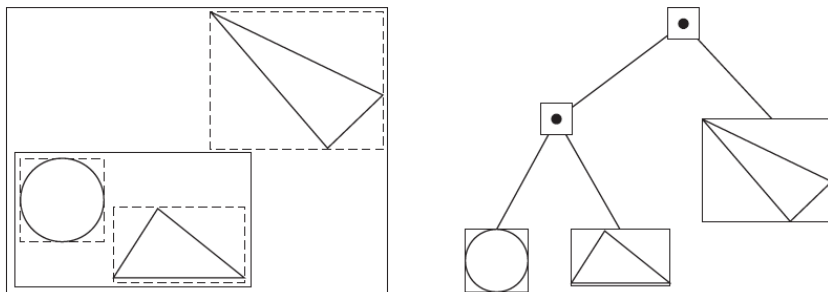
3.2 Bounding volume hierarchy

Bounding volume hierarchy [4] je metoda, jak rozdělit tělesa podle obalové geometrie do hierarchického uspořádání vzájemně disjunktních množin. Nejčastěji jsou tyto množiny uspořádávány do binárního stromu, kde vnitřní uzly reprezentují oblasti ohraničené obalovou geometrií a listy uchovávají tělesa. Podmínka, která platí pro všechny vnitřní uzly, vypadá následovně:

$$\mathbb{B}_L \subset \mathbb{B}_R \wedge \mathbb{B}_P \subset \mathbb{B}_R \wedge \mathbb{B}_P \cap \mathbb{B}_L = \emptyset \quad (3.9)$$

kde \mathbb{B}_R je množina, která reprezentuje rodičovský uzel a $\mathbb{B}_P, \mathbb{B}_L$ jsou množiny, které reprezentují potomky rodičovského uzlu.

Hlavní vlastností *BVH* je fakt, že každé těleso může být v hierarchii pouze jednou. To je rozdíl oproti dříve popsané akcelerační struktuře *grid*, kde jedno těleso mohlo ležet ve více *voxelech*. Důsledkem toho je možné prohlásit, že množství paměti, které potřebuje pro vytvoření *BVH* je shora omezené. Pro binární strom je přesně dané, že pokud list obsahuje jedno těleso, tak počet všech uzlů ve stromě je roven $2n - 1$, kde n je počet těles ve scéně. Počet listu je pak n a počet vnitřních uzlů binárního stromu je $n - 1$. Pokud listy uchovávají více těles než jen jedno pak je paměťová náročnost samozřejmě nižší.



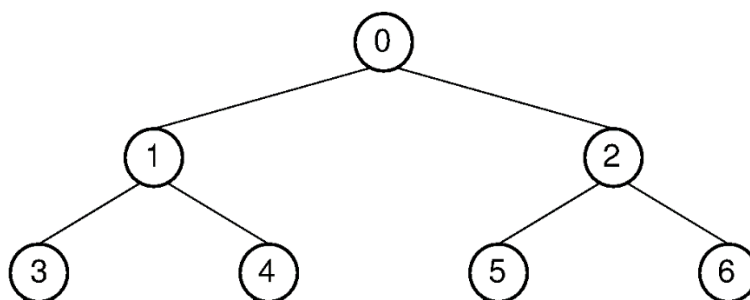
Obrázek 16 – Ukázka jednoduché *BVH*

Čas k vytvoření *BVH* je srovnatelné s akcelerační datovou strukturou *grid*. Tu ovšem velice často předčí rychlostí průchodu strukturou díky tomu, že se *BVH* dokáže lépe přizpůsobit scénám, kde jsou tělesa distribuována ve scéně nepravidelně, vytvářejí shluky apod.

Existují také některé specifické implementace *BVH*. Jako jednoho zajímavého zástupce je možné zmínit *QBVH*. Tento strom je plně přizpůsoben k využívání speciálních *SIMD* instrukcí procesoru. První rozdíl oproti binární reprezentaci *BVH* je fakt, že každý vnitřní uzel obsahuje přesně čtyři potomky. Většina funkcí musí být napsána resp. přepsána pomocí funkcí, které obalují speciální *SIMD* instrukce (například také výpočty průsečíků s geometrií). Díky těmto úpravám je pak možné procházet všechny vnitřní uzly naráz.

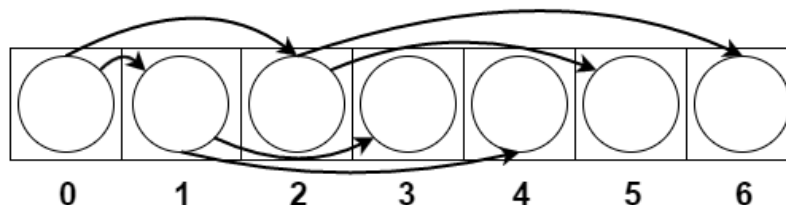
3.2.1 Reprezentace binárního stromu

Reprezentace binárního stromu v operační paměti je možné uskutečnit buď dynamicky, kdy jsou uzly mezi sebou provázeny pomocí referencí nebo ukazatelů. Takto reprezentovaný binární strom neleží v souvislé paměťové oblasti, záleží na tom, kde uzlu systém přidělí místo. Tento přístup je také nevhodný z důvodu optimalizace, protože při přístupu k uzlům není možné uplatňovat princip lokality odkazů, pro ukládání dat do *cache* procesoru.



Obrázek 17 – Dynamická reprezentace binárního stromu v paměti

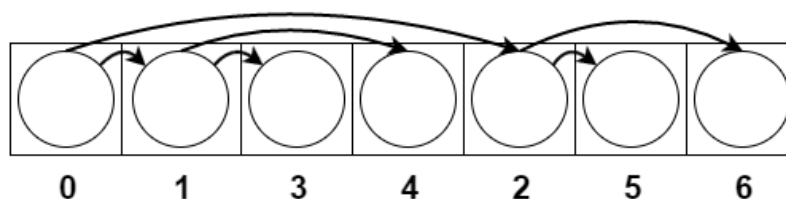
Druhou možností je reprezentovat binární strom jako pole, kde jsou vedle sebe v poli uchovávány uzly ležící v jedné úrovni hierarchie. Obrovskou výhodou této reprezentace je, že není třeba uchovávat žádné ukazatele nebo reference, které odkazují na potomky. Jejich pozici v poli je možné vypočítat jako $2i + 1$ resp. $2i + 2$, kde i je poloha aktuálního uzlu v poli a pole je číslované od 0. Nevýhodou této reprezentace je nutnost častého rozšiřování pole při přidávání uzlů do stromu. Pro použití u zde popisovaných datových struktur je největší nevýhodou fakt, že u nevyvážených datových stromů dochází k plýtvání pamětí pro uzly, které neexistují, ale musí být pro ně v paměti vyhrazeno místo, aby bylo možné nalézt potomky podle výše uvedených vzorců. Tento přístup je velice oblíbenou technikou pro reprezentaci haldy a dalších vyvážených typů binárních stromů.



Obrázek 18 – Reprezentace binárního stromu „do šířky“ na poli

Další možností, která byla využita pro implementaci datových struktur v rámci diplomové práce, je reprezentace na poli „do hloubky“. Levý potomek je vždy umístěn na indexu $i + 1$ kde i je pozice aktuálního uzlu v poli. Pozici druhého potomka je pak nutné uchovávat v aktuálním uzlu. Takovýto strom musí být

vytvářen do hloubky, pokud je již jednou vybudován je obtížné do něj přidávat a odebírat prvky. To ale pro potřeby zde popisovaných datových struktur není potřeba, a strom je budován rekurzivně do hloubky. Další výhodou je, že zde popisované struktury vycházející ze stromů jsou procházeny vždy do hloubky a při tomto uspořádání je možné lépe využít princip lokality odkazů. V neposlední řadě je vždy na konci procesu budování alokován přesně takový počet uzlů, který je skutečně třeba.



Obrázek 19 – Reprezentace binárního stromu „do hloubky“ na poli

3.2.2 Obalová geometrie

Obalovou geometrií nazýváme taková geometrická tělesa, která co nejtěsněji obalují těleso, případně skupinu těles. Tato tělesa se využívají ke zjednodušení komplikovaných tvarů, případně pro vytyčení hranic skupiny objektů. Dva nejdůležitější aspekty obalové geometrie jsou, že k tělesům resp. skupině těles přilehá co nejtěsněji, druhý pak fakt, že takové těleso je třeba získat rychle a efektivně s ním provádět zamýšlené operace. Je nutné uvědomit si fakt, že tyto dvě podmínky jsou vzájemně protichůdné.

Obalová geometrie má své využití nejen v případě zobrazovacích metod, úspěšně se využívá například v herním průmyslu, kde je využívána algoritmy pro detekci kolizí.

Nejčastěji používanými tělesy pro obalovou geometrii jsou kvádr a koule. Další možností je vytvářet obálky pomocí sady obecných ploch, kterými se definují hranice oblasti. Tato metoda je typická spíše pro detekci kolizí než pro zobrazovací algoritmy vzhledem k tomu, že není úplně jednoduché vypočítat s takovým tělesem průsečíky s paprskem.

V diplomové práci je použito jako obalové geometrie kvádrů, jehož strany jsou rovnoběžné s osami souřadného systému (v anglické literatuře jako *axis aligned box*). Právě díky orientaci stran je možné provést značná zjednodušení při výpočtu průsečíku s paprskem. Na problém hledání průsečíku je možné nahlížet jako na hledání průsečíku s tělesem, které vznikne průnikem tří desek. Toto těleso je možné definovat pomocí dvou bodů, kde první reprezentuje minimální hodnoty intervalu v prostoru, který opisuje těleso, druhý naopak maximální

hodnoty. Desku je pak možné se představit jako prostor definovaný mezi dvěma rovinami.

Samotný algoritmus hledání průsečíku vychází z analytické rovnice plochy v trojrozměrném prostoru:

$$Ax + By + Cz + D = 0 \quad (3.10)$$

V obecném případě je pak možné vypočítat průsečík paprsku s plochou jako:

$$t = \frac{-D - (O \cdot (A, B, C))}{\mathbf{d} \cdot (A, B, C)}, \quad (3.11)$$

kde O je počátek paprsku a \mathbf{d} je směr paprsku. Zde je možné odvodit zjednodušení, popis je znázorněn na ploše, která je rovnoběžná s osou X . Protože se jedná o plochu, která je rovnoběžná s osou tak po dosazení do rovnice vyjde:

$$t_1 = \frac{x_1 - (O \cdot (1, 0, 0))}{\mathbf{d} \cdot (1, 0, 0)} = \frac{x_1 - O_x}{d_x}, \quad (3.12)$$

analogicky potom pro vzdálenější plochu:

$$t_2 = \frac{x_2 - (O \cdot (1, 0, 0))}{\mathbf{d} \cdot (1, 0, 0)} = \frac{x_2 - O_x}{d_x}, \quad (3.13)$$

kde x_1 je hodnota minima v ose X a x_2 je hodnota maxima v ose X . Pro zjištění, jestli paprsek protíná krychli, je třeba provést postup analogicky ve všech třech směrech. Na konci každé iterace je pak nutné přiřadit správně hodnotu průsečíku do výstupních parametrů T_1, T_2 , které označují hodnotu parametru na paprsku při průsečíku s krychlí. Počáteční hodnota těchto parametrů je:

$$T_1 = 0, T_2 = \infty, \quad (3.14)$$

podmíněné přiřazení pak vypadá:

$$\begin{aligned} T_1 &= \begin{cases} T_1, & t_1 > T_1 \\ t_1, & \text{jinak} \end{cases} \\ T_2 &= \begin{cases} T_2, & t_2 < T_2 \\ t_2, & \text{jinak} \end{cases} \end{aligned} \quad (3.15)$$

Pokud je při některé iteraci naplněna podmínka, že:

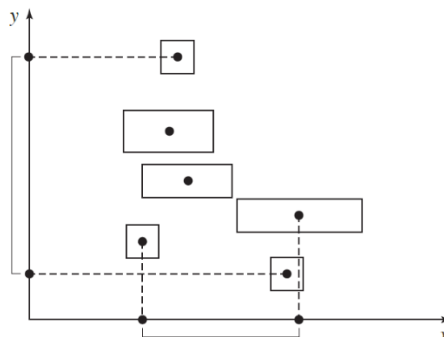
$$T_1 > T_2, \quad (3.16)$$

pak je možné prohlásit, že paprsek neprotíná krychli.

3.2.3 Vytváření struktury

V předchozí kapitole byla popsána obalová geometrie, která je při vytváření *BVH* řazena do hierarchické datové struktury. Při vytváření datové struktury

metodou shora-dolů (od obalu scény po jednotlivá tělesa), tak jak je struktura vytvářena v této diplomové práci, je nutné řešit problém, kterým směrem v jakém poměru rozdělit danou oblast.



Obrázek 20 – Určení osy podle vzdálenosti mezi těžišti krajních těles

Určení směru, ve kterém chceme danou oblast rozdělit, lze provést několika způsoby. Intuitivní metodou je střídat postupně všechny tři osy. Na obrázku 20 je vidět příklad, který demonstruje volbu osy. Tato metoda ale není příliš vhodná, protože je možné, že prostor bude dělen v příslušné ose zbytečně namísto, aby bylo použito dělení podle jiné osy. Dalším přístupem je zvolit směr řezu podle toho, ve které ose je rozměr oblasti maximální.

Pokud již byla vybrána osa, dalším problémem je, ve kterém místě vést řez. V následujících řádcích budou uvedeny dvě základní a jedna pokročilá metoda vedení řezu.

Prvním přístupem je vést řez středem dané oblasti. Tato metoda je nejjednodušší, avšak také nejméně účinná. Často dochází k situacím, kdy je prostor rozdělen ne zrovna optimálně, do jedné části se dostane výrazně menší počet těles než do druhé.

Druhý přístup je, že dělíme prostor tak, aby v obou vytvořených oblastech byl stejný počet těles. To lze provést tak, že tělesa, která jsou v oblasti, budou seřazena do seznamu podle souřadnice té osy, na které bude vykonán řez, a vybere se těleso, které představuje medián. V tomto těžišti tělesa pak bude proveden řez.

Posledním druhem přístupu jsou pak metody založené na *surface area heuristic* [8] a [9]. Většina moderních algoritmů využívá při vytváření *BVH* metodu, která je založena na této myšlence. Myšlenka je vychází z toho, že v hierarchii je možné vytvořit list v libovolné úrovni. Je tedy třeba odhadnout, jestli při průchodu datovou strukturou bude rychlejší projít všech těles v rodičovské

oblasti nebo procházet dva samostatné listy. V případě, že je nutné projít všechna tělesa v oblasti A čas potřebný k vyhledání průsečíku bude roven:

$$c(A) = \sum_{i=1}^{N_A} t_{prus}(i), \quad (3.17)$$

kde N_A je počet těles v oblasti A a t_{prus} je funkce jejíž výsledkem je čas nutný pro výpočet průsečíku s i -tým tělesem.

Na druhé straně je potom cena, kdy bude třeba procházet dva potenciální listy:

$$c(B, C) = t_{trav} + \rho_B \sum_{i=1}^{N_B} t_{prus}(b_i) + \rho_C \sum_{i=1}^{N_C} t_{prus}(c_i), \quad (3.18)$$

kde t_{trav} je cena průchodu datovou strukturou k listům, ρ_B, ρ_C jsou pravděpodobnosti, že paprsek bude muset při svém průchodu strukturou procházet daný uzel, N_B, N_C jsou počty těles v každém z listů a b, c jsou množiny těles v každém listu. Hodnota $t_{trav} > 0$ je z toho důvodu, kdyby při obou variantách došlo ke shodným časům ze sum, tak aby bylo možné rozhodnout o vhodnější variantě.

Pravděpodobnosti ρ_B a ρ_C je možné vypočítat pomocí definice pro geometrickou pravděpodobnost. Pokud je prostor B podmnožinou prostoru A pak pravděpodobnost, že paprsek bude procházet oběma prostory je rovna:

$$\rho(A|B) = \frac{S_B}{S_A}, \quad (3.19)$$

kde S_A a S_B jsou velikosti povrchu tělesa, které obaluje daný prostor.

Prostor je třeba rozdělit podél osy na několik dílů a následně porovnat všechny možnosti. Buď bude nalezen řez s nejnižší cenou a budou vytvořeny dva potomci, nebo bude vyhodnoceno, že průchod listem by byl rychlejší, a bude vytvořen z rodičovské oblasti list. Naivní implementace výpočtu ceny jednotlivých dílů je taková, že pro zjištění ceny jednoho dílu je třeba projít všechny ostatní. Takováto implementace má za následek složitost algoritmu $O(n^2)$, kde n je počet dílů. Existují však modifikace tohoto algoritmu se složitostí $O(n \log^2 n)$ nebo dokonce $O(n \log n)$, viz [10] a [11].

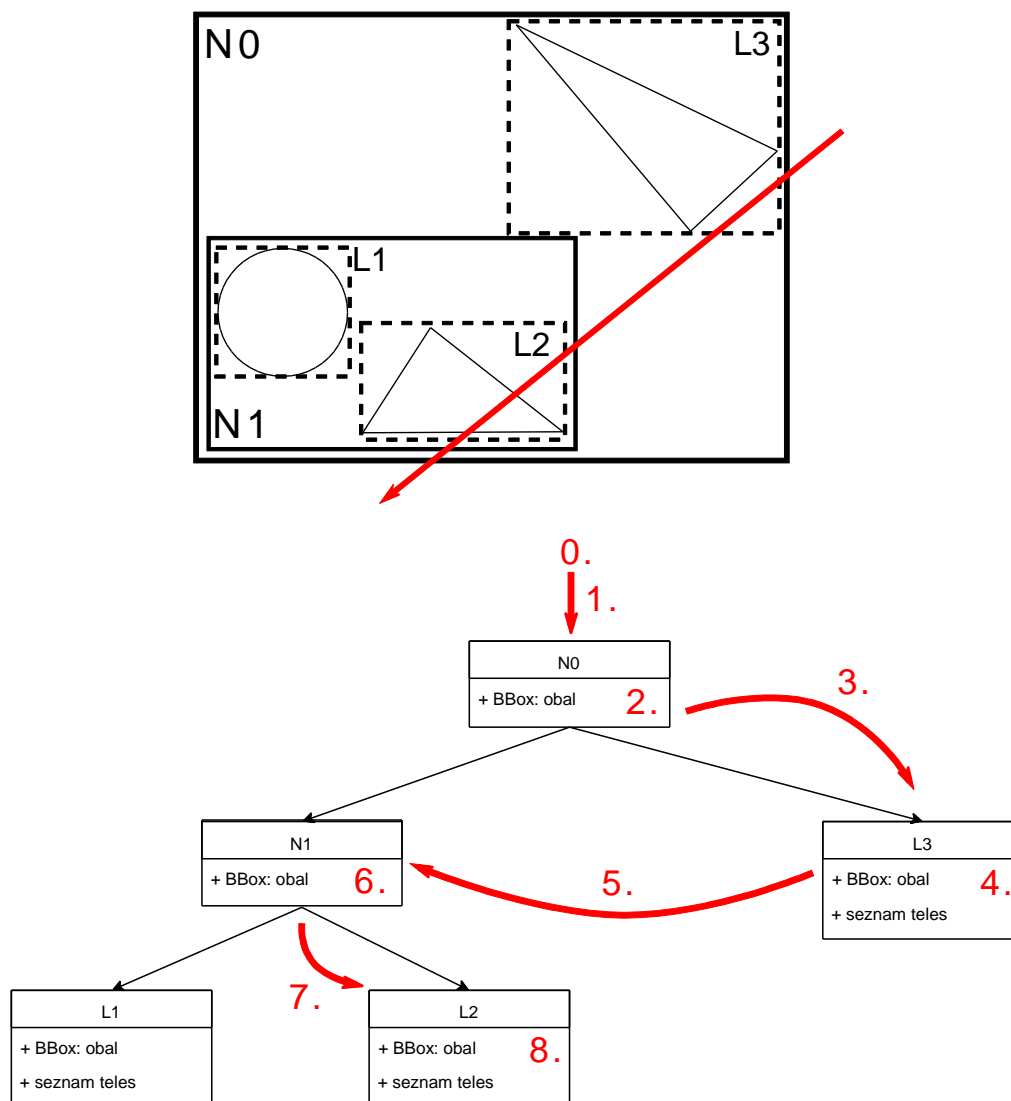
3.2.4 Průchod strukturou

Průchod akcelerační datovou strukturou *BVH* je realizován jako průchod stromem do hloubky. Algoritmus tedy pracuje ve směru od kořene k listům a využívá k tomu datové struktury zásobník. Pokud je zpracováván vnitřní uzel

struktury, tak jsou do zásobníku vloženy jeho potomci. Pokud je zpracováván list, pak jsou otestována všechna tělesa, která mu náleží, jestli nemají průsečík s paprskem.

Při vkládání uzlů do zásobníku se testuje, jestli paprsek protíná obalovou geometrii daného uzlu. Pokud je třeba projít více potomků, pak existují modifikace algoritmu [12], které podle směru určí, kterým potomkem bude paprsek procházet dříve a ten vloží do zásobníku později a dojde tak k jeho dřívějšímu projití. Tyto metody ale nezaručují rychlejší průchod datovou strukturou, jedná se pouze o heuristiku.

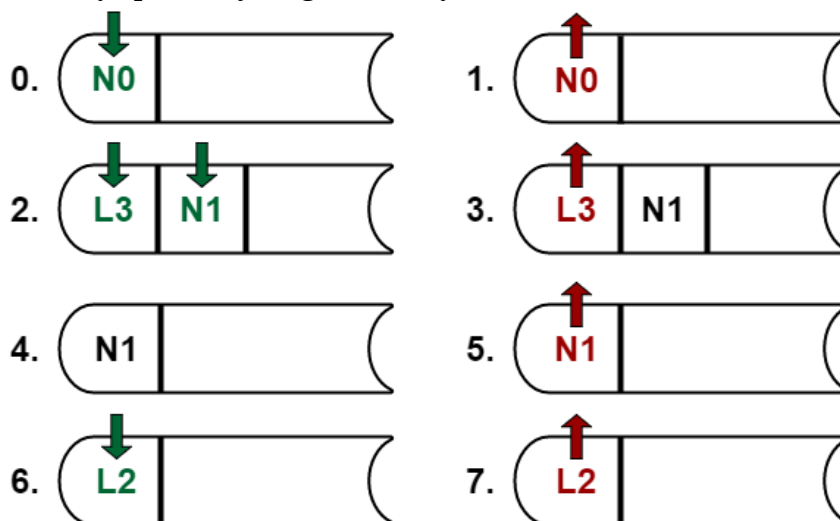
Pro názornou ukázkou je zde předveden průchod přes *BVH* na scéně z obrázku výše. Červenou barvou je znázorněn paprsek, který prochází scénou. Vnitřní uzly jsou značeny jako *N0* a *N1*, listy potom obsahují po jednom tělesu a jsou značeny jako *L1*, *L2*, *L3* a jejich oblast je značena čárkovanou čarou.



Obrázek 21 – Scéna a pořadí průchodu scénou nad binárním stromem

Popis algoritmu průchodu datovou strukturou při hledání nejbližšího průsečíku:

0. Pokud paprsek protíná obalovou geometrii kořene, pak je kořen stromu ($N0$) přidán do zásobníku.
1. Ze zásobníku je odebrán uzel $N0$.
2. Je zpracován uzel $N0$. Nejedná se o list, takže se provádí kontrola, u kterých potomků paprsek protíná obalovou geometrii. V tomto případě se jedná o oba dva potomky. Určí se pořadí, ve kterém se vkládají potomci do zásobníku tak aby byl jako první zpracován uzel, ke kterému dorazí paprsek dříve. V tomto případě je pořadí vkládání do zásobníku $N1$ a $L3$.
3. Ze zásobníku je odebrán uzel $L3$.
4. Je zpracován uzel $L3$. Jedná se o list, tak se provede hledání nejbližšího průsečíku se všemi tělesy, které list obsahuje. V tomto případě nedojde k nalezení průsečíku, algoritmus pokračuje dál.
5. Ze zásobníku je odebrán uzel $N1$.
6. Je zpracován uzel $N1$. Jedná se vnitřní uzel a proběhne kontrola, u kterých potomků paprsek protíná obalovou geometrii. Zde tuto podmínku splňuje pouze uzel $L2$.
7. Ze zásobníku je odebrán uzel $L2$.
8. Zpracovává se uzel $L2$. Jedná se uzel a je provedeno hledání nejbližšího průsečíku se všemi tělesy, která list obsahuje. Průsečík byl nalezen a zásobník je prázdný, algoritmus je ukončen.

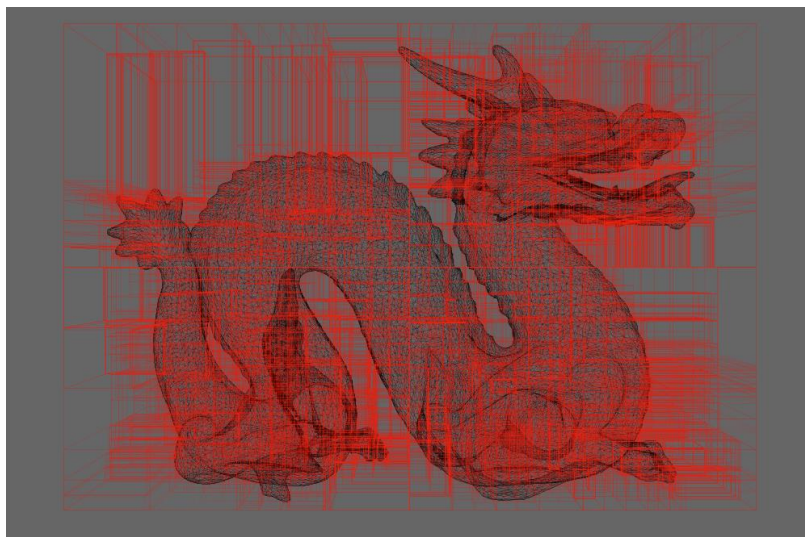


Obrázek 22 – Ilustrace postupného vkládání a odebírání do zásobníku

3.3 Kd-tree

Kd-tree vychází z myšlenky metody *BSP* (*binary space partitioning*), která adaptivně dělí prostor řezy o libovolném směru na nestejně velké oblasti. Díky tomu se tato metoda dokáže vypořádat s nerovnoměrným rozdělením objektů ve

scéně. Nevýhodou využití *BSP* pro metody založené na *ray tracingu* je, že průchod strukturou je velice časově náročný právě díky tomu, že řezy mohou být vedeny libovolným směrem. *Kd-tree* tento přístup zjednodušuje tak, že pro vedení řezů využívá pouze plochy, které jsou rovnoběžné s osami prostoru.



Obrázek 23 – Vizualizace *kd-tree*. Zdroj: <http://blog.yiningkarlli.com/>

Kd-tree je reprezentován jako binární strom. Vnitřní uzly uchovávají oproti klasickému uzlu v binárním stromě ještě informace o směru řezu a jeho poloze. Listy potom obsahují tělesa, která zasahují do vymezeného prostoru. Každé těleso může ležet ve více listech. To je rozdíl oproti *BVH* a není tak možné podle počtu uzlů stanovit velikost datové struktury v paměti.

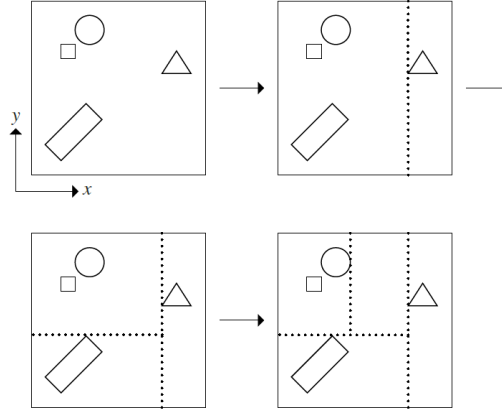
Řezy, jak již bylo zmíněno, jsou vedeny pouze tak, že jsou rovnoběžné s osami souřadného systému. Další nový řez se vytváří pouze v případě, že uzel by měl obsahovat více těles než je přípustné a zároveň nebylo dosaženo maximální hloubky stromu. Umístění řezu je otázkou zvolení vhodné strategie k jeho umístění. Při zvolení správné strategie si algoritmus pro vytváření struktury dokáže poradit i se scénou, kde jsou tělesa distribuována nerovnoměrně a dochází k jejich shlukování.

Alternativou ke *kd-tree* jsou *octree*, které vždy dělí oblast pomocí tří řezů. Tam pak při každém dělení vzniká osm nových oblastí. U těchto vzniklých oblastí se ale může stát, že velká část z nich nebude obsazena žádným tělesem.

3.3.1 Vytváření struktury

Při vytváření akcelerační datové struktury *kd-tree* se zpravidla postupuje směrem od kořene k listům. Vnitřní uzly pak obsahují odkazy na své potomky, listy potom seznam těles. Důležitou vlastností je, že těleso může být přiřazeno

více listům. Je také vhodné si stanovit, jak bude určováno pořadí podoblastí. Jedna z vhodných možností je, říci, že první potomek bude ten, který leží pod řezovou rovinou z hlediska jeho hodnoty na ose kolmé k řezu.



Obrázek 24 – Vytváření *kd-tree*

Pokud se jedná o vnitřní uzel datové struktury, je nutné určit, ve kterém místě a kterým směrem bude veden řez, který bude oddělovat oblasti potomků. Podobně jako u *BVH* je možné využít při budování *kd-tree* různé přístupy:

- dělení podle skutečného středu dané oblasti s tím, že v potomcích nebude stejný počet těles,
- dělení prostoru tak, že bude nalezen mezi tělesy medián v dané oblasti v určitém směru a souřadnice jeho těžiště bude veden řez,
- podobně jako u *BVH* bude využito pro zjištění optimálního vedení řezu *SAH*

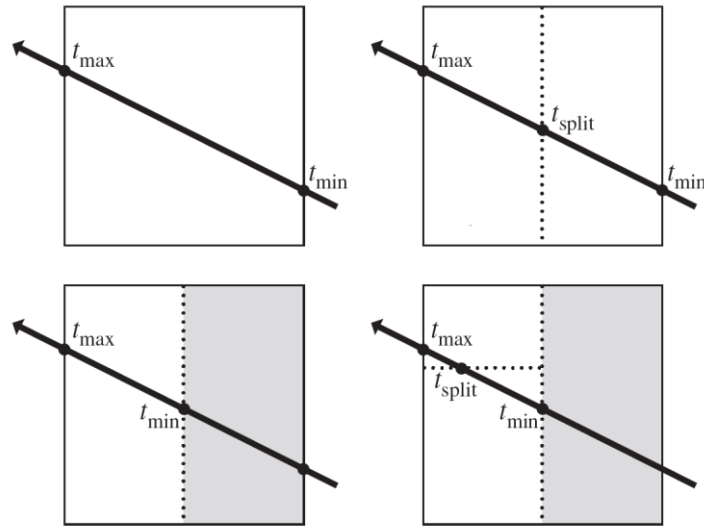
U *kd-tree* bývá metrika *SAH* upravena tak, aby byla dáována přednost takovému vedení řezu, kdy jeden z potomků bude listem a zároveň nebude mít ve svém seznamu žádná tělesa. Toto je dáno tím, že cena za průchod uzlem bývá ve většině implementací *kd-tree* nižší než cena za výpočet průsečíku paprsku s tělesem. Upravená metrika *SAH* rodičovského uzlu pro použití s *kd-tree* pak vypadá:

$$C = t_{trav} + (1 - b_0)(\rho_B N_B t_{prus} + \rho_A N_A t_{prus}), \quad (3.20)$$

kde b_0 je hodnota „ bonusu“, která je různá od 0, pokud jedna z oblastí nebude obsahovat žádná tělesa. Potom je nutné vyzkoušet vybranou množinu řezů v různých směrech a vybrat nejlepší testovanou variantu. Může se však stát, v případě, že u všech testovaných možností budou všechna tělesa náležet oběma novým uzlům, že nebude nalezen žádný vhodný kandidát pro vedení řezu. V tom případě je vhodným řešením vytvořit z oblasti list a vložit do jeho seznamu všechna tělesa z této oblasti.

3.3.2 Průchod strukturou

Průchod datovou strukturou *kd-tree* při hledání nejbližšího průsečíku tělesa s paprskem je založen na průchodu binárního stromu. Na obrázku 23 je vidět zjednodušený postup.



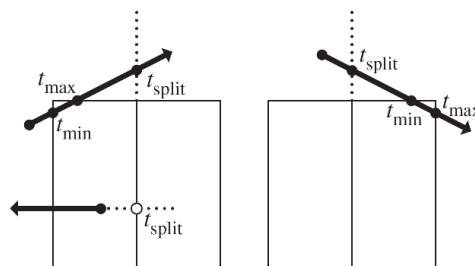
Obrázek 25 – Ilustrace průchodu přes *kd-tree* [13]

Nejprve jsou inicializovány hodnoty t_{max} a t_{min} , které odpovídají průsečíkům paprsku s obalovou geometrií kořene *kd-tree*. V dalším kroku se vypočítá hodnota t_{split} podle následujícího vzorce:

$$t_{split} = (S_{pos} - O_A) \cdot \frac{1}{d_A}, \quad (3.21)$$

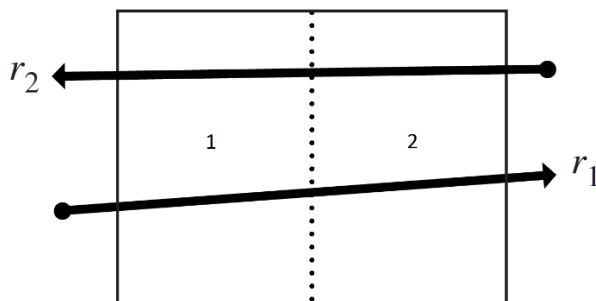
kde S_{pos} je poloha řezu na ose souřadného systému, A je potom osa, ke které je řez kolmý.

Pokud je t_{split} menší než t_{min} , resp. větší než t_{max} , tak průsečík s rovinou řezu se nachází mimo oblast vnitřního uzlu *kd-tree*, nebo počátek paprsku leží v jedné z oblastí a rovinu řezu vůbec neprotíná. Pak je třeba do zásobníku pro procházení přidat jen takové potomky, jejichž oblast paprsek skutečně prochází.



Obrázek 26 – Ukázky případů, kdy je jeden z potomků vynechán [13]

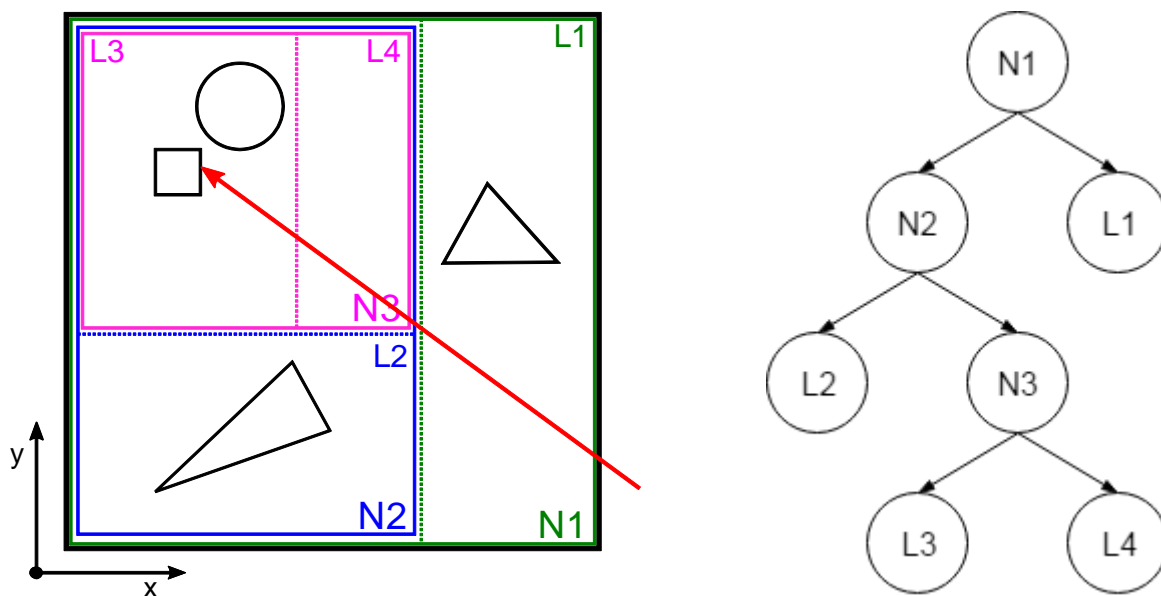
V dalším kroku je nutné určit, kterou z oblastí paprsek navštíví jako první. To je možné určit podle polohy bodu vůči řezové rovině. Na obrázku 27 je vidět různá pořadí vkládání do zásobníku pro paprsky r_1 a r_2 .



Obrázek 27 – Určení pořadí podle osy řezu a počátku

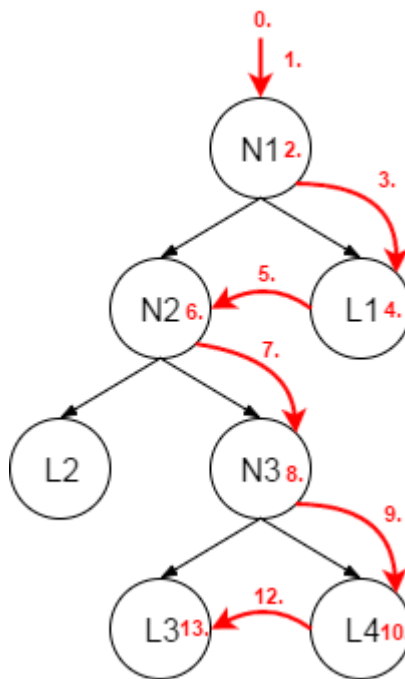
Při vkládání uzlů do zásobníku je nutné ukládat kromě uzlů, které třeba projít také hodnoty t_{min} a t_{max} , příslušící dané oblasti. Pokud algoritmus narazí na list tak při hledání nejbližšího průsečíku iteruje přes všechna tělesa. Algoritmus je ukončen, když je zásobník prázdný.

Jednotlivé kroky algoritmu procházení datovou strukturou *kd-tree* budou předvedeny na následující scéně, kde jsou barevně rozlišeny jednotlivé uzly a k nim příslušné řezy. Pro názornost a přehlednost je ukázka předváděna v 2D. Ve 3D algoritmus probíhá stejně, jen je vše třeba rozšířit pro použití třetího rozměru. Poloha potomků ve stromu je taková, že levý uzel leží pod řezem a pravý nad řezem z pohledu osy pohledu.



Obrázek 28 – Scéna pro ukázkou průchodu strukturou *kd-tree*

Algoritmus pak binární strom prochází do hloubky. Při ukládání uzlů do zásobníku je třeba kromě samostatného uzlu třeba držet ještě také hodnoty t_{min} a t_{max} pro konkrétní uzel.



Obrázek 29 – Průchod přes *kd-tree*

0. Kořen scény je přidán do zásobníku.
1. Kořen scény je odebrán ze zásobníku.
2. Proběhne kontrola, jestli paprsek protíná obalovou geometrii kořene stromu. Při tom se nastaví počáteční hodnoty t_{min} a t_{max} . Dále je vypočítán průsečík s rovinou t_{split} . Kontroluje se, jestli t_{split} leží mezi hodnotami t_{min} a t_{max} . V tomto případě leží a je tak nutné procházet oba potomky. Určí se pořadí procházení potomků, do zásobníku se vloží uzly *L1* a *N2*.
3. Ze zásobníku je odebrán *L1*.
4. *L1* je list, hledá se průsečík se všemi tělesy, které mu náleží. Průsečík není nalezen.
5. Ze zásobníku je odebrán *N2*.
6. *N2* je vnitřní uzel. Je vypočtena hodnota t_{split} . Ta je menší než hodnota t_{min} aktuálního uzlu. To znamená, že to znamená, že uzel pod řezovou rovinou nebude přidán do zásobníku. Do zásobníku je přidán pouze uzel *N3*.
7. Ze zásobníku je odebrán *N3*.
8. *N3* je vnitřní uzel. Je vypočtena hodnota t_{split} . Ta leží mezi t_{min} a t_{max} . Do zásobníku jsou tak ve správném pořadí vloženi potomci. Do

zásobníku jsou vloženy uzly $L3$ a $L4$. List $L4$ je prázdný, pokud je možné zjistit z uzlu tuto informaci, pak není nutné ho vkládat do zásobníku.

9. Ze zásobníku je odebrán $L4$.
10. $L4$ je prázdný, algoritmus pokračuje dále.
11. Ze zásobníku je odebrán $L3$.
12. $L3$ je list a je hledán průsečík se všemi tělesy, která náleží uzlu $L3$. Průsečík je nalezen. Zásobník je prázdný a algoritmus procházení končí.

4 IMPLEMENTACE

Pro implementaci praktické části diplomové práce byl zvolen jazyk C++ v normě z roku 2011. Jako vývojové prostředí bylo použito *Microsoft Visual Studio 2015*. Byla tu však snaha implementovat program multiplatformní, což jde v poslední době naštěstí snáze i díky tomu, že vývojáři kompilátorů více dbají na dodržování norem příslušné verze jazyka.

V diplomové práci bylo implementováno prostředí pro testování výkonových vlastností akceleračních datových struktur. Testovací prostředí je implementováno pouze v textovém režimu. Je to tak ze dvou důvodů:

- nevzniká *overhead* při vykreslování výsledků v reálném čase. Také odpadá nutnost pracné a náročné synchronizace výpočetních vláken s vláknem vykreslovacím.
- Vzhledem k většímu počtu testovacích scén je vhodné mít k dispozici takovou verzi programu, kterou je možné dávkově spouštět a vytvořit tak sadu různých testovacích scén.

Celý výpočet je pak paralelizován pomocí *frameworku OpenMP* [14] a [15]. To s sebou přineslo některé problémy spojené především s nutností provádění synchronizace. Tyto problémy například vznikly při sběru statistik, kdy výpočet probíhal v několika oddělených vláknech, a bylo nutné hledat cestu, jak sumarizovat výsledky z jednotlivých vláken, ideálně bez nutnosti použití synchronizačních primitiv.

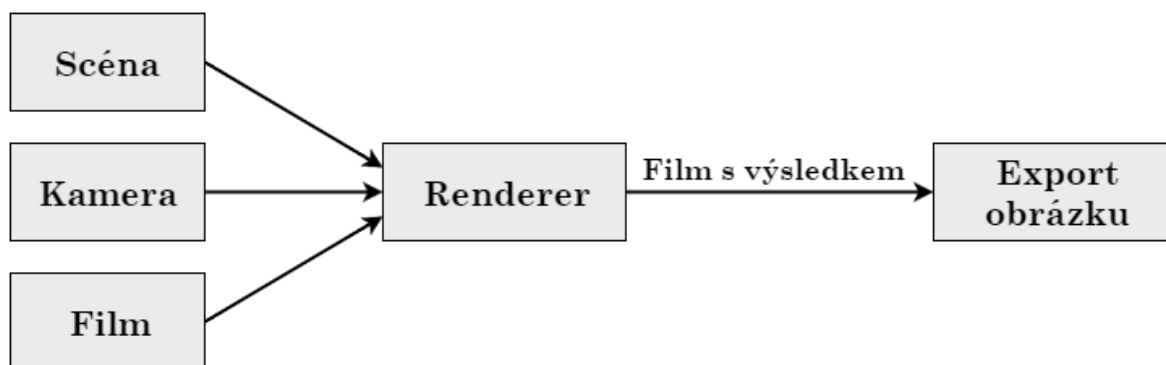
Při implementaci bylo rozhodnuto nevyužívat možnosti správy paměti nabízené programovacím jazykem, jako jsou například *smart pointers* založené na principu počítání referencí. Bylo tak rozhodnuto jak z výkonnostních důvodů a nezátěžování algoritmů akceleračních datových struktur, tak z důvodu čistě tréninkového.

Pro spuštění programu je třeba mít nainstalovanou pouze runtime knihovnu pro *Visual Studio 2015*. Knihovna pro export do *PNG* je linkována staticky.

4.1 Architektura

Architektura programu je poměrně přímočará. Nejprve je ze souboru připravena scéna, kamera a objekt filmu, který je možné si představit jako skutečný film ve fotoaparátu. Tyto objekty jsou pak předány objektu, který zajišťuje vykreslování. Ten vykreslí pohled kamerou na scénu a zapíše tato data v surové podobě na film ve formátu *RGB* s barevnou hloubkou 32 bitů na barevný kanál. Film je pak dále poslán ke zpracování objektu, který zajišťuje export obrázku.

Ten na surová data aplikuje mapovací a filtrovací funkce a poté soubor exportuje do klasického souboru s obrázkem. V diplomové práci byl zvolen formát *PNG*. Bylo tak učiněno z důvodu otevřeného kódu algoritmu, malé velikosti výsledného souboru a podpoře knihoven pro jazyk C++. Program byl napsán modulárně, takže je možné celkem bez problému měnit algoritmy, které se používají pro výpočty. Například není problém napsat modul, který bude zajišťovat export do jiných formátů výsledného souboru (dokonce i ty s plnou barevnou hloubkou jako je například *OpenEXR*¹), další filtrační a mapovací metody, nebo dokonce jiné *renderovací* algoritmy.



Obrázek 30 – Zjednodušený nákres architektury

4.2 Základní třídy

Základem diplomové práce jsou třídy pro práci s vektory. Tyto třídy byly napsány svépomocí. Oproti bakalářské práci bylo rozhodnuto, že pro reprezentaci vektoru, bodu, normály a barvy bude stačit použít jedné třídy. Díky tomu není třeba u některých operací přetypovávat objekt normály na vektor a obráceně. Zároveň jsou však kladeny vyšší nároky na práci programátora, kde je třeba být opravdu pozorný, aby například nedošlo k normalizaci vektoru, který představuje bod apod.

Většina operací těchto matematických tříd je implementována jako *inline* metody a funkce. Tak může kompilátor vkládat tyto funkce do těch částí kódu, kde jsou používány. Díky tomu je možné program optimalizovat poměrně lehkým způsobem, který však nijak neuškodí přehlednosti kódu nebo architektuře programu. Tato optimalizace je vykoupena o něco vyšším časem kompilace a větší velikostí výsledného binárního souboru.

V souboru `./core/Core.h` je možné najít pomocné matematické funkce, které se používají napříč celým programem. Jedná o jednoduché *inline* funkce, které slouží pro zaokrouhlování, převody, 1D interpolace, výpočty základních

¹ <http://www.openexr.com/>

matematických funkcí. V následující ukázce je funkce pro výpočet kvadratické funkce, která přes parametry vrátí hodnoty obou parametrů.

```
inline bool quadratic(float a, float b, float c, float* t0, float* t1)
{
    float disc = b * b - 4 * a * c;

    if (disc < 0.f)
        return false;

    float e = std::sqrt(disc);
    float invLowerPart = 1.f / (2.f * a);

    if (t0)
        *t0 = (-b + e) * invLowerPart;
    if (t1)
        *t1 = (-b - e) * invLowerPart;
    return true;
}
```

V tomto souboru je také třída, která reprezentuje vektory v prostoru. Její možnosti jsou velké, proto je třeba vše používat s rozvahou. V celé práci je tato třída použita k reprezentaci všech trojrozměrných veličin, jako je vektor, normála, bod a barva.

V souborech `./core/Scene.h` a `./core/Scene.cpp` je definována scéna, která se předává *rendereru* k *vyrenderování*. Ta obsahuje ukazatele na akcelerační datovou strukturu a kameru, barvu pozadí, a sadu materiálů. Tento objekt se zároveň také stará, aby byly korektně provolány destruktory všech objektů, které uchovává.

V adresáři `./core` se dále nacházejí také definice rozhraní, která potom musejí již konkrétní objekty implementovat. Najdeme tu tak rozhraní pro materiál, těleso, akcelerační datovou strukturu, kameru a další.

4.2.1 Generátor náhodných čísel

Vzhledem k tomu, že *path tracing* je stochastická metoda, je nutné klást důraz na volbu vhodného generátoru pseudonáhodných čísel. Vzhledem k tomu, že algoritmus často žádá o další pseudonáhodné číslo, tak je nutné, aby byl zvolen kvalitní a zároveň rychlý generátor. Obě tyto podmínky splňuje generátor náhodných čísel *Mersenne Twister*, který je od verze standardu C++11 součástí *STL*. Ten má jak dlouhou periodu opakování ($2^{19937} - 1$), tak velice solidní časovou náročnost.

Vzhledem k tomu, aby byly výsledky měření stejné, bylo třeba v aplikaci využívat jeden generátor náhodných čísel. Navíc je třeba, aby měl na počátku vždy stejně nastavenou hodnotu *seed*. Toho bylo dosaženo vytvořením *wrapperu*

kolem generátoru z knihovny *STL*, který byl vytvořen podle návrhového vzoru *singleton*.

```
class Random {
public:
    static Random* instance() {
        static Random instance;
        return &instance;
    }

    template <class Distribution>
    float next(Distribution& d) {
        return d(g);
    }

    template <class Distribution>
    float operator()(Distribution& d) {
        return d(g);
    }
private:
    Random();
    Random(const Random&) = delete;
    void operator=(const Random&) = delete;
    std::mt19937 g;
};
```

4.2.2 Nástroje pro správu paměti

V souborech *./core/Memory.h* a *./core/Memory.cpp* se nacházejí nástroje pro pokročilou správu operační paměti. K implementaci těchto nástrojů bylo přikročeno z důvodu lepší optimalizace některých výpočtů.

```
void *pathtracer::allocAligned(size_t size, size_t N)
{
#ifdef _MSC_VER
    return _aligned_malloc(size, N);
#elif defined(IS_OPENBSD) || defined(IS_OSX)
    void *ptr;
    if (posix_memalign(&ptr, L1_CACHE_SIZE, size) != 0)
        ptr = nullptr;
    return ptr;
#else
    return memalign(L1_CACHE_SIZE, size);
#endif
}
```

Prvním nástrojem jsou funkce pro alokování resp. *dealokování* zarovnané paměti, které fungují jako *wrappery* nad systémovými voláními. Je to z toho důvodu, že na různých kompilátorech a různých operačních systémech je třeba volat jiné funkce.

Dalším nástrojem je třída, která slouží k zarovnávání svých potomků. Tato šablonová třída přepisuje své operátory *new* a *delete* tak, aby využívali funkci pro zarovnanou *alokaci* resp. *dealokaci*.

```
template <size_t T>
class alignas(T) Aligned
{
    void *operator new(size_t s) { return allocAligned<char>(s, T); }
    void *operator new(size_t s, void *q){ return q; }
    void operator delete(void *ptr) { freeAligned(ptr); }
};
```

Posledním nástrojem pro pokročilou správu paměti je implementován *memory pool*. Tento objekt je vhodný především pro dynamickou alokaci velkého množství malých objektů. Funguje tak, že alokuje jedním systémovým voláním velký blok paměti, který poté dále rozděljuje. Použití je jednoduché stačí vytvořit objekt *memory poolu* a k samotné alokaci pomocí *memory poolu* využít makro, které zavolá operátor *new* se správnými parametry.

4.3 Implementace path tracingu

```
void PathTracer::render() const
{
    auto width = m_film->width();
    auto height = m_film->height();

    #pragma omp parallel for schedule(dynamic)
    for (int r = 0; r < height; ++r) { // po radcich
        for (int c = 0; c < width; ++c) { // kazdy pixel v radku
            float invN = 1.f / m_numberOfSamples;
            Vec3f pixelColor;
            for (int n = 0; n < m_numberOfSamples; ++n) { //kazdy sample
                CameraSample sample;
                sample.x = c + nextUniform();
                sample.y = r + nextUniform();
                Ray ray = m_camera->generateRay(sample);
                pixelColor += radiance(ray, 0) * invN;
            }
            m_film->setPixel(c, r, pixelColor);
        }
    }
}
```

Path tracing je, jak bylo uvedeno v kapitole 2, rekurzivní metoda. Při implementaci takovýchto algoritmů je však vhodné se klasické rekurzi tak, jak ji nabízejí programovací jazyky vyhnout. Její nevýhody jsou zřejmé, při každém dalším zanoření je třeba volat znovu funkci, přepnout kontext a znovu vytvořit všechny lokální proměnné na *stacku*. Je tedy vhodné přepsat rekurzivní algoritmus tak, aby probíhal iterativně bez nutnosti využívat těchto rekurzivních volání. V tomto případě je jednalo o poměrně jednoduchý postup,

kdy bylo využito cyklu, zásobníku a proměnných do kterých se kumulovaly hodnoty.

V dalším kódu je vidět první část algoritmu *path tracingu*, která prochází všechny pixely obrázku a skrze ně náhodně vystřeluje paprsky. Je zde také vidět paralelizace pomocí *frameworku OpenMP*, která se provádí pomocí *#pragma* direktivy preprocesoru. Druhá část je obsažena v metodě *radiance(Ray, int depth)*, která se stará o samotnou integraci celé cesty.

```
Vec3f PathTracer::radiance(const Ray& r_, int depth_) const
{
    Ray r = r_;
    int depth = depth_;
    Vec3f cl(0.f, 0.f, 0.f);
    Vec3f cf(1.f, 1.f, 1.f);
    while (true) {
        Intersection intersection;
        if (!m_scene->intersect(r, intersection))
            return cl + cf.mult(m_scene->background());
        Vec3f color = intersection.material->color();
        Vec3f emission = intersection.material->emmission();
        float p = std::max({ color[0], color[1], color[2] });
        cl = cl + cf.mult(emission);
        if (++depth > m_minDepth) // russian
            if (nextUniform() < p)
                color = color * (1 / p);
            else
                return cl;
        cf = cf.mult(color);

        // cosine hemisphere sampling
        float r1 = 2 * PI * nextUniform();
        float r2 = nextUniform();
        float r2s = sqrtf(r2);

        Vec3f n = intersection.normal;
        Vec3f nl = dot(n, r.d) < 0 ? n : n * -1;

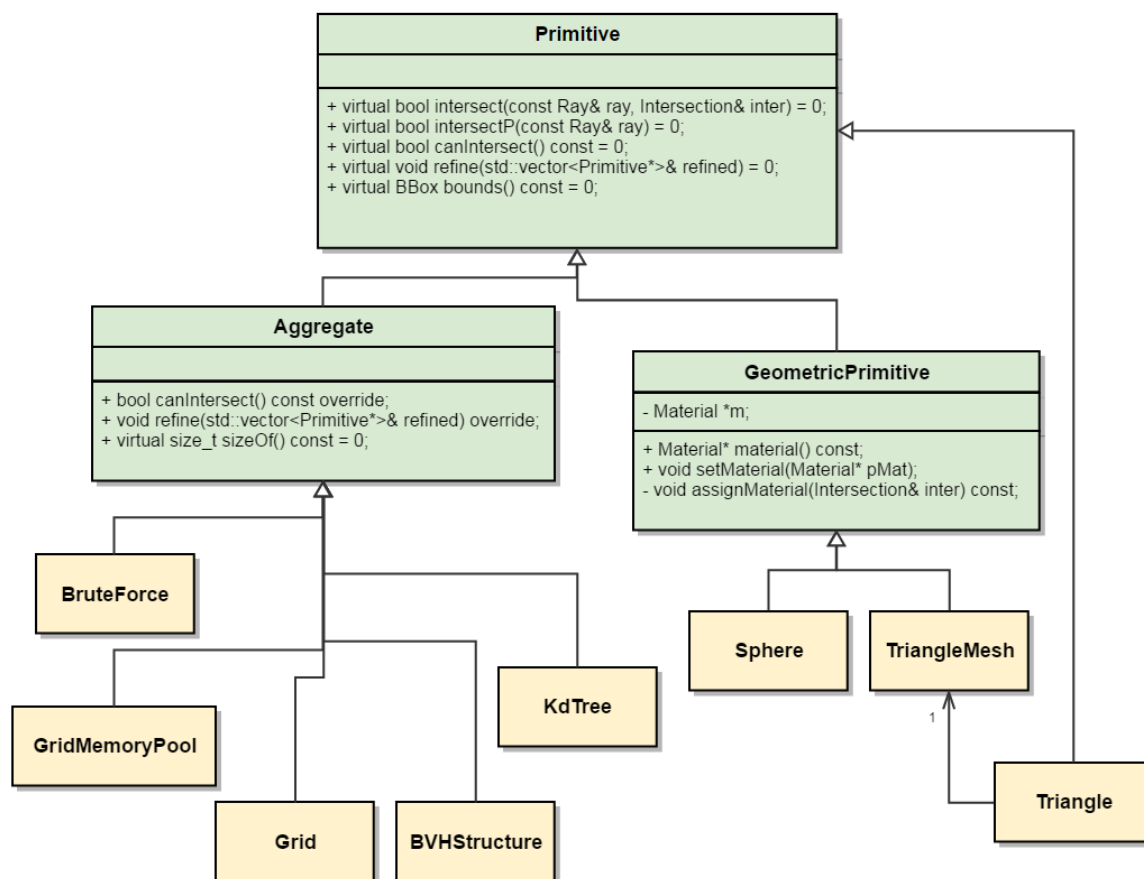
        Vec3f w = nl;
        Vec3f u = cross(fabsf(w.x) > .1f ? Vec3f(0.f, 1.f, 0.f)
            : Vec3f(1.f, 1.f, 1.f), w)
            .normalize();
        Vec3f v = cross(w, u);
        Vec3f dir = (u * cosf(r1) * r2s + v * sinf(r1) * r2s
            + w * sqrt(1 - r2))
            .normalize();
        r = Ray(intersection.hitPoint + EPSILON * nl, dir);
    }
}
```

V metodě *radiance(Ray, int)* je vidět transformace rekurzivní funkce na takovou funkci, která nemusí využívat rekurzi programovacího jazyka. Dále je tam pak

vidět ukončování algoritmu pomocí ruské rulety a vytváření nového paprsku ve směru pomocí kosinového vzorkování.

4.4 Akcelerační datové struktury

Akcelerační datové struktury jsou implementovány tak, že dědí od jednoho společného předka spolu s tělesy. Díky tomu je možné skládat akcelerační datové struktury jednu do druhé.



Obrázek 31 – Diagram tříd pro tělesa a akcelerační struktury²

Třída *Aggregate* nastavuje chování některým metodám, které je společné pro všechny akcelerační datové struktury. Dále zavádí metodu *sizeOf()*, která slouží pro výpočet velikosti aktuální datové struktury. U některých jako je *BVH* je tento výpočet velice prostý, u datových struktur, které dělí prostor a tělesa se tak mohou nacházet ve více listech, to již není tak jednoduchá operace. Je třeba projít celou datovou strukturu uzel po uzlu.

² Třída *Triangle* je poněkud „exotický“ prvek v této hierarchii. Je to z důvodu optimalizace, protože je třeba, aby tato třída dědila z třídy *Primitive*, ale přestože se jedná o těleso tak není vhodné, aby uchovávala ukazatel na materiál, který si může brát z třídy *TriangleMesh*. Do té má přístup k poli bodů a normál, aby se zabránilo duplicitě dat a enormním paměťovým nárokům s tím spojeným.

Datové struktury byly implementovány s ohledem na rychlost a paměťovou nenáročnost. To mnohdy vedlo k poněkud nestandardním postupům v implementaci. Taková řešení byla vždy umísťována tak, aby nebyla dostupná ve vyšších vrstvách programu. Buď byly vytvořeny podtřídy nedostupné vně objektu, nebo byly třídy deklarovány v souborech **.cpp*, které se, podle konvence, pomocí *#include* nevkládají.

4.4.1 Grid

Mřížka je implementována ve dvou variantách. První z nich je v třídě *Grid* a implementuje datovou strukturu bez použití *memory pool* alokátoru. Druhá varianta s *memory pool* alokátorem je pak třída *GridMemoryPool*. Třídy jsou takto rozděleny záměrně, aby bylo možné nezávisle měřit velikost alokovaného prostoru a rozdíl v čase při vytváření datové struktury.

```
// bez memory pool
if (!m_voxels[off]) {
    m_voxels[off] = new Voxel(ptr);
    ++nAllocatedVoxels;
}
// alokace pomocí memory pool
if (!m_voxels[off]) {
    m_voxels[off] = POOL_ALLOC(m_memPool, Voxel)(ptr);
    ++nAllocatedVoxels;
}
```

Vytváření této datové struktury je velice snadná záležitost, jediná zvláštní věc, která stojí za zmínku je ukázka, jak vypadají alokace nového *voxelu* s *memory pool* a bez něj.

```
while (true) {
    int o = offset(pos[0], pos[1], pos[2]);
    Voxel *voxel = m_voxels[o];
    if (voxel) voxel->intersect(ray, inter);

    int bits = ((nextCrossingT[0] < nextCrossingT[1]) << 2) +
               ((nextCrossingT[0] < nextCrossingT[2]) << 1) +
               ((nextCrossingT[1] < nextCrossingT[2]));

    const int cmpToAxis[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };
    int stepAxis = cmpToAxis[bits];
    if (ray.maxt < nextCrossingT[stepAxis]) break;

    pos[stepAxis] += step[stepAxis];
    if (pos[stepAxis] == out[stepAxis]) break;
    nextCrossingT[stepAxis] += deltaT[stepAxis];
}
```

Výrazně zajímavější je potom algoritmus, který zajišťuje průchod datovou strukturou. Ten zajišťuje modifikovaný algoritmus *DDA*. Po výpočtu

počátečních hodnot podle vzorců z kapitoly 3.1.2 následuje samotná smyčka, která zajišťuje průchod.

Zajímavou částí je algoritmu je část, kde se zjišťuje, ve kterém směru bude další průsečík. Lze zapsat buď klasickou cestou pomocí rozvětvených příkazů *if* a *else*. Druhou možností je nahrazení těchto bloků sčítáním a bitovými posuny. V poli hodnot je uloženo, která osa náleží vypočtené hodnotě.

Třída *Voxel* neimplementuje nic jiného než seznam těles a v nich hledání nejbližšího průsečíku.

4.4.2 Bounding volume hierarchy

Při budování *BVH* byl zvolen postup, kdy se nejprve rekurzivním algoritmem vytvoří binární strom v dynamické, který se potom přetransformuje do v paměti zarovnaného pole. Datová struktura pro vytváření v dynamické paměti je zajímavá tím, že implementace používá jednu třídu jak pro vnitřní uzly, tak pro listy.

Zajímavější je struktura, která reprezentuje uzel v poli. Opět je použito techniky, že jedna datová struktura může reprezentovat jak uzly, tak listy. Je to výhodné z toho důvodu, že jinak by muselo být použito dědičnosti a tím pádem virtuálních funkcí, které mají negativní dopad na výkon rychlost běhu programu. Je to dáno tím, že při volání virtuální funkce je nejdříve nutné najít adresu funkce v tabulce virtuálních metod a teprve pak provést samotné volání.

```
struct BVHStructure::LinearBVHNode {
    BBox bounds;
    union {
        uint32_t primitivesOffset;
        uint32_t secondChildOffset;
    };
    uint32_t numberPrimitives;
    uint32_t axis;
    uint32_t pad[2];
};
```

V unii se uchovává buď offset v poli těles, nebo offset na potomka v poli uzlů. Proměnná *pad* pak slouží k „nafouknutí“ velikosti struktury, aby byla v paměti správně zarovnaná. To je také jeden z triků pro zlepšení práce mikroprocesoru s pamětí a tím snížení výpočetního času pro vykreslení.

Pro volbu místa vedení řezu byly implementovány všechny tři strategie uvedené v kapitole 3.2.3.

Při vytváření datové struktury se nepracuje přímo s tělesy, ale jen s jejich obalovou geometrií, těžištěm a ukazatelem na těleso. Ten je tam z toho důvodu,

aby bylo možné tělesa vkládat do seřazeného pole těles, na které se potom odkazují listy.

```
struct BVHStructure::BVHPrimitiveInfo {
    BVHPrimitiveInfo();
    BVHPrimitiveInfo(int pn, const BBox &b);
    int primitiveNumber;
    Vec3f centroid;
    BBox bounds;
};
```

Při vytváření datové struktury se vždy seřadí ta část pole těles, která náleží danému uzlu ve směru jedné z os. Potom je podle zvolené strategie umístěn řez a proces rekurzivně postupuje dál. K této části zde kód není uveden z důvodu jeho značné délky a komplexnosti. Je asi vhodnější podívat se na přiložené CD v příloze B do zdrojového kódu.

Procházení datové struktury probíhá jako průchod binárním stromem do hloubky, kdy potomci uzlů, které paprsek neprotne, nejsou přidáni do zásobníku, který se k průchodu využívá. Pokud algoritmus narazí na list, projde všechna jeho tělesa. Pokud se jedná o vnitřní uzel, tak podle směru paprsku ve stejné ose, jako je veden řez, určí, který potomek se má procházet jako první a v obráceném pořadí je přidá do zásobníku:

```
if (dirIsNeg[node->axis]) {
    todo[todoOffset++] = currentNodeIndex + 1;
    currentNodeIndex = node->secondChildOffset;
} else {
    todo[todoOffset++] = node->secondChildOffset;
    currentNodeIndex = currentNodeIndex + 1;
}
```

4.4.3 Kd-tree

Implementace *kd-tree* je výrazně podřízena paměťové optimalizaci. Lze tam tedy najít poměrně nestandardní postupy.

Kd-tree je budován na poli, bez předchozího vytváření struktury v dynamické paměti. Algoritmus tedy nese odpovědnost za jeho rozšiřování. Datová struktura reprezentující uzly a zároveň také listy se nese v duchu paměťové optimalizace. Pro reprezentaci všech potřebných hodnot byly použity dvě unie. U veřejné unie byly zvoleny tak, aby parametry spolu vzájemně nekolidovaly. U privátní unie spolu kolidují, nicméně je zvolen takový datový typ o vhodné velikosti tak, aby opravdu pokryl celý paměťový prostor. Proměnná *split* určuje u vnitřního uzlu souřadnici, kde je veden řez. Proměnná *onePrimitive* obsahuje číslo tělesa v seznamu, pokud je v listu právě jedno těleso. Nakonec proměnná

primitivesIndicesOffset je pak offset v seznamu těles pokud list obsahuje více těles.

```
struct pathtracer::KdTreeNode {
    float splitPos() const { return split; }
    uint32_t nPrimitives() const { return nPrims >> 2; }
    uint32_t splitAxis() const { return flags & 3; }
    bool isLeaf() const { return (flags & 3) == 3; }
    uint32_t aboveChild() const { return m_aboveChild >> 2; }
    union {
        float split; // vnitřní uzel
        uint32_t onePrimitive; // list
        uint32_t primitivesIndicesOffset; // list
    };
private:
    union {
        uint32_t flags; // oba
        uint32_t nPrims; // list
        uint32_t m_aboveChild; // vnitřní
    };
};
```

Proměnná *flags* si pak rezervuje první dva bity v privátní unii a slouží k rozpoznání, v které ose byl veden řez u vnitřních uzlů (hodnoty 0, 1, 2). Hodnota 3 potom značí, že se jedná o list. Další bity jsou potom rezervovány u listu pro počet těles a u vnitřního uzlu se zde uchovává pozice prvního potomka (pořadí určuje souřadnice řezu). K těmto parametrům jsou vytvořeny obslužné metody, protože je zde velké riziko neúmyslného rozbití zvenčí.

Algoritmus pro budování *kd-tree* využívá modifikovanou heuristiku *SAH* tak, jak je popsána v kapitole 3.2.3 a 3.3.1. Při vytváření uzlu se zjišťuje, jestli je vhodnější možností na základě *SAH*, jestli z oblasti vytvořit list, nebo provést dělení. V tom případě je také nutné určit, v kterém směru a místě vést řez, kterým bude rozdělena oblast. Vzhledem k délce a komplexnosti kódu zde kód není uveden a je možné ho najít v na CD v příloze B.

Průchod stromu je realizován podobně jako u struktury *BVH*. V tomto případě je ale nutností rozpoznat, který z potomků bude procházen jako první.

```
const KdTreeNode *firstChild, *secondChild;
bool belowFirst = (ray.o[axis] < node->splitPos()) ||
                 (ray.o[axis] == node->splitPos() && ray.d[axis] <= 0);

if (belowFirst) {
    firstChild = node + 1;
    secondChild = &nodes[node->aboveChild()];
} else {
    firstChild = &nodes[node->aboveChild()];
    secondChild = node + 1;
}
```


Další rozdíl oproti *BVH* je, že pokud se zjistí, že paprsek neprochází jedním z potomků, tak ten pak nebude přidán do zásobníku.

```
float tPlane = (node->splitPos() - ray.o[axis]) * invDir[axis];
if (tPlane > tmax || tPlane <= 0) {
    node = firstChild;
} else if (tPlane < tmin) {
    node = secondChild;
} else {
    todo[todoPos].node = secondChild;
    todo[todoPos].tmin = tPlane;
    todo[todoPos].tmax = tmax;
    ++todoPos;
    node = firstChild;
    tmax = tPlane;
}
```

4.5 Statistiky

Agregace statistik je v této diplomové práci prostředek, jak efektivně měřit paměťovou náročnost, počty objektů v programu a časy potřebné k vykonání kusů kódu. Největším úskalím implementace bylo vypořádat se s paralelností programu v ideálním případě bez použití synchronizačních primitiv, jako jsou *semaforey* a *mutexy*. Toho bylo možné dosáhnout díky prostředkům nově implementovaných ve standardu C++11. Synchronizace se tak provádí pouze při agregaci statistik a neovlivňuje příliš samotnou dobu výpočtu. Modul statistik se skládá z jednoho globálního *agregátoru* a třídy, která registruje *callbacky*.

```
// Stats macros
#define STAT_COUNTER(title, var) \
    static thread_local int64_t var; \
    static void STATS_FUNC##var(StatsAccumulator &acc) { \
        acc.reportCounter(title, var); \
        var = 0; \
    } \
    static StatRegisterer STAT_REG##var(STATS_FUNC##var)
```

Dále jsou k dispozici makra, která deklarují *thread_local* proměnnou a funkci, kterou zaregistrují jako *callback*. Parametry makra jsou jeho název ve formátu *kategorie/název* a jméno proměnné, která se zaregistruje. Ze jména proměnné se pak také vygenerují funkce a registrační objekt.

Dále existují podobná makra jako pro různé typy úloh. Každý z typů má pak ve výstupu různé formátování tak, aby výstup byl pro člověka čitelný. Časové údaje se tak formátují jako čas, paměť má jednotky v MiB, poměry mají procenta atd. Seznam dalších dostupných maker je možné vidět v následující ukázce kódu.

```
#define STAT_MEM_COUNTER(title, var)
#define STAT_PERCENT(title, m, n)
#define STAT_RATIO(title, m, n)
#define STAT_TIMER(title, var)
```

Agregátor statistik také poskytuje metodu pro tisk do souboru. Ten je formátován podle kategorií a názvů sbíraných statistik. Ten vypadá následně:

```
Statistics:
-----
Scene name: Hairball
Grid
  No. allocated voxels          7417
  No. voxels                    25542
  Construction time             0.01410 s
Memory
  Data structure                1.95849 MiB
Path tracer
  Total rays                    11873891
  RenderTime                    234.56017 s
Triangles
  Count                         79123
```

Jediné omezení, u této techniky je, že všechny proměnné, které jsou označeny klíčovým slovem *thread_local* mohou mít maximální velikost *32kbit*, jinak hlásí linker chybu.

5 MĚŘENÍ

Testování akceleračních datových struktur probíhalo na několika testovacích scénách, které jsou volně dostupné ze stránek *Williams Computer Science, MIT*. Byly zvoleny tři scény postavené na tzv. *Cornell box* scéně, pokaždé s jiným objektem. *Cornell box* je scéna pro porovnávání výsledků *renderovacích* nástrojů s fotografií pořízenou *CCD* snímačem. V těchto testovacích scénách byla akcelerační datová struktura vytvořena pouze nad testovaným objektem, nikoliv nad celou scénou. Další dvě testovací scény mají charakter rozsáhlých scén s nerovnoměrnou distribucí objektů.

Měření byla provedena na sestavě:

- Intel Core i7-4710MQ 2,5GHz, 4 jádra, HT
- 16 GB RAM
- operační systém Microsoft Windows 7 64-bit
- 64-bit sestavení programu s podporou *AVX* instrukcí, maximálního využití *inline* funkcí a *O2* optimalizace
- kompilováno pomocí *MSVC 2015*

Před samotnou prezentací měření je nutné uvést, které hodnoty mají největší váhu při porovnávání jednotlivých datových struktur:

- čas potřebný pro vybudování datové struktury,
- paměťová náročnost datové struktury (zde se nezapočítává paměť využitá samotnými tělesy, na ta se odkazuje pomocí ukazatelů)
- průměrný počet paprsků, které je schopné *path tracer* zpracovat za jednu sekundu

Chybí zde čas potřebný pro výpočet scény. Vzhledem k vnitřní implementaci generátoru náhodných čísel v *STL* dochází k podání mírně odlišných výsledku. Mění se také počet paprsku, pro které musí být vypočítán průsečík. Tak se čas potřebný pro vykreslení scény stává nepřesným údajem, který není vhodný k hodnocení kvality akcelerační datové struktury.

Všechny kombinace scén a datových struktur byly vykreslovány ve více replikacích a hodnoty vypočítány jako jejich průměr, aby došlo k eliminaci nepřesností vzniklých například kvůli procesům běžícím na pozadí.

Typy testovaných akceleračních datových struktur:

- *kd-tree*,
- *bounding volume hierarchy*, strategie kde při dělení je prostor dělen podle souřadnice prostředního tělesa,

- *bounding volume hierarchy*, strategie kde je řez veden středem oblasti
- *grid*
- *grid s memory pool* pro alokaci voxelů

5.1 Jednotlivé scény

První scénou, na které byla prováděna měření kvality implementovaných datových struktur, je scéna založená na *Cornell box* scéně a modelu se jménem *Happy Buddha*. Tento model vznikl pomocí 3D skenování skutečného objektu na univerzitě ve *Stanfordu*. Jedná se o klasického zástupce běžných modelů. Nejsou zde žádné markantní shluky trojúhelníků, ani naopak jejich rovnoměrná distribuce.



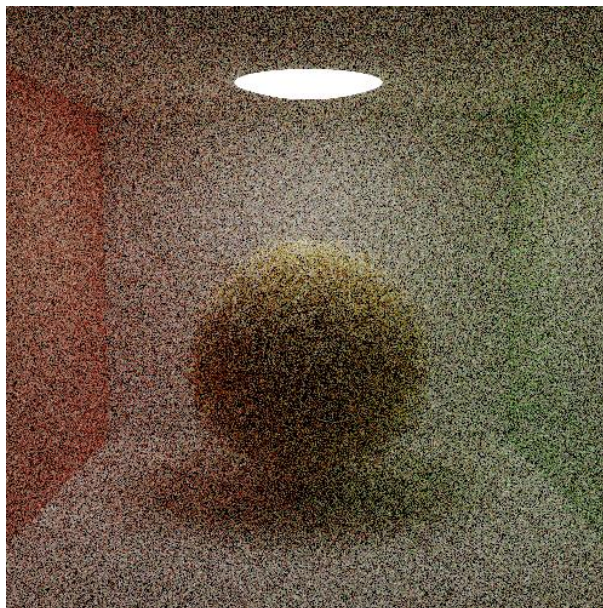
Obrázek 32 – *Happy Buddha*, výstup testu

Scéna byla *renderována* do velikosti 500×500 px, při 50 vzorcích na pixel a minimálně 5 odrazech při rekurzi *path tracingu*. Počet trojúhelníků ve scéně je 1 087 461.

Tabulka 1 – Výsledky měření pro scénu *Happy Buddha*

Akcelerační struktura (strategie)	Čas vytváření [s]	Čas vykreslení [s]	Celkový čas [s]	Počet paprsků	Paprsků za vteřinu	Paměť [MiB]
<i>Kd-tree</i>	8,69	44,18	52,86	10 092 420	228 460	136,30
<i>BVH (CountEqual)</i>	1,15	59,72	60,88	10 102 139	169 148	157,64
<i>BVH (Middle strat)</i>	0,85	52,04	52,88	10 101 363	194 121	157,64
<i>Grid</i>	0,19	183,68	183,86	10 096 558	54 969	16,64
<i>GridMemoryPool</i>	0,17	193,56	193,73	10 100 950	52 186	15,49

Druhá testovací scéna byla opět vytvořena na základě *Cornell box* scény. Do ní byl umístěn objekt *Hairball*. Tento objekt vznikl ve firmě *nVidia* jako model pro testování metod pro vytváření *ambient occlusion* efektu. Tato scéna byla velice náročná pro jakoukoliv datovou strukturu. Jedná se o kouli vytvořenou z tenkých vláken, která má za úkol simulovat vlasy. Tělesa jsou ve scéně rozprostřena rovnoměrně, v tomto testu tak nebyly rozdíly mezi datovými strukturami tak markantní jako u jiných.



Obrázek 33 – *Hairball*, výstup testu

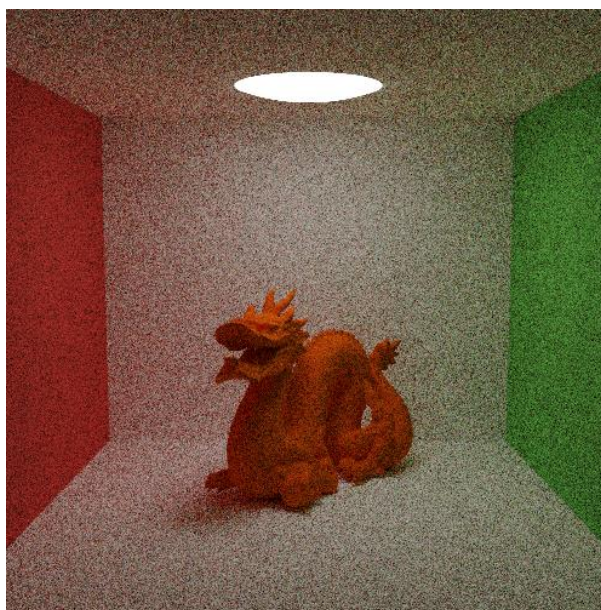
Scéna byla *renderována* do velikosti 500×500 px, při 50 vzorcích na pixel a minimálně 5 odrazech při rekurzi *path tracingu*. Počet trojúhelníků ve scéně je 2 880 010.

Tabulka 2 – Výsledky měření pro scénu *Hairball*

Akcelerační struktura (strategie)	Čas vytváření [s]	Čas vykreslení [s]	Celkový čas [s]	Počet paprsků	Paprsků za vteřinu	Paměť [MiB]
<i>KdTree</i>	75,76	720,07	795,82	11 111 197	15 430	1 045,97
<i>BVH (CountEqual)</i>	3,39	1 080,77	1 084,16	11 106 898	10 276	417,48
<i>BVH (Middle strat)</i>	2,67	834,25	836,91	10 575 635	12 676	417,48
<i>Grid</i>	1,64	867,34	868,98	11 093 635	12 790	276,30
<i>GridMemoryPool</i>	1,57	883,03	884,60	11 100 035	12 570	261,37

Třetí scéna je opět vytvořena na základě *Cornell box* scény. Do ní je vložen model *Dragon*. Ten vznikl jako testovací model pomocí 3D skenování na univerzitě ve *Stanfordu*. Naskenovaný model byl potom přemodelován s menším počtem

trojúhelníků. Tato testovací scéna byla vykreslována s větším počtem vzorků než ostatní.



Obrázek 34 – *Dragon*, výstup testu

Scéna byla *renderována* do velikosti 500×500 px, při 150 vzorcích na pixel a minimálně 5 odrazech při rekurzi *path tracingu*. Počet trojúhelníků ve scéně je 871 316.

Tabulka 3 – Výsledky měření pro scénu *Dragon*

Akcelerační struktura (strategie)	Čas vytváření [s]	Čas vykre- slení [s]	Celkový čas [s]	Počet paprsků	Paprsků za vteřinu	Paměť [MiB]
<i>Kd-tree</i>	6,43	134,01	140,44	34 018 927	253 848	70,65
<i>BVH (CountEqual)</i>	0,98	164,90	165,88	33 511 546	203 220	126,30
<i>BVH (Middle strat)</i>	0,67	147,95	148,63	34 050 313	230 143	126,30
<i>Grid</i>	0,16	301,06	301,22	33 468 750	111 170	20,96
<i>GridMemoryPool</i>	0,18	308,13	308,31	34 052 132	110 512	15,55

Další testovanou je atrium paláce Sponza v Dubrovniku. Tento model používá firma *Crytek* pro ukázky svého *CryEngine*. Model je modifikací původní scény, která se často používá pro testování algoritmů globálního osvětlení. Bylo tu však přidáno více objektů a detailnější geometrie. Jedná se o scénu, kde jsou tělesa distribuována značně nerovnoměrně. Například listy v květináčích proti velikosti geometrie celého objektu. U této scény by se měla ukázat výhoda adaptivních datových struktur jako je *BVH* nebo *kd-tree*. Naproti tomu v nevýhodě by zde měly být akcelerační datové struktury, založené na stejnoměrném dělení prostoru jako je *grid*.



Obrázek 35 – Sponza, výstup testu

Scéna byla *renderována* do velikosti $600 \times 330\text{px}$, při 50 vzorcích na pixel a minimálně 5 odrazech při rekurzi *path tracingu*. Počet trojúhelníků ve scéně je 262 267.

Tabulka 4 – Výsledky měření pro scénu atrium Sponza

Akcelerační struktura (strategie)	Čas vytváření [s]	Čas vykreslení [s]	Celkový čas [s]	Počet paprsků	Paprsků za vteřinu	Paměť [MiB]
<i>Kd-tree</i>	2,66	115,44	118,10	9 404 183	81 460	66,00
<i>BVH (CountEqual)</i>	0,21	241,99	242,20	9 192 989	37 988	38,02
<i>BVH (Middle strat)</i>	0,29	385,83	386,13	9 632 453	24 965	38,02
<i>Grid</i>	0,05	756,89	756,94	9 426 703	12 454	5,81
<i>GridMemoryPool</i>	0,05	745,58	745,63	9 864 430	13 230	5,71

Poslední měřenou scénou je model katedrály sv. Jakuba v chorvatském Šibeniku. Tento model se používá pro testování algoritmů globálního osvětlení. Jedná se o již upravený model. Původní model obsahoval řadu chyb a již není možné ho stáhnout. Jedná se o jednodušší scénu, ovšem střídají se tu různé velké trojúhelníky. Opět by se měly potvrdit lepší výkony adaptivních datových struktur. Vzhledem k tomu, že se jedná o jednoduchou scénu, mohla by se také projevit režie spojená s alokováním pomocí *memory pool*.



Obrázek 36 – Katedrála sv. Jakuba, výstup testu

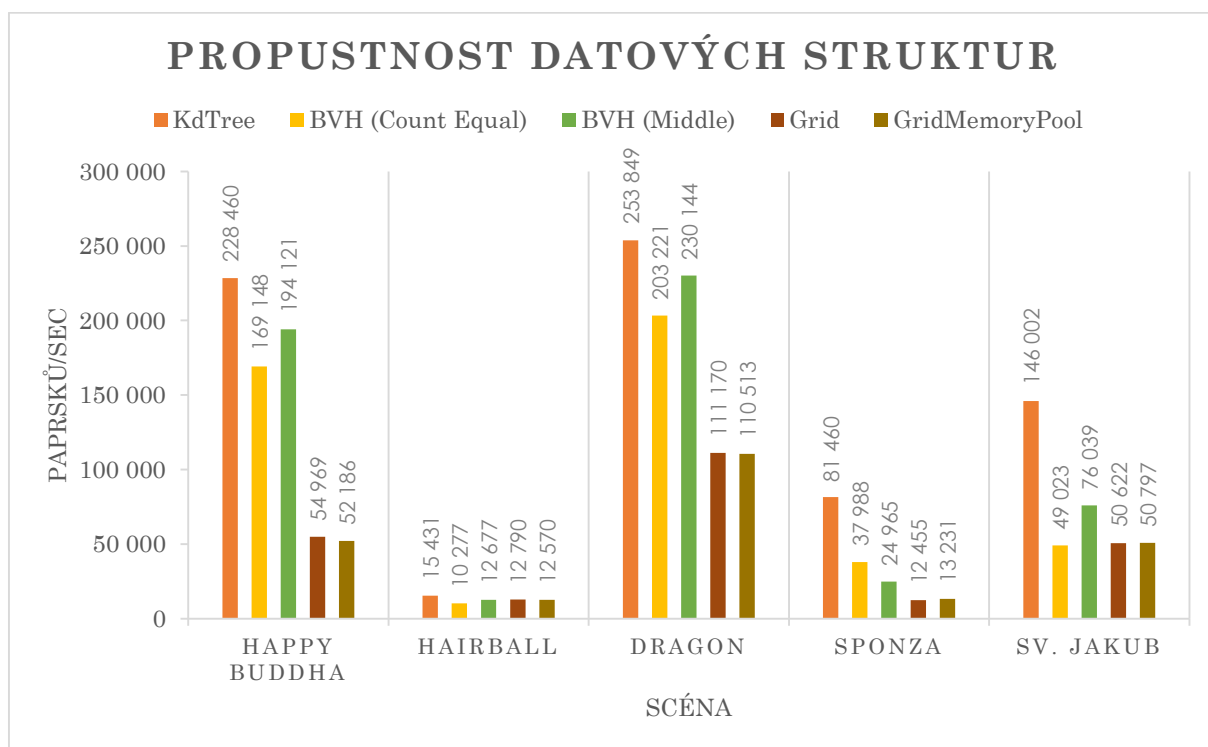
Scéna byla *renderována* do velikosti $330 \times 600\text{px}$, při 5 vzorcích na pixel a minimálně 5 odrazech při rekurzi *path tracingu*. Počet trojúhelníků ve scéně je 79 123.

Tabulka 5 – Výsledky měření pro scénu atrium katedrála sv. Jakuba

Akcelerační struktura (strategie)	Čas vytváření [s]	Čas vykres- lení [s]	Celkový čas [s]	Počet paprsků	Paprsků za vteřinu	Paměť [MiB]
<i>Kd-tree</i>	0,61	82,39	83,00	12 029 619	146 001	16,60
<i>BVH (CountEqual)</i>	0,07	248,61	248,69	12 187 934	49 023	11,47
<i>BVH (Middle strat)</i>	0,29	160,28	160,58	12 187 934	76 039	11,47
<i>Grid</i>	0,01	234,56	234,57	11 873 891	50 621	1,96
<i>GridMemoryPool</i>	0,02	243,16	243,18	12 352 003	50 797	1,84

5.2 Závěry měření

První z prezentovaných grafů je graf propustnosti datových struktur. Jednotkou je počet paprsků za sekundu. V jednom grafu jsou zahrnuta měření ze všech scén.

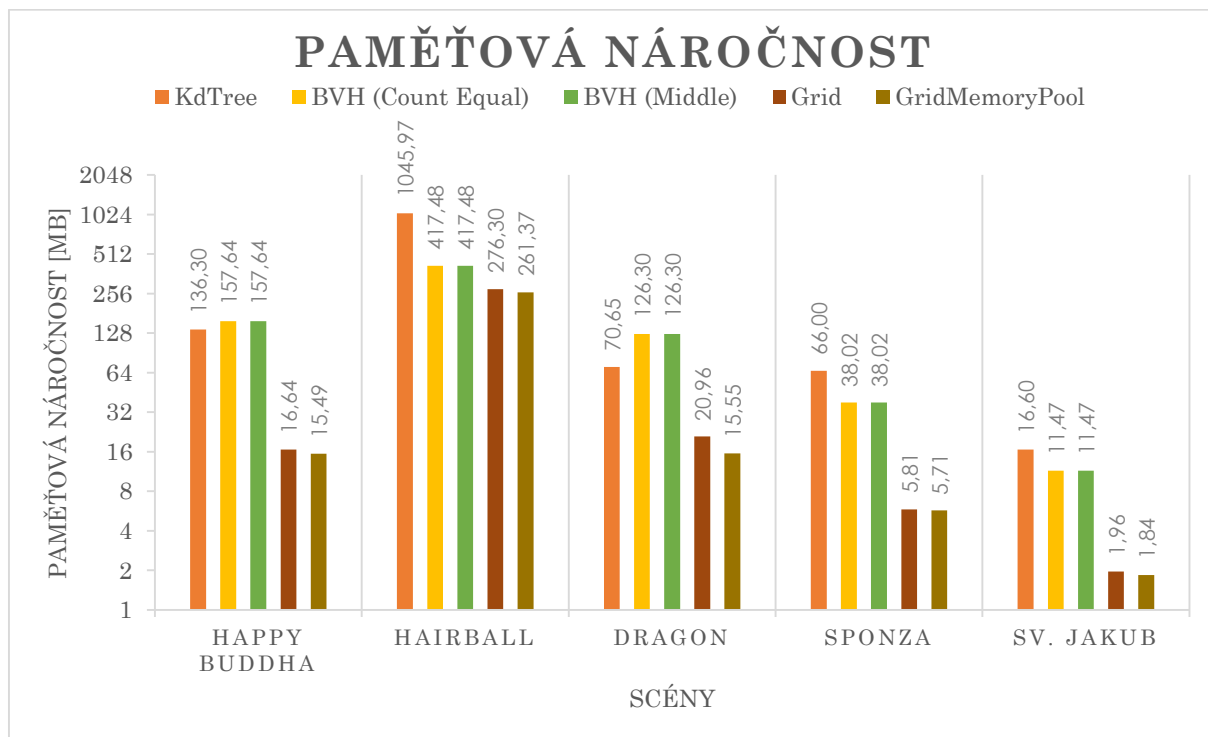


Obrázek 37 – Graf propustnosti datových struktur

Z grafu je patrné, že ve všech scénách, kromě scény *Hairball*, podává nejlepší výsledky datová struktura *kd-tree*. Je to dáno jednak aplikací *SAH* při vytváření a také tím, že hloubka stromu je adaptivní a není shora omezena. Naproti tomu *BVH* má v této implementaci omezenou hloubku, protože k jeho vytváření je použito rekurze pomocí klasického volání jazyka. Pokud by tomu tak nebylo, docházelo by u větších scén k přetečení zásobníku. Naopak nejhorších výsledků dosahovaly obě varianty datové struktury *grid*. Jednak je to dáno charakterem datové struktury a pak také vnitřní implementací datové struktury, kdy každý *voxel* je samostatný objekt a při průchodu přes všechna jeho tělesa je třeba volat metodu tohoto objektu.

Samostatnou kapitolou je scéna *Hairball*. Tam dosahují všechny datové struktury přibližně stejných výsledků. Je to dáno charakterem scény. Distribuce objektů je v této scéně poměrně rovnoměrná. Rozdělení prostoru pomocí datových struktur s adaptivním přístupem se tak blíží mřížce.

Druhý graf ukazuje, kolik paměti potřebují akcelerační datové struktury. Opět jsou v grafu všechny scény najednou. Pro tento graf bylo zvoleno logaritmické měřítko osy při základu logaritmu 2.



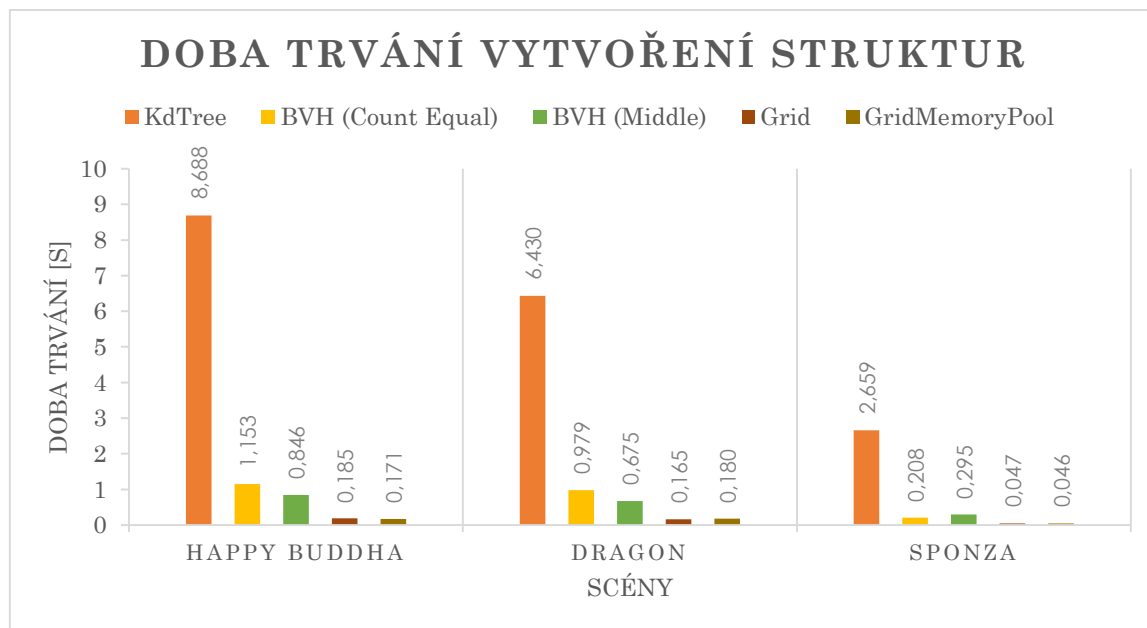
Obrázek 38 – Graf paměťové náročnosti datových struktur

Paměťová náročnost datových struktur je úměrná náročnosti scény a počtu těles v ní. Z grafu se potvrzuje, že nejnižší paměťovou náročnost mají obě datové struktury *grid*. Dále se potvrzuje, že při alokaci souvislých zarovnaných bloků paměti dochází zároveň také k úspoře paměti.

Jako nejvíce náročná paměťová struktura se ukázal ve většině případů *kd-tree*. Ve dvou případech tomu tak není, je to dáno aplikací *SAH*, a tak v některých případech už nedochází k dělení oblastí na menší.

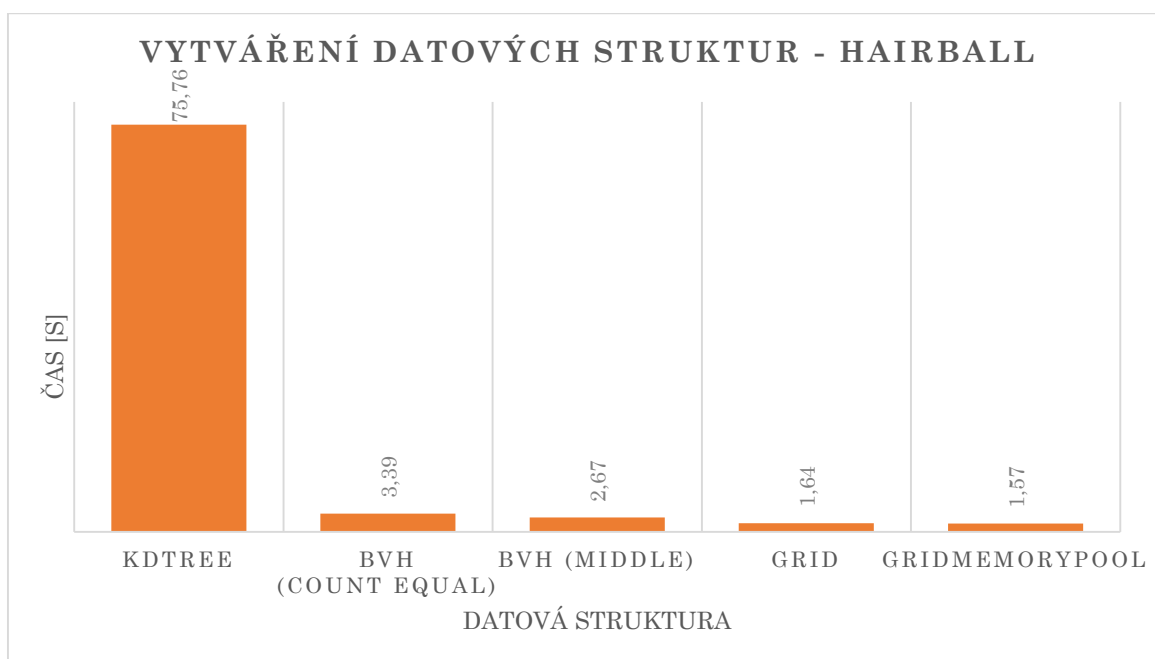
Dalším grafem je pak graf, kde jsou znázorněny časy vytváření datových struktur. Zde jsou vynechány scény *Hairball*, která bude popsána později, a katedrála sv. Jakuba, která je pro svoji jednoduchost nezajímavá, časy jsou si velmi podobné.

Z tohoto grafu se opět potvrzují domněnky vyslovené v předchozích kapitolách. Nejrychleji je vytvořena datová struktura *grid*, v kombinaci s *memory pool* alokací dosahuje zpravidla o něco málo lepších výsledků. Doba konstrukce *BVH* závisí na použité strategii, při strategii *count equals* je třeba řadit tělesa dle souřadnice a to se zpravidla projeví na čase.



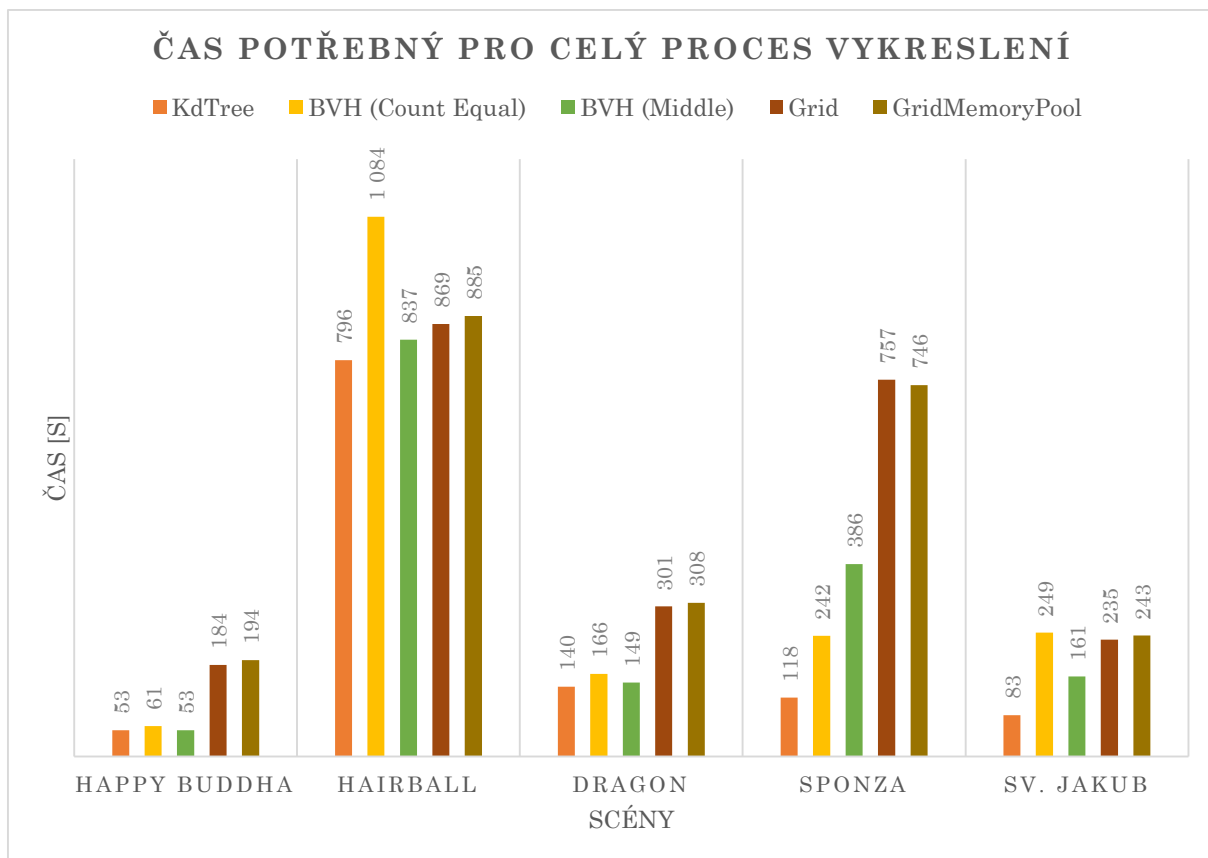
Obrázek 39 – Graf doby trvání vytváření datových struktur

Doba vytváření struktury *kd-tree* odpovídá tomu, že při implementaci SAH bylo použito naivního algoritmu se složitostí $O(n^2)$. Markantní rozdíl je vidět na Obrázek 40.



Obrázek 40 – Graf časů tvorby datových struktur nad scénou *Hairball*

Poslední graf porovnává celkové doby vykreslení obrázku. Zde je však problém s tím, že *path tracing* je stochastický algoritmus a výsledky prezentované v tomto grafu jsou zatíženy velkou chybou.



Obrázek 41 – Graf časů potřebných pro vykreslení scén

V grafu lze vidět, že *kd-tree* i přes vysokou časovou náročnost při vytváření podává stabilně nejlepší výsledky. Zvláštní jsou však dva výrazné výpadky *BVH* se strategií dělení *count equals* ve scénách *Hairball* a katedrála sv. Jakuba. V těchto případech byla provedena ještě další měření, která dopadla s podobnými výsledky. U scény *Hairball* jsou vidět nepříliš rozdílné časy (kromě již zmíněného výpadku *BVH*). *Grid* naopak dosahuje zpravidla nejhorších výsledků.

Z výsledků je možné říci, že lépe si vedou zpravidla adaptivní datové struktury. Jejich volba zpravidla a správné nastavení zpravidla zaručí solidní výsledky. V některých specifických scénách, jako byla v těchto měřeních scéna *Hairball*, však není rychlost zpracování mezi všemi typy akceleračních struktur příliš významný. Akcelerační datovou strukturu pro *renderování* 3D scény je tak třeba vybírat na základě distribuce těles ve scéně, počtu těles a nelze zanedbat ani zkušenosti daného člověka.

6 ZÁVĚR

V diplomové práci byly úspěšně naimplementovány datové akcelerační struktury využívané pro 3D *rendering*. Jednalo se struktury *grid*, *bounding volume hierarchy* a *kd-tree*.

Datová struktura *grid* byla implementována ve dvou verzích. První verze využívá při alokování *voxelů* standardní možnosti alokace v dynamické paměti, které jsou dostupné v jazyce C++. Druhá verze pak využívá při vytváření pro alokování *voxelů* autorem napsaný alokátor, který využívá přístupu *memory pool*. Tento přístup spočívá v tom, že pro velké množství malých datových struktur se dopředu alokují větší zarovnané bloky paměti a z těch je poté paměť přidělována objektem alokátoru. To má za následky snížení počtu systémových volání, která jsou časově náročná. Jako algoritmus procházení je pak implementováno *DDA*, která je rozšířeno o třetí rozměr.

V datové struktuře *BVH* byly implementovány tři různé postupy při vytváření. První přístup je dělení oblasti podle mediánu těžiště těles seřazených podle osy kolmé k rovině řezu. Druhý přístup dělí oblast v geometrické polovině na nejdelším rozměru obalové geometrie. Třetí přístup využívá heuristiku *SAH*. Datová struktura je nejprve vytvářena rekurzivním algoritmem v dynamické paměti a následně transformována do pole přístupem „do hloubky“. Pro procházení datovou strukturou byla implementována heuristika, kdy podle směru paprsku je určeno, který potomek má být projit jako první.

Akcelerační datová struktura *kd-tree* byla implementována rovnou nad polem, bez předchozího budování struktury v dynamické paměti. Při vytváření datové struktury byla využita heuristika *SAH*, která je mírně modifikovaná oproti struktuře *BVH*. U *kd-tree* zohledňuje „bonusy“ za vytvoření prázdného listu a dále také výrazně zohledňuje vyšší cenu za průchod potomky oproti průchodu nad všemi tělesy oblasti. Průchod je pak realizován algoritmem, který postupně prochází uzly a vyřazuje ty, které nezasahují do cesty paprsku. Dále také ignoruje prázdné listy, které mohou vznikat při použití *SAH*.

Pro testování paprsků byla implementována metoda *path tracing*. Tato metoda byla zvolena z toho důvodu, že pro aproximaci zobrazovací rovnice využívá numerické integrace pomocí metody Quasi-Monte Carlo. Bylo tak zajištěno komplexní testování všech datových struktur. Metoda *path tracing* byla implementována pomocí vlastní implementace rekurze pomocí kumulace hodnot výsledné barvy. Díky tomu byly odstraněny nedostatky klasické rekurze, jako je možnost přetečení zásobníku a pomalé vykonávání funkcí při implementaci,

kterou poskytuje programovací jazyk. Dále byl celý *renderovací* proces paralelizován pomocí *frameworku OpenMP*.

Celá aplikace byla psána modulárně, není tak problém vytvářet další datové struktury a pro testování implementovat další *renderovací* metody. Na pozadí těchto metod byla implementována sada funkcí a objektů, které zajišťuje práci s vektory, vektorovými prostory a lineární algebrou. Možností jak rozšířit tuto práci je propojit ji s *GUI* a upravit jádro pro *realtime* vykreslování. Další možností je například portování aplikace na grafickou kartu, která je vhodná pro výpočet silně paralelizovatelných úloh, jako je právě *path tracing*.

Pro sběr statistik o průběhu *renderovacího* procesu a informacích o datových strukturách byl vytvořen speciální modul. Ten využívá pro své použití makra, aby nebylo třeba psát mnoho duplicitního kódu. Vzhledem k paralelizaci celého procesu výpočtu bylo nutné provést synchronizaci zápisu do *agregátoru* statistik. Ta se díky novým možnostem jazyka C++11 musela provádět jen ve velmi omezeném rozsahu. Sběr statistik tak příliš neovlivňoval dobu potřebnou pro *rendering*.

Měření bylo provedeno na pěti různých scénách. Důraz byl kladen především na rozmanitost scén, co do počtu trojúhelníků, tak co se týče jejich distribuce ve scéně. Byly použity scény, které se běžně používají pro měření ve světových výzkumech a vědeckých pracích. Byly mezi nimi také scény pořízené pomocí 3D skeneru. Scény lze charakterizovat jako objekty umístěné do interiéru, nebo jako exteriérové scény.

Při měření byly zaznamenávány hodnoty tykající se časové náročnosti výpočtu, výkonu datových struktur, který byl měřen veličinou počet paprsků za sekundu, a paměťová náročnost datových struktur. Záznamy byly zpracovány do přehledných tabulek. Dále byly vytvořeny názorné grafy, ze kterých je možné přehledně porovnat výkonnost a paměťovou náročnost akceleračních struktur.

Z měření byla potvrzena hypotéza, které byla položena před začátkem práce na diplomové práci. Bylo předpokládáno, že lépe si budou vést adaptivní datové struktury za cenu vyšší paměťové náročnosti. To bylo potvrzeno na většině scén. Ve scéně *Hairball* však všechny datové struktury podávaly velice podobné výsledky. To bylo dáno rovnoměrnou distribucí těles ve scéně. Výraznou převahu měly adaptivní datové struktury především ve scénách paláce Sponza a katedrály sv. Jakuba. Tam je distribuce trojúhelníků velice nerovnoměrná a je zde velký výskyt dlouhých trojúhelníků, které se táhnou velkou částí scény.

7 POUŽITÁ LITERATURA

- [1] **LOKVENC, Pavel.** *Zobrazování 3D scény metodou raytracingu.* Pardubice, 2014. Bakalářská práce. Univerzita Pardubice, Fakulta informatiky a elektrotechniky, Katedra informačních technologií. Vedoucí práce Ing. Petr Veselý.
- [2] **SHIRLEY, P. a R. MORLEY.** *Realistic ray tracing.* 2nd ed. Natick, Mass.: AK Peters, 2003. ISBN 15- 688-1198-5.
- [3] **SUFFERN, Kevin G.** *Ray tracing from the ground up.* 1. vyd. Wellesley, Mass.: A K Peters, 2007. ISBN 15-688-1272-8.
- [4] **KAJIYA, James T.** *The rendering equation.* Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86. New York, New York, USA: ACM Press, 1986, , 143-150. DOI: 10.1145/15922.15902. ISBN 0897911962.
- [5] *Quasi-Monte Carlo method.* In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2016-05-11]. Dostupné z: https://en.wikipedia.org/wiki/Quasi-Monte_Carlo_method
- [6] *Metoda Monte Carlo.* In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2016-05-11]. Dostupné z: https://en.wikipedia.org/wiki/Quasi-Monte_Carlo_method
- [7] **WALD, Ingo, Thiago IZE, Andrew KENSLER, Aaron KNOLL a Steven G. PARKER.** *Ray tracing animated scenes using coherent grid traversal.* ACM Transactions on Graphics [online]. 2006, **25**(3), 485- [cit. 2016-04-12]. DOI: 10.1145/1141911.1141913. ISSN 07300301.
- [8] **MACDONALD, J. David a Kellogg S. BOOTH.** *Heuristics for ray tracing using space subdivision.* The Visual Computer. 1990, 6(3), 153-166. DOI: 10.1007/BF01911006. ISSN 0178-2789.
- [9] **SNYDER, John M. a Alan H. BARR.** *Ray tracing complex models containing surface tessellations.* Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87. New York, New York, USA: ACM Press, 1987, , 119-128. DOI: 10.1145/37401.37417. ISBN 0897912276.
- [10] **WALD, Ingo.** *On fast Construction of SAH-based Bounding Volume Hierarchies.* In: 2007 IEEE Symposium on Interactive Ray Tracing [online]. IEEE, 2007, s. 33-40 [cit. 2016-04-23]. DOI: 10.1109/RT.2007.4342588. ISBN 978-1-4244-1629-5.
- [11] **WALD, Ingo a Vlastimil HAVRAN.** *On building fast kd-Trees for Ray Tracing, and on doing that in $O(N \log N)$.* In: 2006 IEEE Symposium on

- Interactive Ray Tracing [online]. IEEE, 2006, s. 61-69 [cit. 2016-04-23]. DOI: 10.1109/RT.2006.280216. ISBN 1-4244-0693-5.
- [12] **BOULOS, Solomon.** *Notes on efficient ray tracing.* ACM SIGGRAPH 2005 Courses on - SIGGRAPH '05. New York, New York, USA: ACM Press, 2005, , 10-. DOI: 10.1145/1198555.1198749.
- [13] **PHARR, Matt a Greg HUMPHREYS.** *Physically based rendering: from theory to implementation.* 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2010. ISBN 01-237-5079-2.
- [14] **GERBER, Richard.** *Getting Started with OpenMP*.* In: Intel Developer Zone [online]. 2012 [cit. 2016- 03-30]. Dostupné z: <https://software.intel.com/en-us/articles/getting-started-with-openmp>
- [15] **GERBER, Richard.** *More Work-Sharing with OpenMP*.* In: Intel Developer Zone [online]. 2012 [cit. 2016-03-30]. Dostupné z: <https://software.intel.com/en-us/articles/more-work-sharing-with-openmp>
- [16] **WEGHORST, Hank, Gary HOOPER a Donald P. GREENBERG.** *Improved Computational Methods for Ray Tracing.* ACM Transactions on Graphics. 1984, 3(1), 52-69. DOI: 10.1145/357332.357335. ISSN 07300301.

8 SEZNAM PŘÍLOH

Příloha A – diagram tříd.....	74
Příloha B – zdrojový kód, výsledky měření a aplikace na přiloženém CD	

Příloha A – diagram tříd

