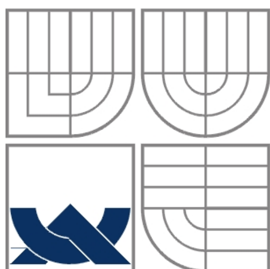
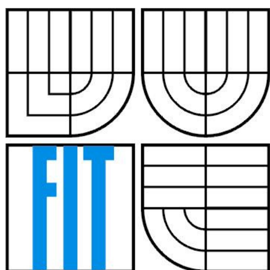


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SLEDOVÁNÍ PAPRSKU POMOCÍ K-D TREE

RAY TRACING USING K-D TREE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE BC. MIROSLAV ŠILHAVÝ
AUTHOR

VEDOUCÍ PRÁCE ING. JIŘÍ HAVEL
SUPERVISOR

BRNO 2010

Abstrakt

Tato práce se zabývá metodami sledování paprsku a jejich akcelerací pomocí k-D stromu. Poskytuje částečný rozbor a přehled algoritmů od klasického střílení paprsku k rekurzivnímu přístupu až k distribuovanému sledování paprsku.

Věnuje se rozboru struktury BSP stromu a dále jeho podtřídy k-D stromu, uvádí základní algoritmus jejich konstrukce i průchodu. Dále se podrobněji zabývá technikami konstrukce k-D stromu, které jsou založeny na správném umístění řezací plochy do buňky stromu. Mezi techniky rozebrané v této práci patří půlení s využitím prostorového mediánu, objektového a poměrně nové techniky cenového modelu SAH, neboli *surface area heuristic.* K závěru práce uvádí výsledky testů a porovnání výkonnosti uvedených metod, ze kterých vychází nejlépe právě SAH.

Abstract

This thesis deals with ray tracing methods and their acceleration. It gives partial study and review of algorithms from classical ray shooting algorithm to recursive approach up to distributed ray tracing algorithm.

Significant part of this thesis is devoted to BSP tree structure and its subclass of k-D tree, it shows simple algorithm for its construction and traversal. The rest of thesis is dealing with k-D tree construction techniques, which are based on the right choice of the splitting plane inside the every cell of k-D tree. The techniques upon the thesis is based on are space median, object median and relatively new cost model technique named SAH, otherwise as *surface area heuristic.* All three techniques are put into testing and performance comparison. In the conclusion the results of tests are reviewed, from where SAH is coming out as a winner.

Klíčová slova

Metody sledování paprsku, BSP strom, k-D strom, konstrukce k-D stromu, průchod k-D stromem, SAH cenový model, objektový a prostorový medián

Keywords

Ray tracing methods, BSP tree, k-D tree, construction of k-D tree, traversing k-D tree, SAH cost model, object a space median

Citace

Miroslav Šilhavý: Sledování paprsku pomocí k-D tree, diplomová práce, Brno, FIT VUT v Brně, 2010

Sledování paprsku pomocí k-D tree

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Jiřího Havla, Ing. A dále, že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Miroslav Šilhavý
26. 5. 2010

Poděkování

Tímto bych rád poděkoval svým rodičům za všeobecnou podporu po celé délce svého studia, volnost pro napsání této práce a následné korektury gramatických chyb.

© Miroslav Šilhavý, 2010

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah ... 3
Kapitola 1 ... 5
Úvod ... 5
1.1 Idea ... 5
1.2 Motivace ... 5
1.3 Přehled obsahu ... 6
Kapitola 2 ... 7
Metoda sledování paprsku ... 7
2.1 Teoretický úvod ... 7
2.1.1 Objekty ve scéně ... 8
2.2 Sledování paprsku ... 9
2.2.1 Algoritmus sledování paprsku ... 9
2.2.2 Přímé osvětlení ... 12
2.2.3 Globální osvětlení ... 13
2.3 Akcelerační techniky ... 15
2.3.1 Redukce primárních a sekundárních paprsků ... 16
2.3.2 Akcelerace výpočtu kolize paprsku se scénou ... 16
Kapitola 3 ... 18
Prostorové strukturování ... 18
3.1 Prostorové dělení scény ... 18
3.1.1 BSP stromy a k-D stromy ... 19
3.2 Konstrukce k-D stromu ... 22
3.2.1 Časová složitost ... 22
3.2.2 Výhody k-D stromu ... 23
3.2.3 Umísťování řezací plochy ... 24
3.2.4 Ukončovací kritéria konstrukce k-D stromu ... 25
3.3 Rozbor konstrukčních metod ... 26
3.3.1 Prostorový medián ... 26
3.3.2 Objektový medián ... 26
3.3.3 Cenový model SAH ... 27
Kapitola 4 ... 32
Implementace k-D stromu ... 32
4.1 Struktura k-D stromu ... 32
4.1.1 Kolize AABB trojúhelník ... 33

4.1.2 Implementace SAH 34
Kapitola 5 36
Testování konstrukčních algoritmů 36
5.1 Testovací scény 36
5.2 Metodika testování 37
5.3 Výsledky měření 38
Závěr 41
Literatura 42
Seznam příloh 44
Příloha 1 45
Příloha 2 48
Příloha 3 50

Kapitola 1

Úvod

V této práci se věnuji rozboru algoritmů pro sledování paprsku a metod pro jejich akceleraci. Zejména pak akcelerační metodě pro dělení prostoru do menších buněk s využitím vícerozměrné struktury k-D stromu.

V další části si tato práce klade za cíl probrat konstrukční algoritmy pro stavbu k-D stromu a otestovat jejich schopnosti v různých scénách při urychlování algoritmu sledování paprsku.

1.1 Idea

Tvorba realistických obrazů zaznamenávajících nebo simulujících výjevy z reálného či smyšleného světa je jednou z prvních dovedností, kterou si člověk snažil od svého počátku osvojit. Ať už to bylo ve formě nástěnných kreseb v jeskyních, malby na malířské plátno, či fotografickým záznamem nebo formou počítačové tvorby.

Vždy se pod záměry autorů všech takto vytvořených obrazů ukrývala podobná idea. A tou je lidská podstata, která způsobuje touhu tvořit a objevovat. A to platí i v takových případech jako je vývoj nových grafických algoritmů a technologií. I v tom samotném zkoumání a vývoji, je ta skrytá touha něčeho dosáhnout, předat své znalosti nebo být pamatován.

To vše nakonec způsobuje, že lidé chtějí malovat, fotit, modelovat, vyvíjet nebo objevovat nové věci a někteří tak nakonec i činí.

1.2 Motivace

Počítačová grafika během posledních pár desítek let zaznamenala obrovský technologický skok kupředu. Její hnací silou je zejména filmový a herní průmysl, ale nemalou roli hraje i oblast aplikací pro 3D vizualizaci v celé řadě odvětví (zdravotnictví, průmyslový design a další). Z hlediska reálnosti a rychlosti syntetizovaných obrazů by se dal obor počítačové grafiky rozdělit na dva typy, prvním typem by byla zřejmě oblast rychlé, interaktivní grafiky, občas s výstupy až téměř přibližujících se fotorealistické kvalitě, zatímco tím druhým typem by byla oblast tvorby grafiky s výstupy fotorealistické kvality.

K prvnímu typu se řadí počítačové aplikace využívající grafické knihovny OpenGL a DirectX. Největší výhodou této kategorie je její rychlost syntézy obrazů, která je založena na využití speciálních hardwarových prostředků od firem AMD ATI a nVidia. Proces výpočtu výstupního obrazu, který dosahuje této rychlé výpočetní rychlosti, avšak není kompletně založen na přesné fyzikální simulaci reálného světa, ale částečně na jeho aproximaci.

Odvětví této rychlé interaktivní grafiky je využíváno hlavně v herním a zábavním průmyslu, ale v poslední dekádě se jeho využití dostává i k dalším technickým oborům, jako je např. lékařství a oblast průmyslového designu.

Druhý typ pro zobrazování počítačové grafiky, fyzikálně velmi realistické, je odlišný od prvního v tom, že nevyužívá žádných speciálních hardwarových prostředků a metoda vykreslování je založena čistě na fyzikální simulaci reálného světa. Aplikace implementující tuto myšlenku jsou ve většině případů založeny na metodě sledování paprsku a pro dosáhnutí ještě vyšší kvality bývají rozšířeny o algoritmy globálního osvětlení, mezi ty známější algoritmy patří např. metoda Monte Carlo.

Obecně metody pro fotorealistické zobrazení jsou známy svou časovou náročností, to je způsobeno opakováním se algoritmům nutných pro výpočet viditelnosti. A téměř žádnou technologickou podporou ze strany hardwaru, to je zapříčiněno tím, že oblast fotorealistického vykreslování je oproti interaktivnímu vykreslování grafiky použitím GPU pouze na začátku širšího nasazení do oblasti praktického použití. Menší pokrok lze už nyní pozorovat ze strany společnosti NVidia a její technologie CUDA, Intelu a obecně v nové architektuře grafických karet.

Aplikace pro fotorealistické vykreslování jsou základem tvorby efektů v oblastech filmového průmyslu, animací a 3D vizualizací.

1.3 Přehled obsahu

Následující část práce je strukturně rozdělena do čtyř částí, tj. úvod do metody sledování paprsku. Dále část rozebírající a popisující algoritmy prostorového dělení s větším zaměřením na rozbor struktury k-D stromu. Další kapitola se zaobírá konkrétními algoritmy konstrukce k-D stromu, kde je část věnována rozboru složitější metody *SAH*. Tato metoda je dále spolu se samotným návrhem implementována a popsána v kapitole implementace. V poslední kapitole jsou všechny implementované metody podrobeny různým testům. Následuje závěr společně se shrnutím a návrhem možnosti budoucího pokračování.

Kapitola 2

Metoda sledování paprsku

Pojem sledování paprsku neboli raytracing se často užívá v mnoha různých významech, od sebe docela odlišných, i když podle definice správných. Pod pojmem sledování paprsku si tedy lze představit nejen ten nejzákladnější princip pro hledání průsečíků paprsku s množinou objektů, ale i klasický algoritmus s rekurzivním přístupem sledování paprsku.

Nebo i v takové formě, že když se řekne sledování paprsku, tak si pod tímto pojmem lze představit i většinu metod či algoritmů globálního osvětlení, protože v základu taktéž používají pro vykreslení obrazových výstupů právě metodu sledování paprsku.

A proto je zřejmé brát v úvahu, že tato práce se zabývá algoritmem sledování paprsku popsaným dále.

2.1 Teoretický úvod

Základním poměrně jednoduchým principem každé implementace algoritmu metody sledování paprsku je vystřelování orientovaných paprsků. A to z určitého bodu, oka pozorovatele, dále do scény, která obsahuje množinu geometrických objektů.

Jeden paprsek je vlastně polopřímka definovaná svým počátkem $O = (x, y, z)$ a směrovým vektorem $\vec{U} = (x, y, z)$. V algoritmu je vyžadována forma v parametrickém zápisu přímky, a to proto, že parametr *t* v rovnici je důležitý pro další výpočty bodů $P(t)$ ležících na paprsku. Takže výsledná parametrická forma zápisu paprsku by vypadala takto:

$$P(t) = O + t \cdot \vec{U}, \quad t \in R, t \geq 0 \qquad \text{rovnice 2.1}$$

Je možné definovat maximální hodnotu t_{max}, která specifikuje maximální vzdálenost, do které paprsek hledá průsečíky s objekty. Potom se berou v úvahu pouze průsečíky ve vzdálenosti $0 \leq t_{hit} \leq t_{max}$. Toto omezení vzdálenosti není většinou nutno definovat, používá se pro určité typy implementací metody detekce průsečíku.

Avšak ve sledování paprsku vystává celkem častý problém, a to je vlastní okluze. Jelikož v metodě sledování paprsku často mají sekundární paprsky svůj počátek na objektech, tak při následném výpočtu nejbližšího průsečíku u tohoto nového paprsku může nastat numerická chyba, která může způsobit vlastní okluzi objektu. Pro předejití tohoto častého problému se používá posunutý rozsah pro detekci průsečíků $\epsilon \leq t_{hit} \leq t_{max}$, kde ϵ bývá zpravidla $0 < \epsilon \ll 1$.

Při definici metody sledování paprsku bylo zřejmé, že základním problémem je detekce průsečíku. Nicméně pro správné vykreslení scény je třeba získat nejbližší průsečík paprsku s nějakým objektem ve scéně. Algoritmus vystřelování paprsků počítá právě tyto požadované hodnoty t_{hit}.

Základní verze tohoto algoritmu bývá označována jako naivní. To je zřejmé z toho, že tento algoritmus hledání průsečíku prochází úplně všechny objekty ve scéně pro každý vyslaný paprsek. Z toho plyne, že nárůstem počtu objektů ve scéně lineárně stoupá náročnost tohoto algoritmu, proto se

tento typ algoritmu používá pouze při vykreslování triviálních scén, v případě komplexnější scény je tento algoritmus příliš náročný na výpočet.

Častěji používaná metoda pro výpočet nejbližšího průsečíku počítá s tím, že před samotným algoritmem vystřelování paprsku je scéna s objekty předzpracována do určité datové struktury. A až po vytvoření je teprve tato struktura použita dále při vyhledávání nejbližšího průsečíku. Časová složitost je v porovnání s naivním přístupem často mnohonásobně menší, proto je tento typ algoritmu ideální pro komplexní scény, nicméně pro dosáhnutí většího zrychlení velice záleží na použitém algoritmu pro zpracování scény.

2.1.1 Objekty ve scéně

Popis scény je důležitou součástí každé aplikace implementující algoritmus sledování paprsku, bez ní by tato aplikace nebyla úplná, protože bez adekvátně vytvořeného modelu vykreslovaného prostředí uživatel nedokáže zhodnotit kvalitu použitého algoritmu a případně jeho nadstavby v podobě rozšíření např. o algoritmy globálního osvětlení.

Každá scéna se obvykle skládá z geometrických objektů, které je možné matematicky vyjádřit pomocí funkce pro prostorový souřadný systém nebo pomocí množiny bodů je definujících. V každém případě, pro zadaný geometrický útvar, musí jít spočítat průsečík paprsku s takovým objektem. Avšak je poměrně časté, že pro složitější objekty nelze spočítat průsečík přímo a přesně, a je nutné použít nějakou numerickou metodu pro nalezení řešení. Např. rovnice popisující povrch koule či plochy patří k objektům, pro které je nalezení průsečíku s paprskem analytickou metodou nejrychlejší a přesné, naopak pro popis superelipsoidu je tento problém nalezení průsečíku analyticky neřešitelný, proto je třeba použít řešení numerickou metodu. Poté je nutné brát v úvahu rychlost takových výpočetních metod a zhodnotit zda se nevyplatí namodelovat objekt pomocí sítě trojúhelníků.

Zde vystává další otázka. Zda do popisu scény má význam zahrnout pouze objekty tvořené trojúhelníkovou sítí nebo se zaměřit na více typů objektů. Je to zřejmě důležitá otázka, jelikož většina existujících programů založených na přímém vykreslování (rasterizaci) pracuje pouze se scénami skládající se z trojúhelníků. Ačkoliv pravdou je, že právě jednou z hlavních výhod metody sledování paprsku je ta šířka podporovaných typů. Je praxí např. ve filmovém průmyslu, že scény jsou tvořeny pouze trojúhelníkovou sítí, jelikož se zřídkakdy dají využít takové objekty, jako jsou „dokonalé“ koule apod., protože v reálném světě se dokonalé tvary objektů téměř vůbec nevyskytují.

Scéna podporovaná metodou sledování paprsku tedy může obsahovat objekty, jako jsou koule, plochy, kvadratické plochy, superelipsoidy, válce, trojúhelníky, implicitní plochy, Bézierovy plochy nebo izoplochy. Přehled dalších objektů, které lze vykreslit metodou sledováním paprsku lze vidět v literatuře na [6]. Algoritmy pro výpočet průsečíků paprsku s různými typy objektů na [1].

2.2 Sledování paprsku

Metoda sledování paprsku je tedy založena na zmíněném principu vystřelování orientovaných paprsků z určitého bodu do scény. Historicky tato metoda prodělala poměrně dosti zásadních změn, která každou novou revizí umožňovala dosáhnout realističtějšího obrazového výstupu. V této části budou popsány dřívější i nejaktuálnější přístupy k metodě pro vykreslování fotorealistických snímků.

2.2.1 Algoritmus sledování paprsku

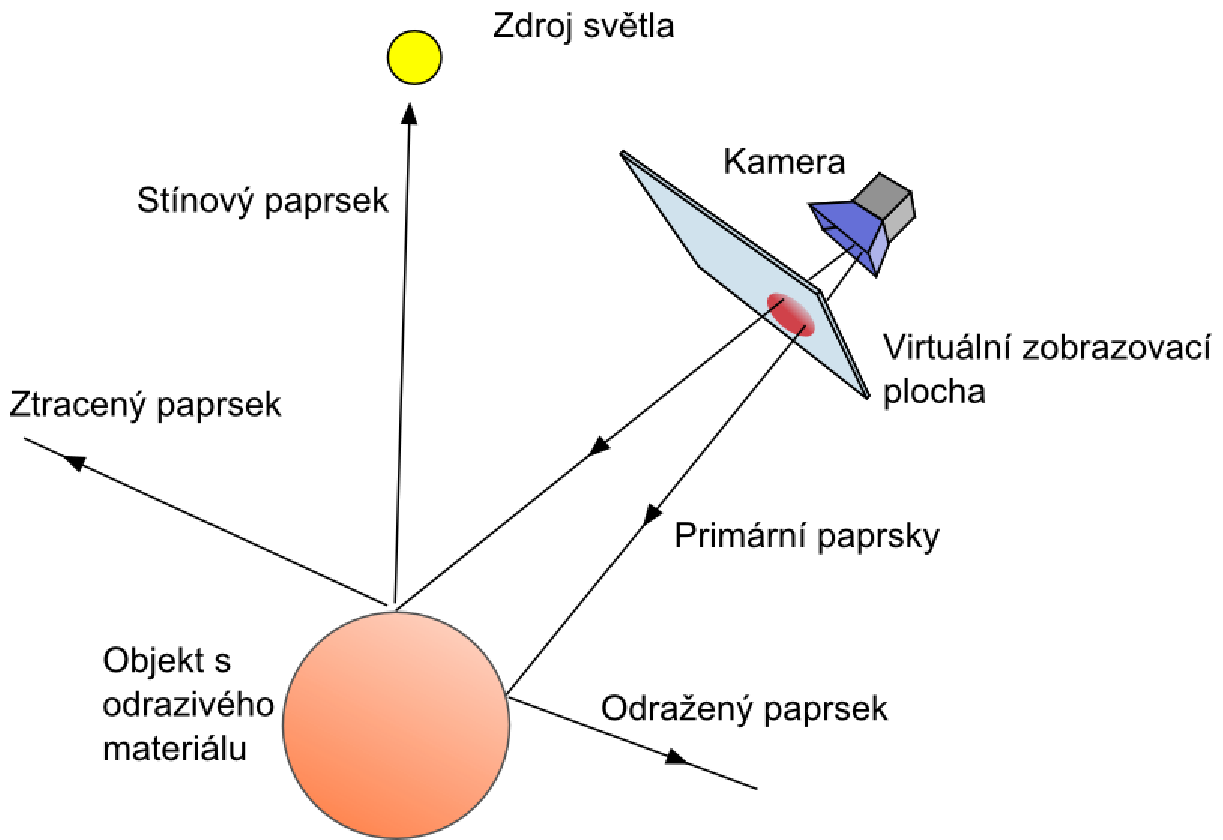
Pro představu, jak funguje algoritmus, je třeba si představit virtuální kameru či oko ve třídimenzionální prostoře scény. Z této virtuální kamery, využitím metody vystřelování paprsků, vyšleme paprsky skrz dvojdimenzionální plochu představující obrazovou plochu, na kterou se promítne výsledný obraz. Tato plocha je definována jako dvojrozměrné pole pixelů. Paprsky letící skrz každý pixel této plochy se střetávají s objekty ve scéně, kde se v tento moment nejvíce uplatňují algoritmy pro hledání nejbližšího objektu a jeho průsečíku (tzn. buď takzvaným naivním, nebo přístupem s použitím určitého předzpracování scény), tyto paprsky poté nabírají barevnou informaci z materiálu zasaženého objektu. Výsledný obraz potom vzniká uložením všech těchto barevných informací z jednotlivých paprsků do každého pixelu obrazové plochy.

Takto vypadala první myšlenka a princip algoritmu metody sledování paprsku představená Arturem Appelem [2] v roce 1968.

2.2.1.1 Rekurzivní sledování paprsku

Aktuálně používaná metoda představená Turnerem Whittedem [24] v roce 1980 vychází z Appelova přístupu. Významnou úpravou původního algoritmu ho rozvíjí o další procesy. Původní algoritmus předpokládal pouze tzv. primární paprsky, letící z oka pozorovatele, ale nový přístup ho rozšířil o nové typy paprsků, o tzv. sekundární paprsky. Interakcí primárního paprsku s objektem mohou vzniknout až tři nové typy paprsků, jsou to paprsky vzniklé odrazem od lesklého povrchu objektu, lomem do průhledného a tzv. stínové paprsky. Myšlenka tvorby stínových „paprsků“ byla již také navržena i Appelovým přístupem, nicméně v dosti odlišné podobě, takže jsem ji jako součást sledování paprsku předtím nezmínil. Každý povrch objektu je tedy oproti Appelovu návrhu rozšířen o další vlastnosti, je to odrazivost povrchu, lámavost a index lomu. Dále se u každého průsečíku paprsku s objektem zjišťuje, zdali neleží ve stínu, tzn., že se vystřelí paprsek směrem ke každému zdroji světla a pokud neexistuje žádný objekt, který by ležel mezi původním průsečíkem a zdrojem světla, tak se spočítá barevná informace paprsku z intenzity dopadajícího světla a barvy objektu.

Whittedův přístup zavedl rekurzivní podobu metody sledování paprsku, jelikož pro sekundární paprsky se uplatňují stejné procesy jako pro primární. Ale protože počet takových sekundárních paprsků může být teoreticky nekonečně mnoho, tak se pro zajištění konečnosti výpočtu tohoto algoritmu definuje parametr omezující hloubku rekurze, který omezuje generování dalších sekundárních paprsků. Výsledný obraz poté vznikne skládáním barevných informací všech vygenerovaných paprsků. Obrázek 2.1 ilustruje rekurzivní podobu metody sledování paprsku.



Obrázek 2.1: Rekurzivní sledování paprsku

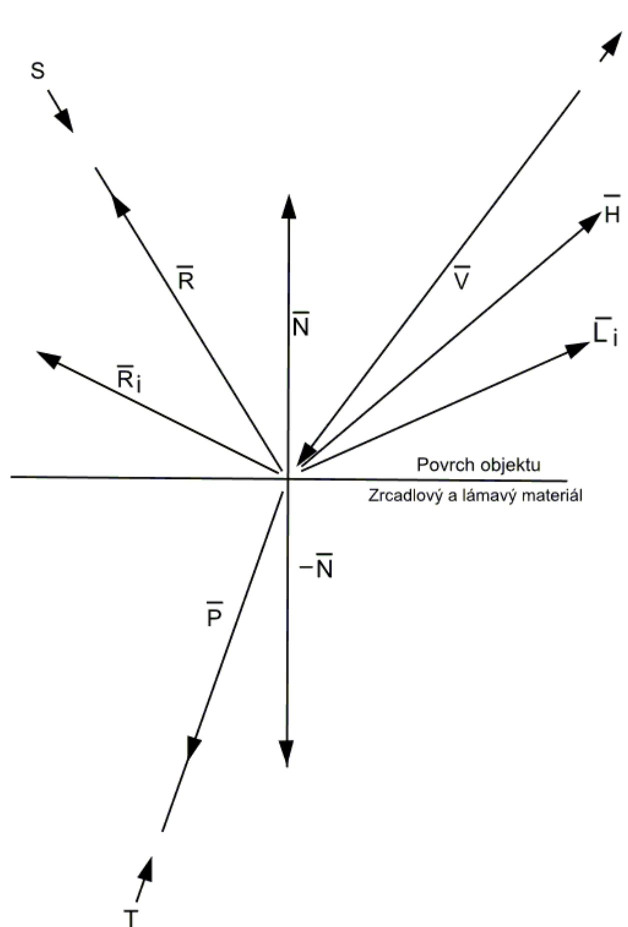
Návrh dále kromě počítání barevných intenzit sekundárních paprsků uvažoval i modely k výpočtu osvětlení daného bodu. V tomto případě se nejčastěji používá Phongův výpočetní model intenzity světla, prezentovaný Bui-Tuong Phongem v roce 1975 [7] (rovnice 2.2), nebo jeho upravená podoba označována jako Blinn-Phong model, což je modifikovaná verze Phongova modelu výpočtu intenzity prezentovaná Blinnem a Newellem v roce 1977 [5] (rovnice 2.3), který použil i Whitted jako základ algoritmu výpočtu osvětlení ve své metodě sledování paprsku.

$$I = I_a + k_d \sum_{i=1}^{i=ls} (\vec{N} \cdot \vec{L_i}) + k_s \sum_{i=1}^{i=ls} (\vec{N} \cdot \vec{R_i})^n \qquad \textit{rovnice 2.2}$$

$$I = I_a + k_d \sum_{i=1}^{i=ls} (\vec{N} \cdot \vec{L_i}) + k_s \sum_{i=1}^{i=ls} (\vec{N} \cdot \vec{H})^n \qquad \textit{rovnice 2.3}$$

V rovnici 2.3 lze vidět, že Whitted stále ještě nebral v úvahu intenzitu světla vytvořenou sekundárními paprsky. Toho dosáhl úpravou rovnice 2.3 do podoby rovnice 2.4, do které přidal parametr S, označující intenzitu světla dopadajícího ve směru vektoru R a parametr T, který označuje intenzitu ze směru vektoru P a k_t, což je koeficient určující poměr přínosu intenzity světla procházejícího skrz průhledný objekt. Další významné veličiny jsou zobrazeny na obrázku 2.2, včetně jejich významu.

$$I = I_a + k_d \sum_{i=1}^{i=ls} (\vec{N} \cdot \vec{L_i}) + k_s S + k_t T \qquad \textit{rovnice 2.4}$$



Obrázek 2.2

Význam veličin z obrázku 2.2.

I – odražená intenzita světla dopadající do oka pozorovatele
I_a – intenzita způsobená všesměrovým odrazem světla (ambientní osvětlení)
k_d – koeficient difusní
k_S – koeficient spekulárního odrazu světla
k_T – koeficient přenosu světla skrz průhledný objekt
$\vec{N}$ – normálový vektor objektu v místě průsečíku
$\vec{L_\iota}$ – vektor směrující k *i-tému* zdroji světla
$\vec{R_\iota}$ – odraz vektoru $\vec{L_\iota}$ směrující od *i-té* zdroje světla
$\vec{H}$ – vektor v polovině úhlu mezi pozorovatelem a zdrojem světla
n – exponent definující lesk objektu
S – intenzita světla přicházejícího ze směru vektoru $\vec{R}$
T – intenzita světla přicházejícího ze směru vektoru $\vec{P}$, sekundární paprsek

Kde *S,* se dá rozložit podle rovnice 2.1 na:

$$S = \sum_{j=1}^{j=ls} (\vec{N} \cdot \vec{H})^n + S' \qquad \textit{rovnice 2.5}$$

S' – intenzita světla z odraženého paprsku od pozorovatele, má význam pouze při použití sekundárních paprsků

2.2.1.2 Distribuované sledování paprsku

V předchozí části byl popsán algoritmus, který sice dokázal pracovat s jakkoliv složitou scénou, ale i přesto poskytoval pouze částečný náhled na skutečnou simulaci fyzikálních a zejména optických jevů. Mezi techniky, které algoritmus zavedl a dokázal je slušně simulovat, patřily bodová světla, dokonalé odrazy, lom světla, sekundární paprsky a tedy i možnost další rekurze. Bodová světla omezovala algoritmus k vytváření ostrých stínů. Je zřejmé, že vykreslená scéna použitím těchto technik nemůže skutečně napodobit reálnou scénu. Proto jako další etapa vývoje vykreslování pomocí sledování paprsku byla vytvořena metoda distribuovaného sledování paprsku.

Tato metoda odstraňuje zavedené nedostatky simulací dalších efektů pravděpodobnostní distribucí paprsků, čili náhodným (stochastickým) vzorkováním (Cook, 1984 [8]). Metoda přináší možnost simulace efektů, jako jsou měkké stíny (přidáním světel s větší plochou), difusní odrazivost, hloubka ostrosti, rozmazání pohybem, rozostřené odrazy, lom světla a jeho rozklad.

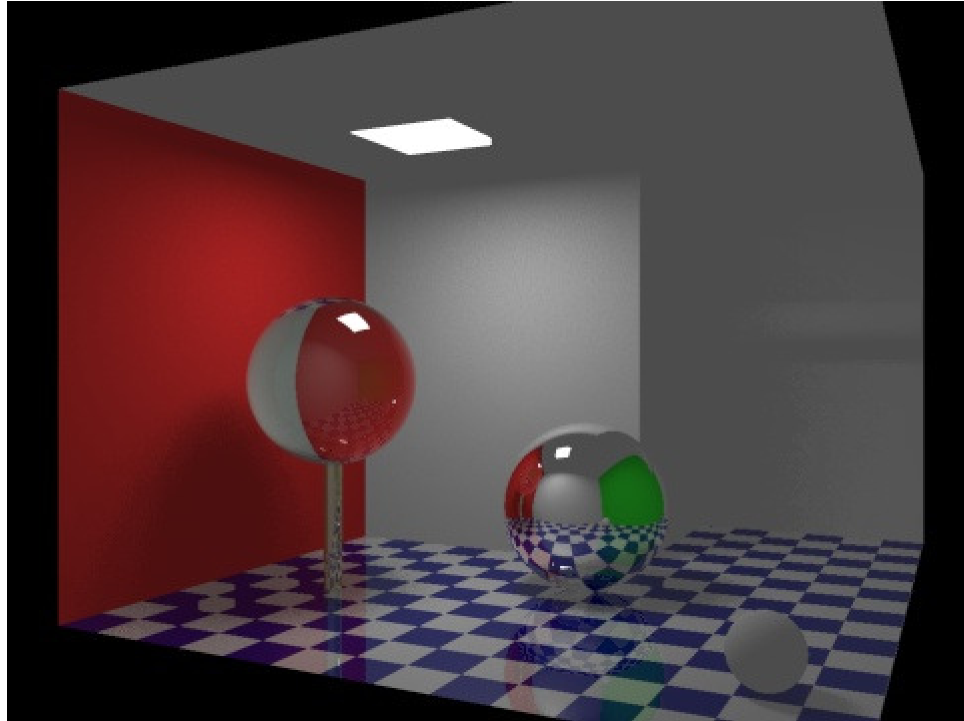
Například měkké stíny jsou simulovány stochastickým vzorkováním pravděpodobnostní distribuce stínových paprsků vyslaných směrem ke zdroji světla. Přesněji to probíhá tak, že plocha světla je rozdělena do pravidelné mřížky (typicky stačí 8x8) a v oblasti každé buňky mřížky je náhodně vybrán bod, který vytvoří s průsečíkem objektu jeden stínový paprsek. Takto vytvořené stínové paprsky se testují, zdali nejsou překryty jiným objektem. Výsledná barva daného bodu je potom vypočtena s ohledem na počet nezakrytých paprsků k počtu překrytých. Na stejném principu jsou založeny i další zmíněné nově zavedené techniky metody distribuovaného sledovaní paprsku. Touto metodou vytvořené scény dosahují mnohem reálnějších výstupů, než tomu bylo u klasické rekurzivní metody.

Na obrázku 2.3 je vidět příkladná scéna vytvořená Whittedovým sledováním paprsku rozšířeným o efekty distribuovaného sledování paprsku.

Pozn.: Tato metoda je ve skutečnosti aplikací metody *Monte Carlo*, i když ne v takové hloubce využití jako je tomu v algoritmech globálního osvětlení, a proto se tato metoda často označuje jako stochastické sledování paprsku.

2.2.2 Přímé osvětlení

Jako algoritmy přímého osvětlení, neboli *direct illumination*, se označují metody rekurzivního a distribuovaného sledování paprsku. Teoreticky by se daly i tyto algoritmy považovat jako formy globálního osvětlení, protože sekundární paprsky jsou příkladem druhotného osvětlení objektů. Avšak jako algoritmy globálního osvětlení se označují pouze ty splňující body popsané v 2.2.3, protože zavedené sekundární paprsky nedokážou simulovat jisté formy chování difusních materiálů a další optické jevy, jako je kaustika.



Obrázek 2.3: Ilustrační scéna vykreslená pomocí metody distribuovaného sledování paprsku
Převzato z [23]

2.2.3 Globální osvětlení

I když se zdálo, že problém realistického zobrazení scény byl vyřešen, tak i přesto zde byly jisté nedostatky, které nedokázal algoritmus rekurzivního sledování paprsku či distribuovaného patřičně nasimulovat.

Hlavní neschopnost algoritmu vychází z nemožnosti vykreslit scénu pod nepřímým světlem, protože ve skutečnosti je materiál každého objektu odrazivý, ačkoliv ne dokonale, tak jako je u zrcadla, ale tzv. difusně odrazivý (část barevného spektra světla materiál absorbuje a zbytek odráží do všech směrů). Phongův osvětlovací model zavedl parametr I_a (prostorové osvětlení), který tento jev částečně simuloval, ale to pouze formou konstantního nasvětlení scény.

Posledním nedostatkem, který by měl algoritmus globálního je simulovat lom či rozklad světla skrz průhledný objekt. Což metoda sledování paprsku z oka pozorovatele sice částečně podporuje, ale ne tak, aby ji pozorovatel viděl z opačného hlediska. Příkladem budiž algoritmus takový, který vystřeluje paprsky ze světla na průhledný objekt např. hranol a způsobí rozklad světla na barevné spektrum. Bohužel algoritmus, který by dokázal simulovat tento jev tak realisticky, jak se s ním setkáváme, neexistuje. Ale jak se ukáže dále, tak jisté metody formou aproximace tyto optické vlastnosti světla dokážou dostatečně simulovat, pro oko nerozeznatelně od reálných.

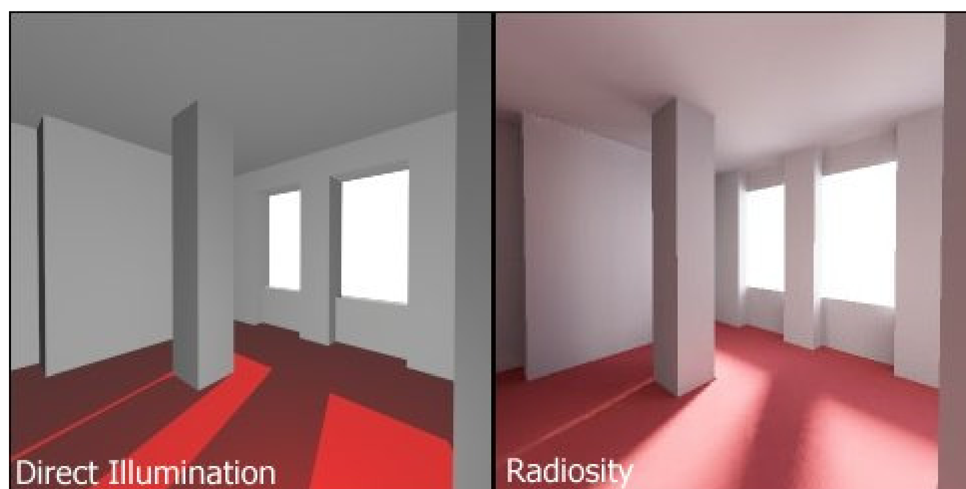
2.2.3.1 Radiosita

Metoda výpočtu globálního osvětlení, která se obejde bez vykreslovací metody sledování paprsku, nicméně se s ní dá zkombinovat. Pro výpočet osvětlení ve scéně používá formu vystřelování světelných paprsků z každého objektu, který je rozsekán na určitý počet stejně velikých částí, tzv. *patchů.* Každou iterací algoritmu jsou otestovány všechny *patche* všech objektů ve scéně, jestli paprsky z nich vyslané nenarazí na zdroj světla, pokud ano, tak se stanou zdrojem světla pro další

testované *patche* v další iteraci. Proces se opakuje stále dokola, takže určité *patche* později potom přijímají více světla z ostatních *patchů* a tím se zároveň stávají silnějším zdrojem. Proces je ukončen obdobně jako u sledování paprsku, tak, že po dosáhnutí určité nastavené hranice se iterace ukončí.

V základu je tohle princip funkce *radiosity*, problémy skryté pod touto metodou jsou obdobné jako u sledování paprsku, scéna taktéž bývá před samotným algoritmem zpracována do nějaké prostorové struktury, řeší se hledání průsečíků s paprsky apod. Tento algoritmus řeší problém difusního rozptylu světla nejreálněji v porovnání s metodami *Monte Carlo*.

Ilustrační obrázek 2.4 porovnává tuto metodu s metodou rekurzivního sledováním paprsku, v obou scénách je zdrojem světla koule za oknem. Bílé pozadí za okny je pouze textura, nemá vliv na výpočet osvětlení. Scéna vykreslená pomocí rekurzivního sledování paprsku byla viditelná zejména díky parametru I_a Phongova modelu, bez ní by scéna byla téměř kompletně černá, nicméně pro zvýraznění detailů objektů byl dovnitř scény vložen ještě bodový zdroj světla. Difusní prostorové osvětlení ve scéně vykreslené pomocí *radiosity* byla iterována šestnácti průchody.



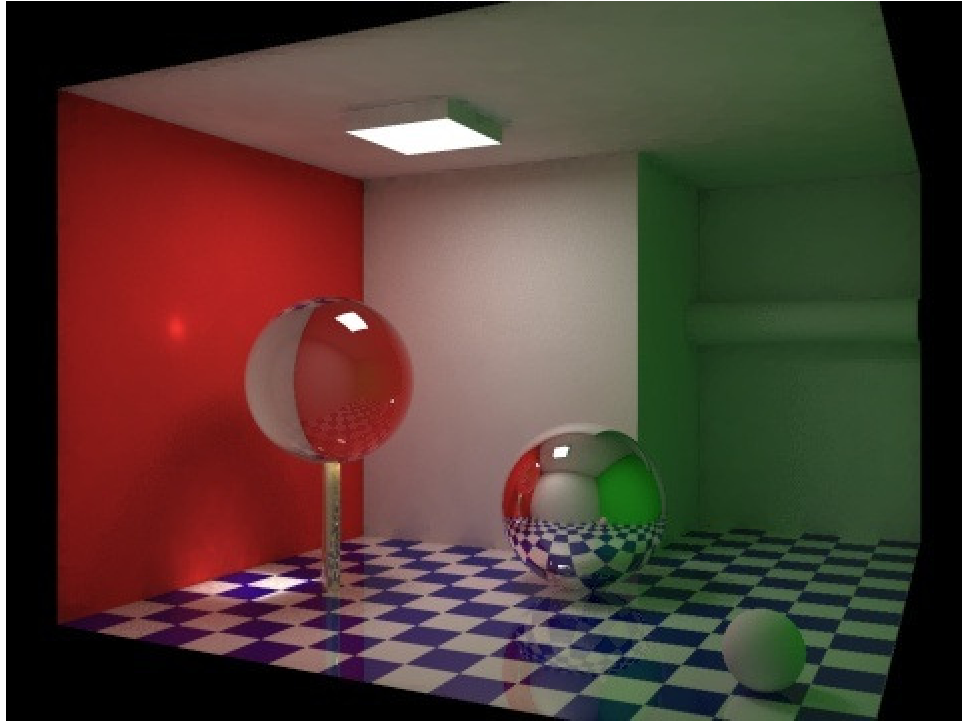
Obrázek 2.4: Rekurzivní sledování paprsku v porovnání s radiositou. Obrázek převzat z [10].

2.2.3.2 Metoda Monte Carlo sledování paprsku

V poslední době patří k nejrozšířenějším způsobům pro simulaci efektů globálního osvětlení. Algoritmů založených na stochastické technice Monte Carlo existuje vícero [14], mezi výhody patří lehká implementace, mnohem menší náročnost v porovnání s radiositou a výstupní kvalita. Nicméně podobně jako tomu bylo u metody distribuovaného sledování paprsku, je nutné nastavovat počet vzorků, které mají přímý dopad na výstupní kvalitu globálního osvětlení. Pokud není počet dostačující, tak se ve scéně zobrazuje náhodný šum. Na obrázku 2.5 lze vidět, jaké techniky přineslo Monte Carlo pro porovnání s obrázkem 2.4.

Tyto metody řeší tzv. vykreslovací rovnici, kterou prezentoval James Kajiya v [15]. Mezi algoritmy patřící do skupiny globálního osvětlení použitím metody Monte Carlo sledování paprsku patří:

- *Photon mapping*
- *Metropolis light transport*
- *Bidirectional path tracing*
- *Path Tracing*
- *Irradiance caching*



Obrázek 2.5: Ilustrační scéna vykreslená pomocí metody distribuovaného sledování paprsku a algoritmu globálního osvětlení. Převzato z [23]

2.3 Akcelerační techniky

Dosud byly pouze zmiňovány různé algoritmy a metody pro realistické vykreslování scény. Všechny jsou známé svými efektními obrazovými výstupy a poměrně jednoduchými vykreslovacími algoritmy. Ať už se jedná o metody sledování paprsku, *radiositu*, metody globálního osvětlení nebo *rasterizaci*, všechny musí čelit určitým problémům s výkonností. Ty zaměřené na využití algoritmu sledování paprsku často musí pro vykreslení pouze jednoho snímku vystřelit miliony paprsků pro dosáhnutí kvalitních výstupů, což ve výsledku zabírá z celkového času běhu programu tu nevětší část. Ale toho si už při prezentování metody rekurzivního sledování paprsku všiml i Whitted [24], u jednoduché testovací scény tvořil čas pro počítání průsečíků 75% z celku, a u komplexní scény až 95%. Proto už od uvedení metody sledování paprsku probíhal nejen další vývoj ve zkvalitnění metod pro vyšší realističnost scény, ale hlavně ve vývoji jejich urychlení.

Metod pro akceleraci počítání průsečíku paprsku existuje několik typů. Může se jednat o různé algoritmy pro výpočet průsečíků paprsku s objektem. Nebo metod pro výpočet kolizí obalových těles s objekty, které jsou součástí jiných akceleračních technik.

Dále existují metody, které snižují počet sekundárních a primárních paprsků, mezi které patří stínová paměť, optimalizace prvního zásahu, stínové, reflexní mapy nebo jedno-vzorkové měkké stíny.

A posledně metody optimalizující a snižující počet potřebných výpočtů průsečíků s paprskem na samotné minimum. Jak už jsem zmínil na začátku práce, tak je třeba tzv. naivní přístup, co nejvíce optimalizovat, a na to se nejčastěji používají metody předzpracování scény do některého z typů prostorových struktur. Rozbor technik zaměřených na toto téma pokračuje kapitolou č.3.

2.3.1 Redukce primárních a sekundárních paprsků

Zrychlení výpočtu scény lze dosáhnout různými způsoby zmíněnými v bodě 2.3, v této části podrobněji popíšu některé z metod zaměřených na snižování počtu paprsků a jejich optimalizaci.

2.3.1.1 Stínová paměť

Stínové paprsky patří mezi nejčastěji vystřelované sekundární paprsky, ať už jsou vytvořené primárními či sekundárními paprsky. Pro stínové paprsky stačí nalézt jakýkoliv objekt mezi světlem a průsečíkem. Princip této metody spočívá v uložení si objektu, který způsobil poslední zastínění daného zdroje světla. To se poté využije při testování zastínění, tak, že se první otestuje zapamatovaný objekt a poté, pokud testem projde, se otestuje i zbytek objektů. Tato metoda je dobře využitelná pro velmi jednoduché scény, nicméně pro složitější přináší spíše výpočty navíc.

Stínová paměť je aplikovatelná pouze pro zastíněné objekty, a v případě nezastíněných způsobují pouze další zatížení. Navíc tato metoda ztrácí využití v případech použití algoritmů globálního osvětlení, plošných světel nebo ve složitějších scénách, kde pravděpodobnost zásahu stínového paprsku malým trojúhelníkem je už poměrně malá.

2.3.1.2 Optimalizace prvního zásahu

Tato metoda využívá *rasterizace* pro urychlení identifikace zasaženého objektu primárním paprskem. Metoda funguje takovým způsobem, že se celá scéna nejdříve *vyrasterizuje*, a to využitím *rasterizačního* hardwaru pro dosáhnutí vyšší rychlosti. Vykreslená scéna z *rasterizéru* potom vypadá tak, že každý objekt je vykreslen unikátní barvou. To umožňuje při vystřelení primárních paprsků na určitý pixel okamžitě identifikovat zasažený objekt podle barvy uložené ve *frame bufferu*.

Navíc tato metoda je použitelná pouze pro primární paprsky, a proto nepřináší přílišný výkon navíc, jelikož v komplexnějších scénách s více světelnými zdroji a sekundárními efekty tvoří primární paprsky pouze zlomek všech paprsků. V případě už velmi komplexních scén s velkým počtem objektů se vyplatí použít klasické sledování paprsku.

2.3.2 Akcelerace výpočtu kolize paprsku se scénou

Jak si všiml Whitted, tak čas potřebný pro výpočet kolizí paprsků s objekty tvoří až 95% výpočetní času algoritmu. Proto má tato část akcelerace největší potenciál přinést opravdové zrychlení a bez jakýchkoliv vedlejších dopadů.

2.3.2.1 Varianty pro výpočet průsečíků s objekty

Je evidentně zřejmé, že průsečík paprsku s jakýmkoliv objektem může být spočítán různými způsoby. Každý z těchto různých algoritmů se může lišit v rozdílném počtu operací s desetinnými čísly, celými čísly, počtem podmínek, spotřebou paměti a numerickou přesností. Různé aplikace upřednostňují tyto odlišné metody, protože některé ze součástí těchto algoritmů jsou jinak rychlé na různých hardwarových konfiguracích.

Množství různých algoritmů je známo pro většinu typu objektů. Pro samotné trojúhelníky existuje několik způsobů výpočtu, včetně jejich dalších upravených variant. Pro obecnější objekty jsou metody výpočtu průsečíků jednodušší, efektivní a obecně docela známé, takže nejsou tyto metody často vylepšovány a tedy i publikovány. Zatímco pro složitější typy objektů platí, že se je vyplatí spíše namodelovat s pomocí trojúhelníků a poté je vykreslit. Proto se vývoj zaměřil zejména

na metody výpočtu kolize paprsek-trojúhelník. V samotné praktické části této práce jsem použil metodu uvedenou v [12] a [21].

2.3.2.2 Snižování celkového počtu kolizních výpočtů

Mezi další metody urychlení patří metody, které snižují celkový počet výpočtů pro hledání nejbližšího průsečíku. Tyto metody vychází z předpokladu, že ve scéně složené z více překrývajících se objektů není nutné testovat všechny objekty ležící v dráze vystřeleného paprsku, což zabírá v komplexnějších scénách to největší procento celkového výpočetního času, ale pouze ty objekty, které leží nejblíže počátku paprsku.

Mezi jednou z metod, která svou strukturou není příliš složitá a je zároveň použitelná i pro zbylé metody, patří obalová tělesa. Tuto metodu zde zmiňuji schválně, protože se v častých případech kombinuje právě se strukturou k-D stromů. Jako obalová tělesa se často používají například obalové boxy AB nebo koule.

Na rozdíl od metod rozebraných v další v kapitole tato metoda nemívá konstrukční část automatizovánu. To proto, že nepřináší takové urychlení jako další metody a hlavně se těžko aplikuje na komplexní scény. Proto její využití závisí především na požadavcích na samotný výsledný program založený na sledování paprsku.

Takovými požadavky mohou být například schopnosti vykreslit objekty definované parametrickými rovnicemi, mezi které patří superelipsoidy, kvadriky a jiné typy (více na [6]). Pro tyto objekty bývá typické, že jsou buď nekonečné, nebo pro ně neexistují jednoduché způsoby pro nalezení průsečíku. V tomto případě se uplatňují obalová tělesa, která omezí velikost objektů, respektive možní při výpočtu průsečíku numerickými metodami určit rozsah kořenů.

Zbylé metody a zejména k-D stromy jsou rozebrány dále.

Kapitola 3

Prostorové strukturování

Nejúspěšnější způsoby urychlení sledování paprsku probíhají právě touto formou, a to snížením celkového počtu operací pro výpočet průsečíku. Obvykle tyto metody pracují způsobem postavení speciální indexované datové struktury, která umožňuje co nejrychlejší nalezení objektů, které leží v dráze paprsku nejblíže a přeskočení objektů, které jsou daleko. Během letu jsou tedy testovány pouze ty objekty, které patří mezi potenciální kandidáty na zásah, a právě tímto jo dosaženo snížení celkového počtu výpočtů průsečíků a zároveň obrovského urychlení vykreslení jednoduchých i komplexních scén.

Za posledních pár let bylo vyvinuto mnoho různých typů těchto akceleračních struktur. V principu se tyto struktury dělí na to, jakým způsobem organizují objekty ve scéně. Buď rozdělují prostor scény na menší pod-prostory (*voxely*), nebo uspořádají scénu do nějaké hierarchie.

Mezi základní datové struktury použitelné pro metodu sledování paprsku patří tyto:

- Prostorové datové struktury
  - Obalová tělesa
  - Hierarchická obalová tělesa
  - Prostorové dělení
    - binární prostorové dělení (BSP) stromy
    - k-D stromy
    - octree
    - rovnoměrná mřížka
    - nerovnoměrná mřížka

Jak už jsem zmínil, tak obalová tělesa je možné zkombinovat dohromady s dalšími metodami a získat tím ještě schopnější algoritmus.

3.1 Prostorové dělení scény

Metody založené na tomto principu rozdělují scénu do buněk. Tak, že jistým způsobem řadí objekty ve scéně v závislosti na jejich vzájemné vzdálenosti od sebe. Tato vzdálenost je poté uložena do parametrů datové struktury. Kvalita řazení těchto vzdáleností je poté rozhodující při přístupu paprsku k danému objektu. Společný princip všech těchto struktur je takový, že scénu vymezenou obalovým boxem AB rozdělí na množinu buněk vložením množiny hranic, vymezující prostor mezi buňkami. Každá z těchto buněk obsahuje seznam ukazatelů na objekty, které jsou buď celkově uvnitř buňky, nebo částečně.

Algoritmus pro průchod těmito strukturami funguje tak, že jsou nalezeny buňky v dráze paprsku. Jestli existuje průsečík mezi objektem a paprskem uvnitř této buňky, a jestli tento průsečík zároveň leží uvnitř buňky, tak je tato hodnota vrácena jako výsledek hledání. Pokud je v buňce

obsaženo více objektů, tak je vrácena hodnota toho nejbližšího. V opačném případě, kdy průsečík není nalezen, tak algoritmus pokračuje v testování další nejbližší buňky ležící na dráze paprsku. Algoritmus identifikující tyto buňky v dráze paprsku se nazývá jako algoritmus průchodu paprskem (*ray traversal algorithm*).

Společnou vlastností těchto struktur je ta, že žádná z těchto elementárních buněk se nepřekrývá a konstrukce u těchto struktur obvykle probíhá stylem shora dolů.

3.1.1 BSP stromy a k-D stromy

BSP stromy neboli *Binary Space Partitioning* (binární prostorové dělení) je struktura používaná na řešení různých geometrických problémů. Původně byly vyvinuty jako prostředek pro řešení problému skrytého povrchu v počítačové grafice. Jsou analogické k binárním vyhledávacím stromům, akorát jsou použitelné i pro vyšší dimenze. BSP stromy existují ve dvou variantách a to jako osově souměrné a polygonově souměrné.

Polygonově souměrné BSP stromy se používají zejména ve scénách složených pouze z polygonů, a proto nenalezli širší uplatnění mezi aplikacemi používající metody sledování paprsků. Jejich princip dělení spočívá v nalezení plochy pod polygonem a následném rozdělení na dvě části.

Osově souměrné BSP stromy fungují na principu dělení prostoru plochou kolmou na jednu ze souřadnicových os. To činí právě osově souměrné BSP stromy jako výhodné pro použití se sledováním paprsku, protože umožňuje výrazně zjednodušit testování průsečíku s rozdělující plochou. Z testů provedených V. Havranem [13] bylo dokázáno, že použití osově souměrných řezacích ploch a počítání průsečíku s nimi je až třikrát rychlejší než výpočet průsečíku s použitím libovolně umístěných ploch.

BSP strom pro množinu objektů scény S, je definován takto: Každý uzel u v BSP stromu je asociován s osově souměrným obalovým boxem $AB(u)$, který je zároveň buňkou. Buňka asociovaná s kořenem BSP stromu je osově souměrný obalový box AB, který obsahuje všechny objekty v množině S. Každému vnitřnímu uzlu u BSP stromu je přiřazena řezací plocha p_u, která rozděluje $AB(u)$ na dvě buňky. Nechť p_u^+ je kladná polovina a p_u^- záporná polovina ohraničená p_u. Buňky asociované s levým a pravým potomkem uzlu u jsou $AB(u) \cap p_u^+$ a $AB(u) \cap p_u^-$. Levý podstrom uzlu u je BSP strom pro množinu objektů $S_u^- = \{s \cap p_u^- \neq \emptyset \mid s \in S_u\}$, obdobně je definováno pro pravý podstrom. Každý *listový* uzel u^E může obsahovat seznam objektů S_{u^E}, které se protínají s osově souměrným obalovým boxem $AB(u^E)$, patřící uzlu u^E. Jestli uzel obsahuje minimálně jeden objekt, nazývá *plný list*, v opačném případě *prázdný list*.

BSP strom se obvykle konstruuje stylem shora dolů. V aktuálním listu u je vybrána řezací plocha, která rozdělí $AB(u)$ na dvě buňky. List se poté stane vnitřním uzlem s dvěma novými listy. A objekty asociované s u se rozdělí mezi dva nově vzniklé potomky. Proces dělení se opakuje rekurzivně, dokud nejsou splněny podmínky k ukončení. Tyto podmínky bývají většinou určitý počet objektů v listu nebo maximální hloubka listu.

Algoritmus 1 konstrukce BSP stromu v pseudokódu

```
procedure BuildTree(aktuální_buňka, aktuální_hloubka, řezací_plocha)
begin
   if ((aktuální_buňka obsahuje příliš mnoho objektů)
        and (aktuální_hloubka není příliš veliká))
   then
      kladný_potomek->obalové_těleso = aktuální_buňka->obalové_těleso
      záporný_potomek->obalové_těleso = aktuální_buňka->obalové_těleso
      if{ řezací_plocha = X-osa }
      then
         kladný_potomek->obalové_těleso.X.min = střed aktuální_buňka->obalové_těleso.X
         záporný_potomek->obalové_těleso.X.max = střed aktuální_buňka->obalové_těleso.X
         nová_řezací_plocha = Y-osa
      else if{ řezací_plocha = Y-osa }
      then
         kladný_potomek->obalové_těleso.Y.min = střed aktuální_buňka->obalové_těleso.Y
         záporný_potomek->obalové_těleso.Y.max = střed aktuální_buňka->obalové_těleso.Y
         nová_řezací_plocha = Z-osa
      else if{ řezací_plocha = Z-osa }
      then
         kladný_potomek->obalové_těleso.Z.min = střed aktuální_buňka->obalové_těleso.Z
         záporný_potomek->obalové_těleso.Z.max = střed aktuální_buňka->obalové_těleso.Z
         nová_řezací_plocha = X-osa
      end if
      for (všechny objekty v  aktuální_buňka ) do
         if ( kladný_potomek->obalové_těleso obsahuje objekt ) then
             kladný_potomek ->přidej_objekt(objekt)
         end if
         if ( záporný_potomek->obalové_těleso obsahuje objekt ) then
            záporný_potomek->přidej_objekt(objekt)
         end if
      end for
      BuildTree(kladný_potomek, aktuální_hloubka + 1,  nová_řezací_plocha)
      BuildTree(záporný_potomek, aktuální_hloubka + 1, nová_řezací_plocha)
   else
       aktuální_buňka = list
   end if
end
```

Uvedený algoritmus 1 funguje na principu dělení obalového tělesa na dvě stejné poloviny, s tím, že se orientace plochy sekvenčně mění po všech osách.

Veškerá inteligence BSP a k-D stromů leží právě ve správném umístění řezací plochy, a toho se dá dosáhnout jedině nějakým algoritmem, který je schopen se adaptovat na scénovou geometrii a vybrat ty nejlepší pozice řezacích ploch. Metody pro výběr řezacích ploch jsou zásadním bodem při konstrukci těchto stromů, jelikož správná konstrukce stromu poté rozhoduje o tom, kolik času se stráví s testováním a hledáním nejbližšího průsečíku.

Umísťování řezacích ploch se někdy používá k odlišení BSP stromů od k-D stromů, publikováno autorem Bentley [4]. V podstatě jsou ale oba ekvivalentní, nicméně v různých zdrojích se často liší jejich definice.

K-D stromy jsou tedy všechny BSP stromy s osově souměrnými řezacími plochami, a stejně jako u BSP stromů, mohou tyto plochy být umístěny dle libosti kdekoliv podél osy, na kterou jsou kolmé. Nicméně naopak některé literatury definují osově souměrné BSP stromy jako případ k-D stromů, které umísťují řezací plochy doprostřed aktuálního boxu, takže vzniknou vždy dva noví potomci s buňkami stejné velikosti, obdobně jako je tomu v algoritmu 1. V této práci jsou tedy k-D stromy brány jako všechny osově souměrné BSP stromy s možností libovolného umístění řezací plochy, samozřejmě se zachováním kolmosti k některé z os, zatímco BSP stromy jsou pouze ty, které umísťují řezací plochy doprostřed. Z práce publikované Vlastimilem Havranem v [13], je zřejmé, že umísťování řezacích ploch má velký vliv na výkon metody sledování paprsku, a to zejména ve scénách s nerovnoměrným rozložením objektů.

Algoritmus 2 pro průchod BSP a k-D stromem

```
function TreeIntersect(paprsek, buňka, min, max): objekt
begin
   if ( aktuální_buňka je prázdná ) then
      return „žádný průsečík“
   else
      if ( aktuální_buňka je list ) then
         for (všechny objekty v  aktuální_buňka ) do
            nalezni průsečík s objektem a zahoď ho, pokud leží dál, než je hodnota max
            return „objekt s nejbližším průsečíkem“
         end for
      else
         t = vzdálenost paprsku od řezací plochy v buňce
         bližší_potomek = potomek buňky obsahující počátek paprsku
         vzdálenější_potomek = druhý potomek buňky
      if(( t < 0 ) or ( t > max )) then
         return TreeIntersect(paprsek, bližší_potomek, min, max)
      else
         if( t < min ) then
            return TreeIntersect(paprsek, vzdálenější_potomek, min, max)
         else
            hit_data = TreeIntersect(paprsek, bližší_potomek, min, t)
         if( hit_data udávají, zda byl nějaký objekt zasažen ) then
            return hit_data
         else
            return TreeIntersect(paprsek, vzdálenější_potomek, t, max)
         end if
      end if
   end if
end
```

Tento algoritmus je základním algoritmem pro průchod BSP a k-D stromů, je to jeden z prvních typů rekurzivního algoritmu průchodu paprskem. Jako takový byl už několikrát popsán v [3], [13]. V principu mohou při průletu paprsku stromem nastat tři případy chování tohoto algoritmu. První, že

počátek paprsku a i zbytek délky leží na straně kladné od řezací plochy, takže je testován bližší potomek, v druhém případě leží celý paprsek na záporné straně od řezací plochy, je testován vzdálenější potomek, a ve třetím případě leží počátek na kladné straně, ale směruje částečně do záporné strany, takže je testován první bližší potomek. Pokud je nalezen průsečík, algoritmus se ukončí, jinak pokračuje ve hledání ve vzdálenějším potomku buňky.

3.2 Konstrukce k-D stromu

Konstrukce k-D stromu vychází z algoritmu 1., s tím rozdílem, že poloha řezací plochy není pevně daná, ale může se měnit v závislosti na složitosti rozmístění objektů ve scéně. Typy metod, jakým způsobem se řezací plochy vybírají, budou rozebrány v následující části 3.2.3.

3.2.1 Časová složitost

Důležitým parametrem konstrukčního algoritmu k-D stromu je časová složitost. Je zřejmé, že pokud klasický naivní algoritmus sledování paprsku v komplexní scéně zabíral při výpočtu průsečíků až 95% výpočetního času, a celková doba výpočetního času zároveň rostla v lineárně s počtem objektů ve scéně, tedy pracoval s nejhorší možnou časovou složitostí $O(N)$, kde N je počet objektů.

Díky tomu se nedal téměř vůbec použít pro vykreslování realistických scén. Výjimku tvořily aplikace založené na sledování paprsku a tzv. real-time renderingu, kde stíhal, a to pouze při použití velmi nízkého počtu objektů nebo s použitím obalových těles, ale v takovém případě už realističnost scény nehrála takovou roli.

Pro k-D stromy je otázka rychlosti vykreslování, které dokážou vykreslovat složité scény několika set násobně rychleji než při použití klasické naivní metody, stejně důležitá jako rychlost konstrukce. Nynější algoritmy pro průchod stromem potřebují $O(log\ N)$ operací v nejlepším případě a $O(N\ log\ N)$ v nejhorším pro nalezení nejbližšího průsečíku, kde N je počet buněk ve struktuře stromu. Výhodou algoritmu je v tomto případě nezávislost na počtu objektů ve scéně, protože se dá předpokládat, že počet objektů patřících jedné buňce je omezený, takže složitost nalezení správného objektu v listu stromu patří do $O(1)$.

Otázka rychlosti konstrukce k-D stromu je druhý problém. V jistých případech, konkrétně ve statických scénách je rychlost konstrukce sice méně důležitá, nicméně pro využití v dynamických scénách je tento problém poměrně zásadní.

V minulosti byl výzkum k-D stromů téměř exklusivně zaměřen pouze na rychlé algoritmy průchodu nebo na algoritmy, jak stavět tyto stromy co nejefektivněji, aby počet testovaných objektů v dráze paprsku zůstal co nejmenší. Otázka ceny a složitosti konstrukce zůstávala široce ignorována. V poslední době díky požadavkům vytvořit co nejrealističtější scény a kdy se případně uvažuje o nasazení metody sledování paprsku do her, či jiných grafických aplikací vystává problém.

Časová složitost konstrukce kvalitních stromů se nyní díky pokroku ve vývoji pohybuje na hranici složitosti $O(N\ log\ N)$, v horším případě až na $O(N^2)$, N počet objektů. S tím, že algoritmus pro hledání nejbližších průsečíků pracuje v nejhorším případě se složitostí $O(N\ log\ N)$, N počet buněk stromu. Zatímco klasická naivní metoda pracuje se složitostí $O(N)$, N počet objektů.

3.2.2 Výhody k-D stromu

Výběr správné metody pro optimální umístění řezací plochy při konstrukci k-D stromu je nejdůležitější součástí algoritmu. Zde se potom rozhoduje o velikosti buněk, do kterých se přiřazují jednotlivé objekty. Přiřazení objektů k buňkám se dá předpokládat z toho, že víme, že každý objekt ve scéně je konečné velikosti, takže pro něj musí také existovat obalový box AB konečné velikosti schopný tento objekt celkově nebo částečně obalit. A proto po umístění řezací plochy algoritmus rozhoduje, zdali budou jednotlivé objekty přiřazeny levému, pravému nebo oběma potomkům obalového boxu rozděleného řezací plochou.

Mezi jednu z dalších výhod k-D stromů patří jejich schopnost se přizpůsobit prostoru libovolné dimenze, což je vlastně zřetelné z názvu této struktury, která říká že k-D strom je k-Dimenzionální stromová struktura.

K-D strom je dále svou unikátností schopen topologicky modelovat vesměs většinu zmíněných struktur prostorového dělení. Mezi které patří octree, rovnoměrné i nerovnoměrné mřížky nebo quadtree v případě 2-D prostoru.

Jednou z výhod je také paměťová náročnost k-D stromu. Kdy počet buněk stromu je téměř lineární s počtem objektů ve scéně, citováno z výzkumu provedeného V. Havranem v [13].

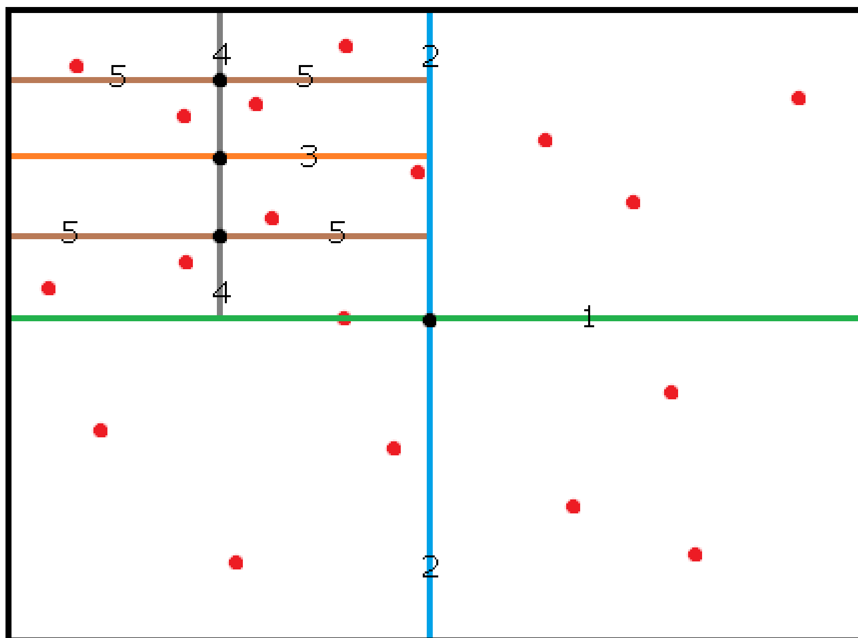
Další předpoklad využití a zároveň jedna z výhod k-D stromu je schopnost se vyrovnat s nerovnoměrně složitou scénou, což je typické v případě realistických scén ve sledování paprsku. Kde klasické prostorové dělení s rovnoměrnou strukturou výkonnostně velmi zaostává. V jistých případech, kdy je scéna méně složitá a více rovnoměrná rozložená jsou metody s rovnoměrným dělením trochu výkonnější. Tyto rozdíly ve výkonu a složitosti scén budou více zřejmé v části testování.

3.2.3 Umísťování řezací plochy

Prací, věnujících se problematice nejefektivnějších metod pro umístění řezacích ploch, je značné množství, MacDonald v [18], Vlastimil Havran v [13], [22], Wald Ingo v [21]. Ve všech těchto pracích se uvádí algoritmus *SAH* tzv. *Surface Area Heuristic*, přináší téměř dvojnásobný výkon oproti klasickým metodám umísťování, s tím, že zachová dobu konstrukce v časové složitosti s asymptotickým dolním omezením $O(N \log N)$.

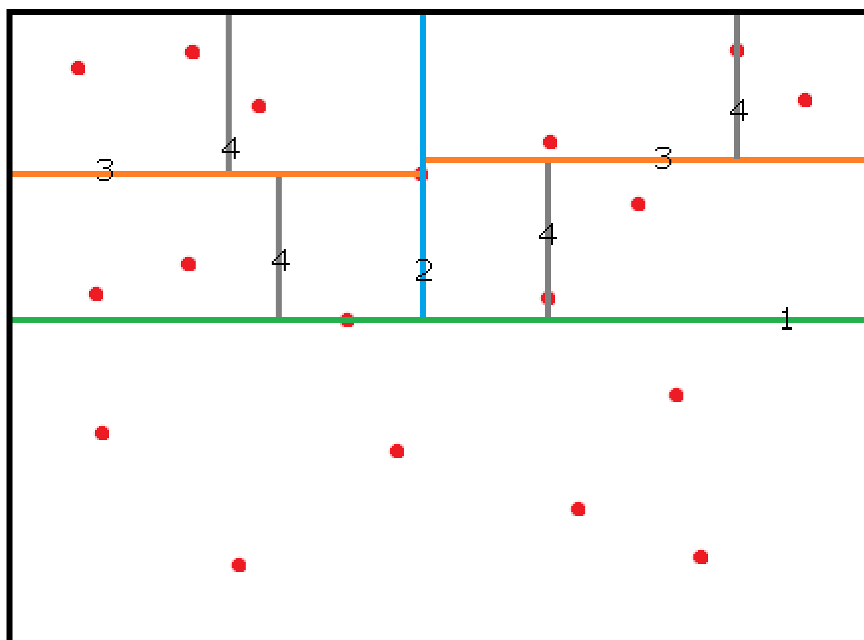
Zde jsou zmíněny dosud známé techniky pro umísťování řezacích ploch.

Prostorový medián: Základní metoda pro umístění řezací plochy v BSP stromu, aktuální uzel je rozdělen na dvě velikostně stejné poloviny, proto s rostoucí hloubkou zůstávají potomci uzlů stále stejně veliké. Tato metoda tedy vyrovnává prostor na obou stranách řezací plochy. Na obrázku 3.1 lze vidět, jak takový algoritmus funguje ve 2D prostoru, ve 3D metoda probíhá stejným způsobem akorát s jednou souřadnicí navíc.



Obrázek 3.1: Ilustrace metody půlení prostorovým mediánem, typické pro BSP strom

Objektový medián: Metoda podobná prostorovému mediánu, akorát, že namísto vyrovnávání prostoru po stranách řezací plochy vyrovnává počet objektů. Můžou nastat případy, že objekty jsou přiřazeny oběma potomkům. Tato metoda se jeví, že by měla být pro umístění řezacích ploch ideální a zároveň nejúčinnější, jelikož v jednorozměrném prostoru, například při řazení čísel, to tak platí. Problém je v tom, že i přes toto vyrovnané rozdělení objektů v obalových boxech nastávají případy, že paprsek neprotne žádný objekt v prvním listu a musí pokračovat v hledání dalších po cestě dráhy paprsku. Na obrázku 3.2 je ilustrován postup půlení touto metodou.



Obrázek 3.2: Ilustrace metody pro umísťování řezací plochy do objektového mediánu

Cenový model: Podle V. Havrana [13] tento model překonává obě předchozí metody pro optimální umístění řezací plochy ve všech testovaných scénách, princip spočívá v počítání průměrné ceny průletu paprsku skrz k-D strom během jeho stavby. Metoda byla představa v roce 1989 McDonaldem a Boothem, s její revizí v roce 1990 [18].

3.2.4 Ukončovací kritéria konstrukce k-D stromu

Po přehledu konstrukčních algoritmů, které dávají recept na to kam vložit řezací plochu, zbývá ještě jedna esenciální otázka, na kterou neodpovídají. Kdy ukončit půlení buněk stromu, a kdy označit buňku jako list?

Každá z buněk k-D stromu u má tyto charakteristické parametry:

- Hloubku $d(u)$ od kořenové buňky.
- Osově souměrný obalový box $AB(u)$.
- Seznam ukazatelů na všechny objekty N, protínající $AB(u)$.

Z hlediska principu dělení ve struktuře k-D stromu se dá předpokládat, že počet objektů N v buňce $AB(u)$, klesá s hloubkou $d(u)$. Takže z pohledu na tyto tři charakteristické parametry každé buňky si lze položit otázku, jak s pomocí těchto tří parametrů rozhodnout, kdy ukončit dělení buněk?

3.2.4.1 Ad Hoc ukončovací kritérium

Ad Hoc ukončovací kritérium bylo publikováno zároveň s uvedením algoritmu pro využití BSP stromů ve sledování paprsku M. Kaplanem v [15]. Je založeno na dvou předchozích parametrech buněk k-D stromu, a to na hloubce buňky $d(u)$ a počtu objektů protínající buňku N.

Podle Ad Hoc kritéria je buňka stromu u označena jako list v momentě kdy počet objektů protínajících obalový box $AB(u)$ je menší nebo roven pevně definované konstantě n_{max}, nebo jeho hloubka $d(u)$ dosáhne na další fixní hodnotu d_{max}. Kde n_{max} a d_{max} jsou hodnoty definovány uživatelem.

3.3 Rozbor konstrukčních metod

V této části se podrobněji zaměřím na všechny doposud zmíněné metody umístění řezacích ploch a tedy i zároveň typy konstrukce k-D stromu. Kromě dvou klasických, a to prostorového a objektového mediánu výběru řezací plochy, rozeberu způsob konstrukce s pomocí cenové modelu, známého více pod zkratkou *SAH* (dále už jenom pod *SAH*), neboli *Surface Area Heuristic.*

3.3.1 Prostorový medián

Metoda půlení prostorovým mediánem je identická s uvedeným algoritmem 1, který zároveň slouží jako algoritmus konstrukce BSP stromu.

Drobnou změnou lze dosáhnout v závislosti na složitosti scény vyššího výkonu. Tuto techniku zmínil V. Havran v [13]. Způsob jak toho dosáhnout leží ve správném výběru osy pro řezací plochu. Klasický způsob je založen na sekvenčním střídání os řezacích ploch (tzv. *round robin*), nový způsob je založen na výběru osy ve kterém má daný aktuální $AB(u)$ box buňky u tu největší délku. Tuto techniku je možné použít jak v objektovém mediánu, tak při cenovém modelu.

Z hlediska času konstrukce je prostorový medián nejrychlejší metodou, z analýzy časové složitosti rovnice 3.1 pro určení řezací plochy lze rozpoznat, že výběr polohy řezací plochy p_u pro aktuální buňku je vykonán v konstantním čase v nezávislosti na počtu objektů v aktuálním obalovém boxu $AB(u)$. Časová složitost algoritmu patří do $O(1)$.

Kde p_d je osa podle které se bude dělit aktuální $AB(u)$, ta se určí se znalosti hloubky aktuální buňky $d(u)$.

$$p_d = d(u)\ mod\ 3 \qquad p_u = \frac{1}{2}(AB(u)_{min,\,p_d} + AB(u)_{max,\,p_d}) \qquad \textit{rovnice 3.1}$$

3.3.2 Objektový medián

Půlení objektovým mediánem je metoda založená na rozdělení obalového boxu $AB(u)$ na dvě poloviny, kde každý z potomků obsahuje stejný počet objektů. Tato metoda tedy vyrovnává rozložení objektů po stranách řezací plochy p_u. Pro výpočet polohy p_u je třeba znát centra všech objektů ve scéně, nebo v případě scény složené jen z trojúhelníků stačí vzít souřadnice jednoho z vertexů. Jakmile jsou známy souřadnice všech objektů, stačí opět určit orientaci budoucí řezací plochy p_d podle klasického střídání nebo podle nejdelší dimenze $AB(u)$.

Vybráním a seřazením souřadnic ležících v ose p_d lze snadno určit medián a tím mít určenou polohu p_u.

$$p_d = d(u)\ mod\ 3 \qquad p_u = median(array(N_{center,\,p_d})) \qquad \textit{rovnice 3.2}$$

Zde je čas konstrukce založen zejména na rychlosti určení mediánu ze souřadnic všech objektů ve scéně a tedy podle použitého řadícího algoritmu. Z řadícího algoritmu lze jednoduše určit i

složitost určení polohy řezací plochy. Ze známých řadících algoritmů patří nejrychlejší pod $O(N \log N)$, a to je tedy i složitost určení polohy plochy pomocí objektového mediánu.

Ve výsledku měření se řadí tento strom rychlostně před prostorový medián, v porovnání se *SAH* je výkonnostně daleko za ním. Jedním ze strukturálních problémů je ten, že strom vytvořený pomocí této metody neobsahuje téměř žádný prázdný list, což ho činí rychlostně vyrovnaným po celou dobu vykreslování.

3.3.3 Cenový model SAH

Tato technika byla vyvinuta v roce 1989 autory MacDonald a Booth, [18]. Jejich práce se zabývala faktem, že plocha konvexního objektu je úzce spjata s pravděpodobností, že objekt bude zasažen paprskem. Prováděli testy, ze kterých vyšel zřejmý výsledek této vzájemné závislosti.

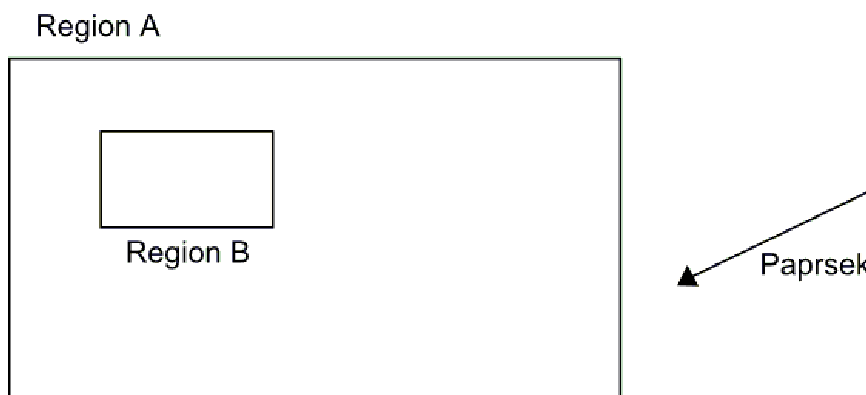
I v té době se již o této závislosti plochy objektu na pravděpodobnosti zásahu paprskem nějaké roky vědělo. Ale i přesto tento fakt, to trvalo dlouhou dobu, než se tím někdo začal více zabývat.

V další části studie *SAH* MacDonald prezentuje výsledky svého výzkumu a uvádí, že s užitím této metody je počet testovaných objektů na průsečík o tři řády nižší v porovnání s dalšími metodami. MacDonald tedy vymyslel algoritmus, který umísťuje řezací plochy do lokálně optimálních pozic. Tento algoritmus se projevil nevídanou rychlostí při vykreslování, byl až tisícinásobně rychlejší u specifických scén, konkrétně scény tvořené velkým počtem rovnoměrně rozložených malých objektů.

3.3.3.1 Geometrická pravděpodobnost

Cenový model je tedy úzce spjat s pozorováním, které provedl jak MacDonald a Booth v [18], tak už nějakou dobu před ním Stone v [20] a Goldsmith, Salmon v [9].

Z tohoto pozorování plyne že pravděpodobnost zásahu regionu B, ležícím kompletně v regionu A, paprskem, u kterého se předpokládá že zároveň protne region A, se dá vyjádřit jako $p_{B|A}$. Toto slovní vyjádření lze vidět na obrázku 3.3.



Obrázek 3.3

Tuto pravděpodobnost lze poté vyjádřit jako závislost plochy regionu B uvnitř k ploše regionu A obalující B, rovnice 3.3.

$$p_{B|A} = \frac{SA(B)}{SA(A)} \qquad \textit{rovnice 3.3}$$

Pro 3D prostor lze tuto rovnici jednoduše přepsat do podoby rovnice 3.4 a požadavků obalových boxů AB. Kde indexy w, h, d u boxů $AB(A)$ a $AB(B)$ jsou jejich rozměry, tedy šířka, výška a hloubka.

$$p_{B|A} = \frac{(B_w.B_h + B_w.BA_d + B_h.B_d)}{(A_w.A_h + A_w.A_d + A_h.A_d)} \qquad \textit{rovnice 3.4}$$

Na základě této pravděpodobnosti lze odhadnout cenu $C(p_u)$, pro vybranou řezací plochu p_u, pro jeden krok průchodu paprskem.

$$C(p_u) = \mathrm{K_T} + p_{V_l|V}.C(V_l) + p_{V_r|V}.C(V_r) \qquad \textit{rovnice 3.5}$$

Kde V_l a V_r jsou potomci (voxely) rodičovského obalového boxu AB rozděleného p_u řezací plochou. Cena $C(T)$ průchodu celým stromem T se dá vyjádřit rovnicí 3.6. Kde nejlepší strom T pro scénu S je právě ten, ve kterém má tato rovnice globální minimum.

$$C(T) = \sum\nolimits_{\mathrm{n \in buňky}} \frac{SA(V_n)}{SA(V_S)} \mathrm{K_T} + \sum\nolimits_{\mathrm{l \in listy}} \frac{SA(V_l)}{SA(V_S)} \mathrm{K_I} \qquad \textit{rovnice 3.6}$$

V_S je v rovnici 3.6 ve významu jako obalový box celé scény. V rovnicích 3.4 a 3.5 jsou ještě dva neznámé parametry $\mathrm{K_T}$ a $\mathrm{K_I}$. Parametr $\mathrm{K_T}$ lze vyjádřit jako cena průletu paprsku buňkou a $\mathrm{K_I}$ jako cena průletu paprsku listem. Tyto parametry jsou obvykle vyjádřeny jako konstanty před stavbou stromu.

3.3.3.2 Lokálně nenasytná heuristika SAH

Vyřešit rovnici 3.6 je nicméně i pro jednoduchou scénu téměř nemožná operace. Počet možných stromů roste extrémně rychle s velikostí scény.

Proto namísto řešení rovnice pro nalezení globálního minima lze použít lokální hladovou aproximaci (*greedy* algoritmus). U které se předpokládá, že při výpočtu ceny pro rozdělení buňky V plochou p_u se stanou oba vytvoření potomci listy.

$$C(p_u) = \mathrm{K_T} + \mathrm{K_I}\left(\frac{SA(V_L)}{SA(V)} \mathrm{N_L} + \frac{SA(V_R)}{SA(V)} \mathrm{N_R}\right) \qquad \textit{rovnice 3.7}$$

Tato aproximace je už použitelná pro implementaci lokálního výběru optimální řezací plochy. Důležitým poznatkem je to, že rovnice 3.7 je už částečně upravena do podoby pro implementaci, kde parametry $\mathrm{K_I}.\mathrm{N_L}$ a $\mathrm{K_I}.\mathrm{N_R}$ původně vyjadřovali odhadnutou cenu levého a pravého podstromu $C(\mathrm{T_L})$ a $C(\mathrm{T_R})$. Pro zjednodušení do nynější podoby se uvažovalo, že odhadnutou cenu lze vyjádřit počtem objektů v jednotlivých listech, tedy $\mathrm{N_L}$ a $\mathrm{N_R}$, a cenou pro průchod těmito listy $\mathrm{K_I}$. Cena takto odhadnutá nicméně nepředpokládá další dělení, skutečná cena proto může být i nižší.

Rovnice 3.7 je velkým zjednodušením rovnice 3.6, samotný odhad ceny může být touto aproximací lokálně sice přesný, nicméně cena výsledného stromu globálně nemusí dosahovat minima. I přesto tato metoda funguje velmi efektivně a přináší pro většinu testovaných scén obrovské urychlení vykreslování.

3.3.3.3 Automatické ukončovací kritérium

Metoda pro výpočet lokálně optimální řezací plochy byla rozebrána, nyní zbývá ještě určit kdy se má při použití *SAH* buňka označit jako list, případně jestli se má pokračovat s konstrukcí stromu.

Zde na rozdíl od klasických metod, kde se používá Ad Hoc kritérium, cenový model umožňuje použít pro ukončení konstrukce stromu, jak Ad Hoc model, tak vlastní elegantní řešení. Toto řešení je založeno na porovnání minimální ceny pro umístění řezací plochy $C(p_u)$ s odhadnutou cenou rodičovského stromu $C(\mathrm{T_L})$, a jak bylo zmíněno, tak lze tento odhad nahradit cenou průletu listem krát počet objektů v aktuální buňce $\mathrm{K_I}.\mathrm{N}$.

A tedy buňka je označena jako list, jestli platí podmínka v rovnici 3.8.

$$ukončitDělení\begin{cases} true; \; C(p_u) > \mathrm{K_I}.\mathrm{N} \\ false; jinak \end{cases} \qquad \textit{rovnice 3.8}$$

3.3.3.4 Výběr kandidátů řezacích ploch pro výpočet ceny

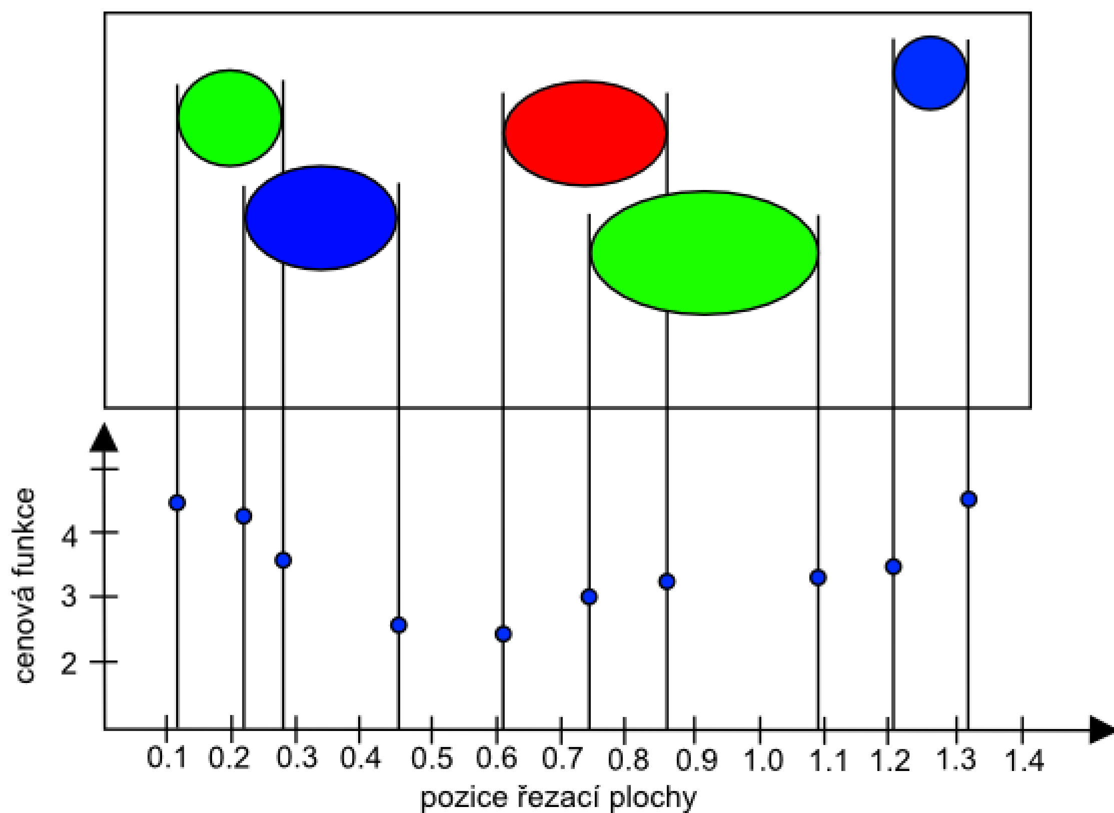
U dvou předchozích metod byl výběr polohy řezací plochy jednoduchý a rychlý. Nyní pro určení té nejlepší řezací plochy je zatím uvedena pouze metoda pro výpočet její ceny. Teoreticky na základě rovnice 3.6 je počet kandidátů na řezací plochu nekonečně mnoho.

Na základě lokální aproximace, která vychází ze změn počtu objektů na levé straně řezací plochy, resp. pravé straně, lze tyto kandidáty určit podle plochy, které mění právě tento počet objektů po jejich stranách.

Jednoduchý způsob jakým vybrat tyto kandidáty je obalit každý konečný objekt ve scéně obalovým boxem $AB(t)$, kde t je objekt. Poté lze jednoduše, jelikož k-D strom využívá osově souměrné boxy, určit kandidáty jako plochy tohoto obalového boxu.

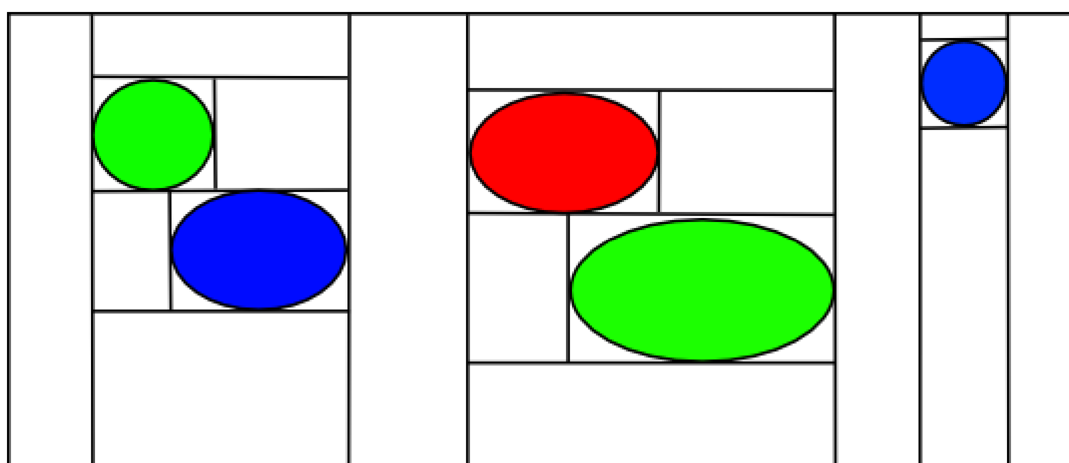
Jak lze vidět tak počet takových kandidátů na řezací plochu je $6N$, tedy pro každou osu dva kandidáti. Což je vzhledem k tomu, že počet objektů v komplexní scéně dosahuje několika milionů, tak dává šesti násobek počtu objektů. I zde lze uplatnit určité optimalizace, V. Havran v [13] ukázal, že tento počet je možné snížit, bez znatelného poklesu výkonu a to využitím už zmíněné metody výběru osy ve směru největší dimenze obalového boxu buňky $AB(u)$. Což dává ve výsledku pouze $2N$ kandidátů. Dále zmiňuje, že tento počet lze opět snížit na N kandidátů nalezením objektového mediánu a testováním pouze těch stran objektů blíže k tomuto mediánu. I přesto je ale N kandidátů je stále poměrně velký počet.

Na obrázku 3.4, který ilustruje odhad cenové funkce pro výběr řezací linie pro 5 objektů ve 2D prostoru, lze vidět, jakou roli hraje její poloha a počet objektů, které tato linie po obou stranách rozděluje. I na tomto obrázku se uplatnilo obalení objektů do obalových boxů AB, které umožnily výběr osově souřadné linie, a taktéž výběr osy dělení na základě nejdelší dimenze obalového boxu $AB(S)$.



Obrázek 3.4: Výpočet cenové funkce pro dělící plochy ve 2D

Při teoretické simulaci průběhu konstrukce k-D stromu pro obrázek 3.4 si lze všimnout, že *SAH* upřednostňuje vytváření co největších prázdných buněk a zároveň obalování objektů do co nejmenší možné buňky. V případě simulace konstrukce by celkový strom mohl vypadat podobně jako na obrázku 3.5.



Obrázek 3.5: Vytvořený 2D k-D strom s využitím heuristiky *SAH*

3.3.3.5 Časová složitost výběru nejlepší řezací plochy

Z hlediska určení časové složitosti je tato metoda velice závislá na implementaci. Jak jsem zmínil na začátku kapitoly 3.2, tak složitosti konstrukce kvalitních stromů patří pod $O(N\ log\ N)$, $O(N\ log^{2}N)$ nebo dokonce $O(N^2)$. Tyto stromy vznikají právě využitím heuristiky *SAH* pro výběr nejlepší dělící plochy.

V testovací části jsem implementoval právě algoritmus patřící pod $O(N\ log^{2}N)$, který se svou složitostí výběru řezací plochy patří stejně jako objektový medián pod $O(N\ log\ N)$.

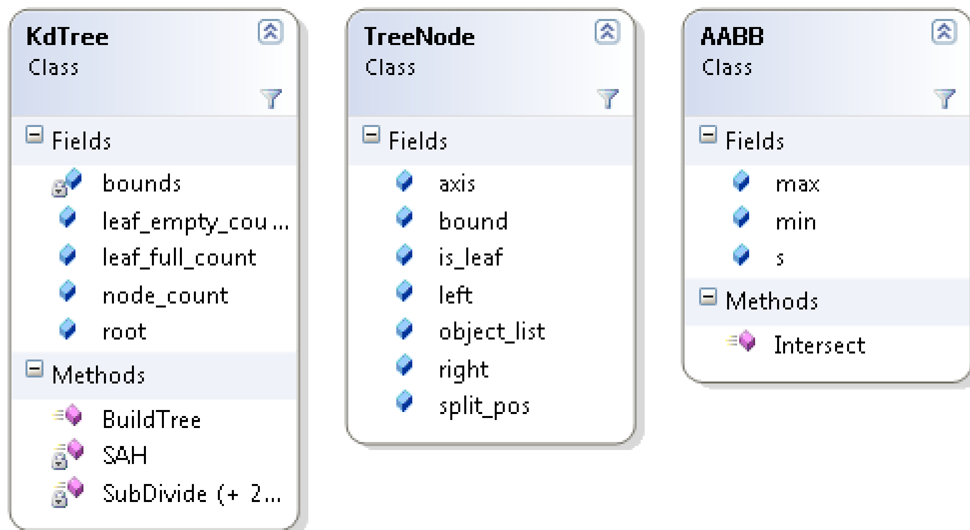
Kapitola 4

Implementace k-D stromu

Metody implementované v experimentálním raytraceru byly rozebrány v teoretické části této práce. V této kapitole se zaměřím na implementační část těchto algoritmů a na samotnou implementaci kompletní struktury k-D stromu.

4.1 Struktura k-D stromu

Před samotnou implementací konstrukčních algoritmů bylo třeba navrhnout jednotlivé třídy tvořící základní funkce struktury k-D stromu.



Obrázek 4.1: Digram základních prvků tvořících k-D strom

KdTree třída je stavebním kamenem každého k-D stromu. Proměnné a funkce, které ji v případě mé experimentální implementace tvoří, jsou:

- *BuildTree()* → funkce, která spouští konstrukci stromu
- *SubDivide()* → funkce obsahující implementaci všech konstrukčních algoritmů
- *SAH()* → funkce pro odhad ceny pro danou řezací plochu pomocí heuristiky *SAH*
- *bounds* → ukazatel na obalový box $AB(S)$ scény S
- *root* → ukazatel na kořenovou buňku k-D stromu, *TreeNode*
- *leaf_empty_count* → proměnná obsahující počet prázdných listů
- *leaf_full_count* → proměnná obsahující počet plných listů
- *node_count* → celkový počet buněk

Buňka stromu, tedy TreeNode na obrázku 4.1, je složena z částí, které jsem popisoval v teoretické části konstrukce:

- *axis* → směr normály dělící plochy aktuální buňky
- *is_leaf* → parametr, který určuje, zdali je aktuální buňka listem
- *left* → ukazatel na levého potomka
- *right* → ukazatel na pravého potomka
- *object_list* → seznam ukazatelů na objekty patřící do buňky
- *split_pos* → poloha řezací plochy

Nicméně existují způsoby, jak lze tyto proměnné uložit efektivněji a tím dosáhnout vyšší optimalizace. I. Wald v [21] navrhuje způsob jak uložit některé z parametrů buňky do velikosti jednoho celočíselného prvku. Tyto optimalizace se uplatní při algoritmu průchodu parskem, kde je třeba tyto parametry číst z paměti co možná nejrychleji, takto optimalizovaný kód podle Walda přináší zrychlení v řádu desítek procent.

Další částí k-D stromu je třída *AABB*, neboli *axis-aligned bounding box*. Tato struktura je velmi důležitá a je nutná preciznost při implementaci vnitřních funkcí obstarávající výpočet kolizí s objekty ve scéně. Tato struktura obsahuje pouze jednu funkci a tři proměnné, které se uplatňují při konstrukci stromu, a to jsou:

- *Intersect()* → funkce pro výpočet kolizí s objekty
- *min* → pole tří prvků obsahující minima pro všechny tři osy
- *max* → obdobně jako u předchozího, namísto minim obsahuje maxima
- *s* → rozpětí obalové boxu, obsahuje šířku, výšku a hloubku

Z algoritmů stojící za zmínku je metoda pro výpočet kolize s trojúhelníkem, a to proto, že trojúhelníky ve většině případů tvoří největší procento z celkového počtu objektů ve scéně. Algoritmus je taktéž důležitý pro správnou konstrukci k-D stromu, jelikož se právě zde rozhoduje o tom, jak buňka vybere optimální umístění řezací plochy a přerozdělí objekty do svého levého a pravého potomka. Špatná implementace může mít i velký dopad na využití paměti k-D stromem, proto je zde třeba dbát na přesnost a efektivnost. Časová složitost algoritmu výpočtu kolize rozhoduje částečně i o výsledné rychlosti konstrukce stromu, i když ne takovou měrou jako výběr optimální pozice řezací plochy v případě objektového mediánu a *SAH*.

4.1.1 Kolize AABB trojúhelník

Pro výpočet kolize trojúhelníku s obalovým boxem AB lze implementovat několik typů algoritmů. Pro můj experimentální raytracer jsem implementoval poměrně jednoduchou efektivní metodu testování. Je založena na několika testech, které dávají rychlou odpověď a přitom se snaží minimalizovat práci pro složitější případy. Výstupem testů těchto kolizí je pouze jednoduchá odpověď *true* pro kolizi, *false* v opačném případě.

V prvním testu jsou testovány vertexy trojúhelníku se 6 stranami obalového boxu AB. Jestli aspoň jeden vertex leží uvnitř obalového boxu AB, tak nastává kolize. V opačném případě, zdali všechny tři vertexy leží společně mimo, tedy před plochou tvořící jednu ze stran AB boxu, tak kolize nenastává.

Pokud trojúhelník projde prvním testem, dojde na porovnávání všech vertexů s plochami, které se dotýkají hran mezi jednotlivými sousedícími stranami obalového boxu a jsou pod úhlem 45° k této ploše. Opět pokud všechny vertexy leží mimo jednu z těchto ploch, kolize nenastává.

U posledního testu jsou vertexy porovnávány s plochami, které se dotýkají rohových bodů na obalovém boxu, tak že plocha svírá stejný úhel se všemi stranami. V tomto případě je test stejný jako v druhém testu, tedy pokud leží všechny vertexy mimo tuto plochu, kolize nenastává. Pokud trojúhelník prošel všemi těmito testy lze vrátit hodnotu pro stav, že došlo ke kolizi s obalovým boxem AB.

I přes tyto 3 testy lze dále omezit počet kolizních trojúhelníků zhruba o 14% (testováno K. Davidem v [16]), ale pro ne příliš složité scény s pár miliony trojúhelníků tento jednoduchý způsob postačuje, další typy algoritmů pro výpočet kolizí jsou na odkazu v [1].

4.1.2 Implementace SAH

Tento algoritmus je založen na práci V. Havrana, I. Walda v [22] pracujícím se složitostí $O(N\,log^2\,N)$. I když jsem původně implementoval svůj algoritmus pracující s $O(N^2)$, tak po testování jsem zjistil, že tudy správná cesta pro *SAH* nevede.

Než zde popíšu postup implementace, tak zmíním, kde spočívá její nejtěžší část a tedy hrdlo samotného konstrukčního algoritmu.

Nejprve je tedy nutné vybrat kandidáty na dělící plochu, v závislosti na vybraném způsobu těchto kandidátů může být $2N$, použitím pravidla výběru osy největšího rozsahu, nebo $6N$ při testování všech tří os. V případě milionu trojúhelníků tedy buď 2 miliony kandidátů ploch k testování, nebo 6 miliónů. Po výběru těchto ploch je v další části nutné spočítat počet N_L a N_R, tedy počet objektů po obou stranách kandidátní plochy p_u, a to pro každého kandidáta. Zde se začala má původní implementace lišit od té nové. Způsob jakým lze spočítat tyto objekty je rozhodujícím faktem pro výslednou časovou složitost.

Původní naivní algoritmus počítal tyto objekty pro každou kandidátní plochu způsobem, jakým se testuje kolize objektu s obalovým boxem AB. Samotný výběr ploch pracoval se složitostí $O(N)$, nicméně výpočet objektů po stranách kandidátní plochy musel opět proběhnout pro všechny objekty, takže se složitostí $O(N)$. Celková složitost v nejhorším případě mohla dosáhnout až $6N^2$, avšak při vyloučení multiplikativních konstant spadá pod $O(N^2)$, rychlost konstrukce k-D stromu v tomto případě pro celkem jednoduchou scénu obsahující 60000 trojúhelníků dosahovala extrémních časů. Proto jsem tuto implementaci zavrhl a zaměřil se na efektivnější implementaci.

4.1.2.1 Konstrukce SAH v $O(N\,log^2\,N)$

Tento algoritmus je založen na efektivnějším výpočtu N_L a N_R pro kandidátní plochu p_u. Vychází z jednoduchého předpokladu, že je znám počet všech kladných a záporných objektů p^+ a p^-, jako záporný počet p^- se myslí všechny objekty ležící od kraje obalového boxu AB až po polohu plochy, respektive p^+ všechny objekty začínající svou polohu na ploše a končící na opačném konci obalového boxu.

Pro správný výpočet p^+ a p^- je třeba seřadit všechny kandidátní plochy od nejmenší polohy po největší. Toho se dá dosáhnout algoritmem se složitostí $O(N\,log\,N)$. Dále už stačí jednoduchým algoritmem pracujícím v $O(N)$ spočítat počet N_L a N_R.

Dále zmíněný algoritmus 3 pracuje na principu postupného posunování polohy kandidátní plochy, které jsou seřazené od nejmenší po největší a uložené do seznamu kandidátů E včetně typu jejich orientace, tedy jestli byly na začátku objektu nebo na jeho konci (zřetelněji lze vidět tyto typy ploch na obrázku 3.4, kde každý objekty vytváří jednu linii vlevo, tedy záporný typ, a jednu vpravo, kladný typ), po směru osy a následného přesypávání objektů N z jedné strany této plochy na druhou. V je v algoritmu opět ve významu obalového boxu AB.

Algoritmus 3 pro výpočet $\boldsymbol{N_L}$ a $\boldsymbol{N_R}$ v $\boldsymbol{O(N)}$

function *NalezniPlochu*(*A, N, V , E*): nejlepší poloha plochy
begin
    $N_L = 0, N_R = N$
    {iterace přes všechny kandidáty v E}
    for *i* = 0; *i* < počet kandidátů;
        $p = E_{i,p}$
        $p^+ = p^- = 0$
        while((i < počet kandidátů) **and** ($E_{i,p} = p$) **and** ($E_{i,typ} = -$))
            zvyš p^-; zvyš i
        while((i < počet kandidátů) **and** ($E_{i,p} = p$) **and** ($E_{i,typ} = +$))
            zvyš p^+; zvyš i
        {následuje přepočet objektů pro pravou a levou stranu dělící plochy}
        $N_R -= p^-$
        {odhad ceny}
        $C = SAH(V, A, N_L, N_R)$
        if $C < C^\tau$ **then**
            $C^\tau = C$
        end if
        $N_L += p^+$
    end for
    return C^τ
end

Tento algoritmus pro rychlý výpočet počtu objektů N_L, N_R a následně nalezení nejnižší ceny pro vybranou polohu je aktuálně nejefektivnější metodou pro optimální umístění řezací plochy. Časová složitost výběru plochy pro jednu buňku spadá pod $O(N\ log\ N)$, ale z hlediska celkového času konstrukce je časová složitost v $O(N\ log^2\ N)$, podle analýzy v [21].

Při pohledu na algoritmus 3 a zmíněny postup si lze povšimnout, že proces výběru plochy počítá pokaždé s tím, že se polohy kandidátů seřazují pro každý stupeň rekurze. Pokud by bylo možné tyto kandidáty seřadit pouze jednou pro všechny osy, a to například při první vykonané rekurzi, a poté udržovat toto pořadí s tím, že při vyšší hloubce stromu by se odstranili kandidáti, kteří leží mimo rodičovskou buňku. Tak lze dosáhnout celkové složitosti konstrukce k-D stromu se složitostí $O(N\ log\ N)$.

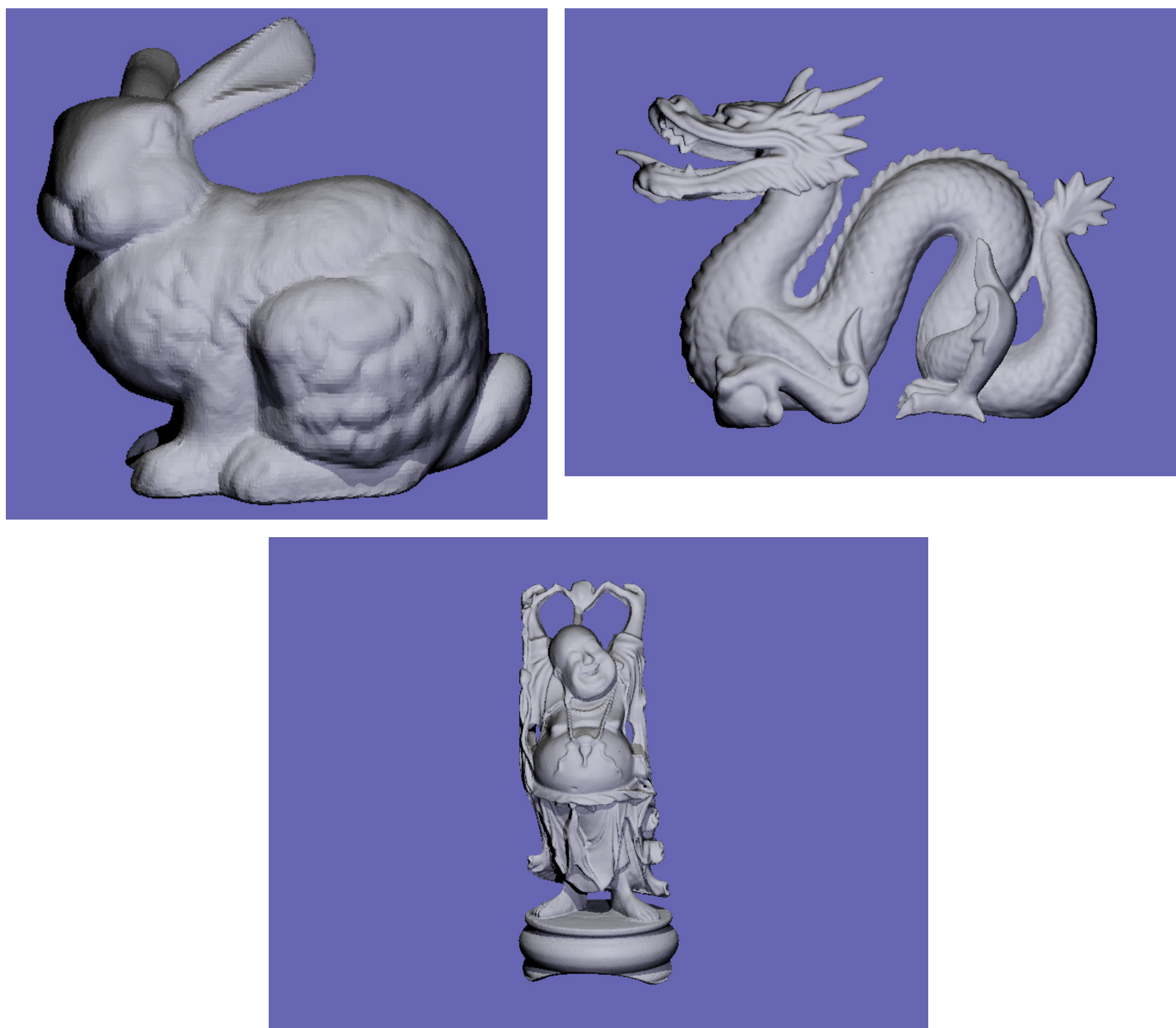
Kapitola 5

Testování konstrukčních algoritmů

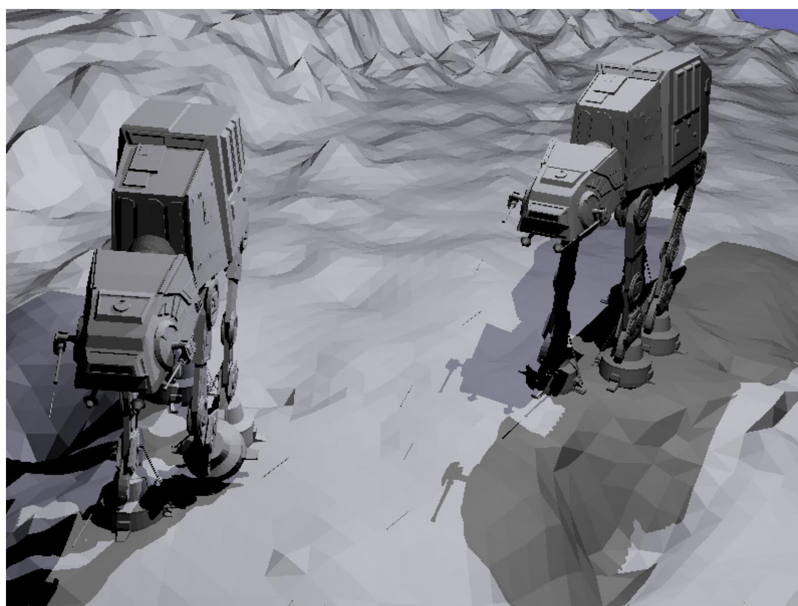
Tato kapitola je zaměřená pro vytvoření uceleného přehledu o testovaných parametrech, použitých scénách a samotných výsledcích z měření výkonnosti konstrukčních algoritmů k-D stromu.

5.1 Testovací scény

Pro testování výkonnosti jsem použil poměrně známé modely, které bývají taktéž součástí měření ve výzkumných pracích pro hodnocení rychlosti grafických algoritmů sledování paprsku. Obrázky 5.1 až 5.3.



Obrázky 5.1, 5.2, 5.3: Zleva doprava model *stanford bunny*, *dragon* a *happy budha*. Modely byly vykresleny testovacím experimentálním raytracerem



Obrázek 5.4: Model *robots*. Vlastní testovací scéna modelovaná v open-source softwaru *Blender*

Tři modely na předcházející straně pocházejí z repositáře Stanfordské Univerzity pořízené skenováním skutečných soch, modely v různých formátech jsou dostupné na odkazu v [19]. Poslední testovaný model na obrázku 5.4 pochází z vlastní tvorby. První tři scény jsou svou velikostí a rozložením trojúhelníků poměrně rovnoměrné, proto jsem přidal scénu, která je specifická velmi nerovnoměrným rozložením objektů po celé scéně, ve které by se měl nejvíce projevit algoritmus *SAH*.

Testovací sestava:

- Procesor: Intel Pentium Dual-Core E2160 2700 MHz
- Operační paměť: 2048MB DDR2 800MHz
- Testováno v rozlišení 320x240

Parametry testovacích scén, seřazené vzestupně podle počtu trojúhelníků:

- *Bunny* 69 451 trojúhelníků
- *Robots* 241 100 trojúhelníků
- *Dragon* 871 414 trojúhelníků
- *Budha* 1 087 716 trojúhelníků

5.2 Metodika testování

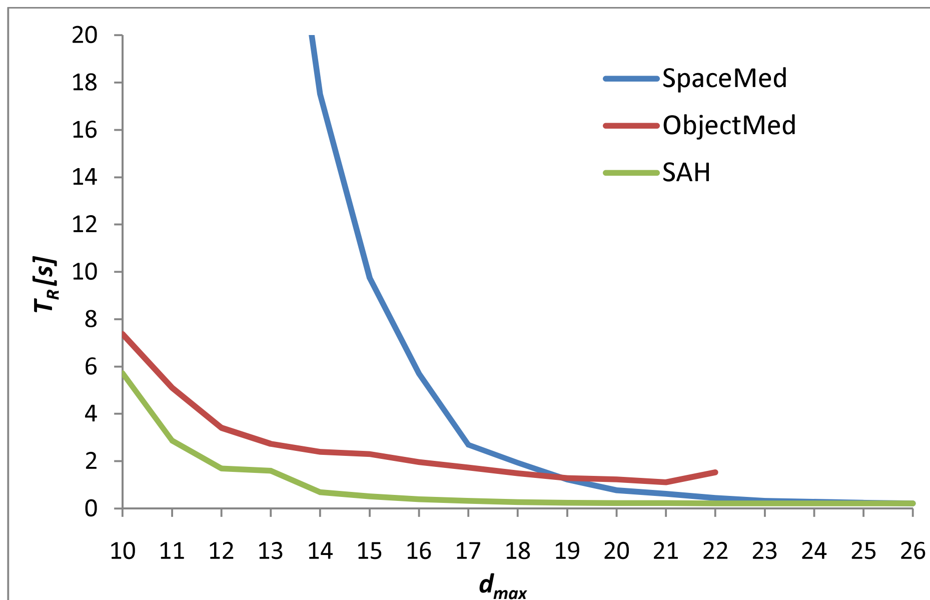
Před samotným testováním konstrukčních algoritmů je nutné vědět, jaké parametry stromu jsou vhodné k testování. Pokud se zaměřím pouze na časy konstrukce a vykreslování, tak vzhledem k tomu, že výsledky jsou velmi závislé na použité scéně, implementaci konstrukčního algoritmu, testovací sestavě a vstupních parametrech, a v případě doby vykreslování i na použitém algoritmu průchodu paprskem, tak nemají z objektivního hlediska efektivity konstrukčního algoritmu téměř žádnou vypovídající hodnotu.

Proto jsem se rozhodl otestovat předem vstupní parametr d_{max}, který mi umožní při komplexním testování dalších parametrů k-D stromu objektivnější výstupy. Mezi měřené výstupní hodnoty raytraceru patří:

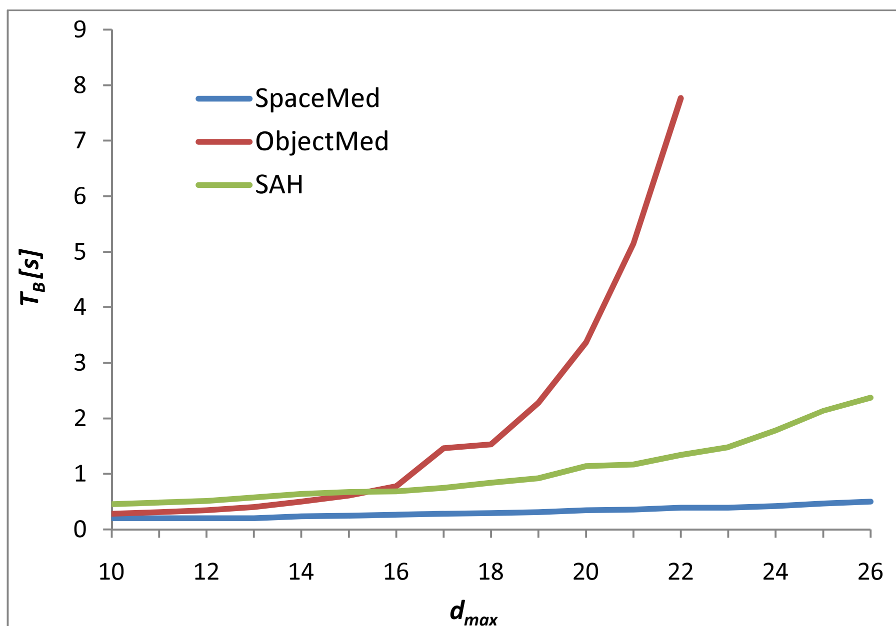
- N Celkový počet objektů
- L Počet světel
- R Počet vystřelených paprsků
- T_R Doba vykreslování
- T_B Doba konstrukce k-D stromu
- K Celkový počet kolizí s objekty
- C Počet buněk k-D stromu
- C_F Počet plných listů
- C_E Počet prázdných listů
- N_C Součet objektů ve všech listech
- N_P Průměrný počet objektů na list

5.3 Výsledky měření

V následujících grafech 5.1 a 5.2 (pouze pro scénu *Bunny*) jsem měřil dobu vykreslení a konstrukce k-D stromu, v závislosti na jeho maximální hloubce d_{max}, s tím že parametr n_{max} byl nastaven na hodnotu 1. Výsledky z těchto grafů posloužily k definování optimální hloubky d_{max} při testování dalších parametrů k-D stromu.



Graf 5.1: Závislost doby vykreslení na maximální hloubce d_{max}



Graf 5.2: Závislost doby konstrukce na maximální hloubce d_{max}

Nastavením pevné hodnoty maximální hloubky stromu lze podstatně omezit paměťové nároky k-D stromu, z tohoto pevného omezení plyne i maximálně možný počet vytvořených buněk stromu na hodnotu $2^{d_{max}}$.

Jak je vidět v grafech 5.1 a 5.2, tak pro objektový medián končí průběh na hodnotě hloubky $d_{max} = 22$, důvodem tohoto přerušení byla velikost, jakou strom zabíral v paměti, kdy tato velikost obsazení rostla stejným trendem jako čas konstrukce samotného stromu na grafu 5.1. V celkových naměřených výsledcích lze pozorovat původ tohoto problému, zejména v počtu plných listů C_F, což je důsledkem principu balancování počtu objektů po obou stranách dělící plochy.

V. Havran [13] se taktéž zabýval optimálním nastavením hloubky d_{max}, a optimální hodnotu pro konstrukční algoritmus *SAH* našel v ($d_{max} \approx 16 \pm 2$), pro kterou se výrazně neměnila celková cena k-D stromu při konstrukci, stejně tak jako výkonnostní rozdíly. Při měření jsem tedy nastavil pevnou hodnotu na $d_{max} = 18$, kterou jsem odhadl jako optimální vzhledem k průběhům v grafech a výzkumu V. Havrana.

Zde uvedu stručný přehled naměřených výsledků pro všechny testované scény.

$d_{max} = 18, n_{max} = 1$

Tabulka č. 1

	SpaceMed		*ObjectMed*		*SAH*		*Classic*
Scéna	$T_B[s]$	$T_R[s]$	$T_B[s]$	$T_R[s]$	$T_B[s]$	$T_R[s]$	$T_R[s]$
Bunny	0,390	2,605	2,137	2,933	1,108	0,374	395,720
Robots	1,264	533,930	5,787	15,319	3,869	2,933	1703,265
Dragon	4,836	32,838	10,967	5,601	29,157	1,701	4356,130
Budha	5,975	42,838	12,745	4,461	31,910	0,999	5280,565

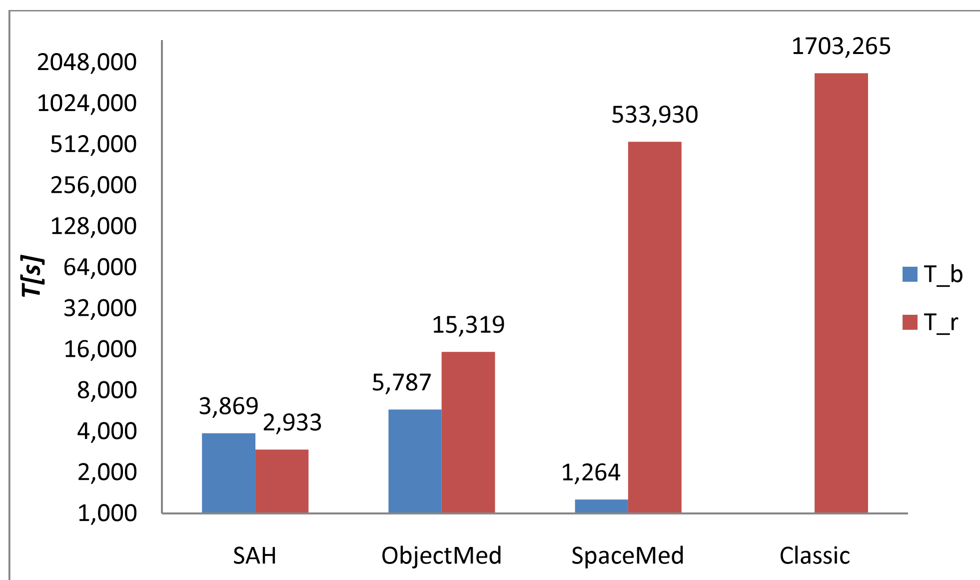
V tabulce č. 1 jsou naměřené výsledky jednotlivých dob. Z těchto výsledků lze vyčíst, že *SAH* se řadí dobou vykreslení ke špičce. A jak jsem zmiňoval, tak druhá scéna svým nerovnoměrným rozložením ukázala přednosti *SAH*, zatímco zbylé metody se podle očekávání propadly se svým výkonem velice hluboko pod cenový model. Avšak existuje způsob, jak by šlo tyto rozdíly částečně

vyrovnat. Změnou maximální hloubky se dá dosáhnout nižších dob, jak už bylo vidět na průběhu grafu 5.1, ale i tak to pro nerovnoměrnou scénu nemusí stále stačit na cenový model, jak je vidět následující tabulce. V tabulce č. 2 jsem změnil hodnotou $d_{max} = 22$ pro scénu *robots*.

Tabulka č. 2

	SpaceMed		*ObjectMed*		*SAH*	
Scéna	$T_B[s]$	$T_R[s]$	$T_B[s]$	$T_R[s]$	$T_B[s]$	$T_R[s]$
Robots	1,576	231,865	25,834	13,447	3,869	2,933

Do tabulky je vložena hodnota *SAH* s $d_{max} = 18$ pouze pro porovnání. Změna hloubky nepřinesla kýžený výkon navíc a to i s nevýhodou, že obě metody měly vyšší paměťovou využití, z tohoto hlediska se tedy jeví prostorový a objektový medián jako velice neefektivní pro nerovnoměrné scény.



Graf 5.3: Doby konstrukce a vykreslování pro scénu *robots.*
T_b doba kostrukce, T_r doba vykreslování.

Graf 5.3 zobrazuje asi nejlépe přednosti heuristiky *SAH*. A to v tomto případě byl implementován pouze konstrukční algoritmus se složitostí $O(N\,log^2\,N)$ a jak je vidět, tak ve scénách pro něj optimálních dosahuje nejlepších časů, jediný problém nastává pouze ve scénách s vysokým počtem rovnoměrně rozložených trojúhelníků, kde konstrukce je poměrně dosti náročná díky obalování celého modelu a následnému konstantnímu seřazování tak vysokého počtu kandidátních ploch.

Výsledky ze všech testů a s naměřenýma hodnota, které byly popsané v metodice lze nalézt v příloze č. 1.

Závěr

V práci jsem rozebral různé metody a algoritmy pro sledování paprsku, a ať už se jednalo o klasický základní Whittedův rekurzivní přístup nebo přístup přinášející nové rozšíření v podobě měkkých stínů, hloubky ostrosti a dalších efektů, tedy distribuované sledování paprsku. Nebo algoritmy globálního osvětlení, které se v dnešní době používají téměř v každé aplikaci pro vytváření realistických scén. Všechny tyto algoritmy od počátku musí řešit výkonnostní problémy. Jak poukázal Whitted v [24], tak vrhání paprsků zabírá ten největší podíl v zabraném výkonu.

Proto je důležité se také zabývat technikami, které přináší způsoby jak snížit počet vržených paprsků a jak omezit počet testů pro nalezení nejbližšího průsečíku. Zde se časem prokázal jako nejefektivnější způsob pro toto urychlení dělit scénu do prostorových struktur.

K-D stromy se časem prokázaly jako velice efektivní struktura a tak bylo jen otázkou času, kdy se začala využívat i pro sledování paprsku. Do nedávna se k-D strom konstruoval jednoduchými metodami a fungoval velice efektivně v 2-D prostoru, avšak s nasazením do metody sledování paprsku se objevily komplikace, pro které neexistovalo optimální řešení.

S růstem výkonu procesorů zároveň přicházely i větší požadavky na složitost scény, která se odrážela ve výsledných dobách konstrukce a vykreslování. Proto bylo třeba přijít s lepším řešením, výzkum konstrukčních algoritmů přinesl na svět speciální algoritmus založený na pravděpodobnosti a geometrickém rozložení scény. Tento algoritmus předčil očekávání svou rychlostí konstrukce a vykreslování. Díky tomu se nyní řadí před nejrychlejší doposud známé struktury pro urychlení sledování paprsku.

Z naměřených výsledků srovnávajících konstrukční algoritmy k-D stromu, prostorový medián, objektový medián a *SAH* dopadl nejlépe právě *SAH*. Naměřené doby konstrukce odpovídají časové složitosti, objektový medián se *SAH* dosahují řádově stejných časů, to plyne i z toho, že oba spadají pod $O(N\ log^2\ N)$.

Přínos konstrukčního algoritmu k-D stromu *SAH* do metody sledování paprsku je zřejmý. Tato struktura dosahuje zrychlení několika řádů v porovnání s klasickým naivním přístupem a ve druhé testovací scéně dokonce i ve srovnání s prostorovým mediánem. Objektový medián svou rychlostí konstrukce a vykreslování je na tom velice dobře a řadí se za *SAH*, avšak nevýhody jaké tento konstrukční algoritmus přináší, mu neumožňují širší využití pro sledování paprsku s porovnáním ostatních metod. V předpokladu použití toho algoritmu pro mnohem složitější scény než jsem testoval, by jeho paměťová náročnost rostla velice rychle nahoru. U měřených hodnot N_C u scény *dragon* a *budha*, tedy celkový počet objektů ve všech listech, je vidět řadový rozdíl tohoto počtu při porovnání *SAH* a prostorového mediánu.

V případě implementace *SAH* algoritmu se složitostí $O(N\ log\ N)$, lze dosáhnout ještě dalšího urychlení konstrukce. Toto téma bych viděl jako možnost pro pokračování studia struktury k-D stromu. A jak zmínil V. Havran a I. Wald v [22], tak zde jsou i další možnosti, protože doposud používaná lokální hladová aproximace pro odhad lokální ceny polohy řezací plochy byla několikrát porovnávána s jinými aproximacemi, avšak doposud žádná z nich nepřinesla lepší výsledky.

Literatura

[1] *3D Object Intersection Home Page*. Maintained by T. Moller, E. Haines and P. Foscari, 2000.
Dostupné na URL: http://www.realtimerendering.com/int/

[2] Appel, A.: *Some techniques for shading machine renderings of solids*, AFIPS 1968 Spring Joint Comptr. Conf., 37-45, 1968.

[3] Arvo, J.: *Linear-time voxel walking for octrees*, Ray Tracing News, E-Mail edition, 26.březen, 1988.

[4] Bentley J.: *Multidimensional binary search trees used for associative searching,* Communications of the *ACM*, 18:509–517, 1975.

[5] Blinn, J. F.: *Models of light reflection for computer synthesized pictures*, SIGGRAPH 1977, San Jose, Calif., pp. 192-198, 1977.

[6] Bourke, P.: *Geometry, Surfaces, Curves, Polyhedra*.
Dostupné na URL: http://local.wasp.uwa.edu.au/~pbourke/geometry/.

[7] Bui-Tuong Phong: *Illumination for computer generated images*, Comm. ACM 18, 311-317, 1975.

[8] Cook, R.L., Porter, T., Carepenter, L..: *Distributed ray tracing*, Computer Graphics, 137-146, 1984.

[9] Goldsmith J., Salmon J.: *Automatic creation of object hierarchies for ray tracing*. IEEE Computer Graphics and Applications, 7(5):14–20, Květen 1987.

[10] Hugo Elias: *Radiosity*, 2000
Dostupné na URL: http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm

[11] Haines E.: *Standard Procedural Databases.*
Dostupné na URL: http://tog.acm.org/resources/SPD/

[12] Havel Jiří, Herout Adam: *Yet Faster Ray-Triangle Intersection (Using SSE4)*, IEEE TVCG, 2009.

[13] Havran Vlastimil: *Heuristic Ray Shooting Algorithms*, PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[14] Jensen, H.W., Arvo, J., Dutre, P., Keller, A., Owen, A., Pharr, M., Shirley, P.: *Monte Carlo ray tracing*, ACM SIGGRAPH 2003 Course Notes. ACM, 2003.

[15] James T. Kajiya.: *The Rendering Equation*, In David C.Evans and Russell J. Athay, editors, Computer Graphics (Proceedings of SIGGRAPH 86), Volume 20, strany 143–150, 1986.

[16] Kirk David: *Graphics Gems III*, Academic Press 1992.

[17] Kaplan M.: *The Use of Spatial Coherence in Ray Tracing*, ACM SIGGRAPH'85 Course Notes 11, strany 22–26, Červenec 1985.

[18] MacDonald J. D. and Booth K. S.: *Heuristics for ray tracing using space subdivision*, Visual Computer, 6(6):153–65, 1990.

[19] Stanford University Computer Graphics Laboratory: *The Stanford 3D Scanning Repository*
Dostupné na URL: http://graphics.stanford.edu/data/3Dscanrep/

[20] Stone Lawrence: *Theory of Optimal Search.* Academic Press, 1975.

[21] Wald Ingo: *Interactive Global Illumination*, PhD thesis, Computer Graphics Group, Saarland University in Saarbrucken, 2004.

[22] Wald Ingo, Havran Vlastimil: *On building fast kd-Trees for Ray Tracing and on doing that in O(N log N),* Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, strany 61-69, 2006.

[23] Wikipedia: *Global illumination*, 2008
Dostupné na URL: http://en.wikipedia.org/wiki/Global_illumination

[24] Whitted, T.: *An Improved Illumination Model for Shaded Display*, CACM, 343–349, 1980.

Seznam příloh

Příloha 1.: Naměřená data
Příloha 2.: Dokumentace k programu
Příloha 3.: DVD

Příloha 1

Scéna: *Bunny; Data pro grafy 5.1 a 5.2*

	SpaceMed		*ObjectMed*		*SAH*		*Classic*
d_{max}	$T_B[s]$	$T_R[s]$	$T_B[s]$	$T_R[s]$	$T_B[s]$	$T_R[s]$	$T_R[s]$
10	0,203	79,529	0,281	7,379	0,453	5,726	275,778
11	0,203	61,621	0,312	5,101	0,483	2,870	
12	0,203	49,687	0,343	3,417	0,515	1,685	
13	0,203	32,152	0,406	2,730	0,577	1,600	
14	0,234	17,519	0,499	2,402	0,639	0,686	
15	0,250	9,750	0,609	2,294	0,671	0,515	
16	0,265	5,700	0,780	1,966	0,687	0,390	
17	0,281	2,698	1,460	1,732	0,748	0,328	
18	0,296	1,934	1,529	1,482	0,843	0,266	
19	0,312	1,217	2,278	1,279	0,921	0,250	
20	0,344	0,765	3,370	1,233	1,140	0,234	
21	0,359	0,624	5,148	1,107	1,170	0,234	
22	0,390	0,452	7,769	1,530	1,342	0,218	
23	0,390	0,328	-	-	1,482	0,218	
24	0,421	0,280	-	-	1,779	0,218	
25	0,468	0,250	-	-	2,137	0,218	
26	0,499	0,218	-	-	2,371	0,218	

Měřené hodnoty:

- N Celkový počet objektů
- L Počet světel
- R Počet vystřelených paprsků
- T_R Doba vykreslování
- T_B Doba konstrukce k-D stromu
- K Celkový počet kolizí s objekty
- C Počet buněk k-D stromu
- C_F Počet plných listů
- C_E Počet prázdných listů
- N_C Součet objektů ve všech listech
- N_P Průměrný počet objektů na list

Scéna: *Bunny*

	SpaceMed	*ObjectMed*	*SAH*	*Classic*
N	69454	69454	69454	69454
L	3	3	3	3
R	114846	114846	114846	114846
T_R [s]	2,605	2,933	0,374	395,720
T_B [s]	0,390	2,137	1,108	-
K	44471823	42850075	4289446	7798581860
C	1211	518023	19021	-
C_F	287	253096	7867	-
C_E	319	5916	2892	-
N_C	82925	1027373	156997	-
N_F	289	4	20	-

Scéna: *Robots*

	SpaceMed	*ObjectMed*	*SAH*	*Classic*
N	241104	241104	241104	241104
L	3	3	3	3
R	131866	131866	131866	131866
T_R [s]	533,930	15,319	2,933	1703,265
T_B [s]	1,264	5,787	3,869	-
K	9016586639	202973996	44263637	55431530910
C	727	520965	27144	-
C_F	158	257766	11714	-
C_E	206	2717	3320	-
N_C	259048	2257984	404413	-
N_F	1640	9	35	-

Scéna: *Dragon*

	SpaceMed	*ObjectMed*	*SAH*	*Classic*
N	871417	871417	871417	871417
L	3	3	3	3
R	229910	229910	229910	229910
T_R [s]	32,838	5,601	1,701	4356,130
T_B [s]	4,836	10,967	29,157	-
K	441864775	79116507	26333102	2,00347E+11
C	1487	524253	29925	-
C_F	396	261927	13112	-
C_E	348	200	3600	-
N_C	927788	3184422	1360637	-
N_F	2343	12	104	-

Scéna: *Budha*

	SpaceMed	*ObjectMed*	*SAH*	*Classic*
N	1087719	1087719	1087719	1087719
L	3	3	3	3
R	98866	98866	98866	98866
T_R [s]	42,838	4,461	0,999	5280,565
T_B [s]	5,975	12,745	31,910	-
K	590833569	63768530	14426061	1,07538E+11
C	681	524241	28033	-
C_F	180	261959	12445	-
C_E	161	162	2930	-
N_C	1134285	3716267	1586100	-
N_F	6302	14	127	-

Příloha 2

Dokumentace k přiloženému programu a zdrojovým souborům na DVD.

Aplikace gRay
Experimentální raytracer je napsán v jazyce C++ a ve své finální verzi funguje pouze pod operačním systémem Windows, a to kvůli standardnímu vykreslování přes winapi.

Program pracuje s adresářovými strukturami */meshes/* a */scene/*. Do adresáře *meshes* lze uložit libovolný počet modelových souborů **.obj*, které aplikace následně po spuštění načte do paměti. Hodnoty, které dokáže program načítat z těchto modelových souborů, patří:

- *v* 0.0 0.0 0.0 pozice vertexů
- *f* 1 2 3 rozmístění faců

Kde následný výpočet normál probíhá v kódu programu.

Druhý adresář *scene* obsahuje pouze jeden soubor s názvem *scene.txt*. V tomto souboru se definují další objekty, které aplikace dokáže vykreslit. Mezi tyto objekty patří koule, kvádr a plocha. Pro každý z těchto objektů se musí zároveň v souboru definovat i jejich materiál. Dále je nutné definovat pozici kamery ve scéně a volitelně přidat neomezený počet světel. Problémem zůstává pouze to, že nelze měnit barvu vykreslovaných trojúhelníku jinak než změnou kódu a taktéž nelze objekty tímto souborem transformovat (tedy měnit rotaci, velikost a posunutí).

Pro samotný výběr metod vykreslování, tedy buď klasickým způsobem, nebo využitím k-D stromu se všemi třemi způsoby jeho konstrukce je nutný opět zásah do kódu a poté program zkompilovat.

Na DVD je spolu se zdrojovými soubory vložen vzorový formát souboru se scénou, s tím i všechny zdrojové modely sloužící jako testovací scény.

Změna parametrů
Veškeré změny parametrů, které aplikace umožňuje měnit, probíhají v hlavičkovém souboru *scene.h.* Proměnné jsou ve formě preprocesorových direktiv.

Úryvek ze souboru *scene.h*

Kód	Popis
// global settings	
#define SCRWIDTH 640	Změna šířky okna.
#define SCRHEIGHT 480	Změna výšky okna.
#define SCENE_FILE "scene.txt"	Soubor, který se bude načítat z adresáře *scene.*
// method settings	
//#define KDTREE 1	Při zrušení komentáře lze použít k-D strom.
//#define SPACE_MEDIAN 1	Umožní použít metodu prostorového mediánu
//#define OBJECT_MEDIAN 1	Konstrukce objektovým mediánem.
//#define SAH_HEURISTIC 1	Použití heuristiky *SAH.*
//#define ROUNDROBIN 1	Sekvenční výběr orientace dělících os
// k-D tree settings	
#define MAX_TREE_DEPTH 18	Maximální hloubka stromu d_{max}
#define MAX_PER_LEAF 1	Maximální počet objektů v listu n_{max}
#define MAX_DEPTH 50	Velikost zásobníku pro algoritmus průchodu paprskem

Kód	Popis
// ray-tracing settings	
//#define PHONG_SHADING 1	Použití Phongova modle namísto Blinn-Phongova
#define FLIP_NORMALS 1	Při špatně definovaných normálách je invertuje
#define MAX_LIGHTS 10	Maximální počet světel
#define MAX_OJECTS 1500000	Maximální počet objektů
#define MAX_DISTANCE 10000	Vykreslovací vzdálenost
#define TRACE_DEPTH 3	Hloubka rekurze
#define BG_COLOR Color(0.4f, 0.4f, 0.7f)	Barva pozadí v případě žádné kolize

Parametry jsou pojmenovány intuitivně, aby bylo umožněno co nejrychlejší nalezení a následná změna. Dále jsou rozděleny do čtyř skupin, podle toho k čemu přesně se tyto parametry vážou.

Program automaticky po vykreslení vytváří soubor *info.txt* se záznamy výstupních testovaných parametrů. Po zavření okna se automaticky vygeneruje obrázek vykresleného snímku ve formátu *output_XX.tga*.

Příloha 3

DVD se zdrojovými soubory a elektronickou verzí této práce.