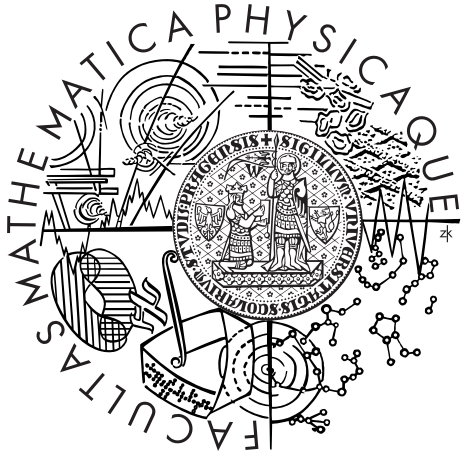
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Ivan Kuckir

Datové struktury pro zobrazování nepolygonální geometrie

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Ing. Jaroslav Křivánek, Ph.D.
Studijní program: Programování
Studijní obor: Informatika

Praha 2013

Chtěl bych poděkovat svému vedoucímu, který mi toto téma pomohl vybrat a směroval mě během mé práce. Dále RNDr. Josefu Pelikánovi, který mě motivoval pro počítačovou grafiku, a všem dalším skvělým lidem, kteří mě bakalářským studiem provázeli.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Datové struktury pro zobrazování nepolygonální geometrie

Autor: Ivan Kuckir

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Ing. Jaroslav Křivánek, Ph.D.

Abstrakt: V moderní 3D grafice se nejčastěji používají scény složené z trojúhelníků a zobrazovací metody založené na sledování paprsků. Pro urychlení hledání průsečíků paprsku se scénou se používají hierarchické datové struktury, tzv. akcelerační stromy. Při testování těch nejlepších současných metod s nepolygonální geometrií (konkrétně úsečkami) jsme zjistili, že v mnoha případech nedokáží postavit efektivní strom.

Tato práce si dává za cíl celý problém řádně matematicky formulovat. Díky tomu se téma stává průhlednější a lze vidět i nedostatky současných metod, na které zatím nikdo neupozornil. Výsledkem je i algoritmus, který zobecňuje všechny současné metody, není závislý na podobě geometrie a přímo ukazuje směr vylepšení.

Klíčová slova: Počítačová grafika, sledování paprsku, datové struktury.

Title: Data structures for rendering non-polygonal geometry

Author: Ivan Kuckir

Department: Department of Software and Computer Science Education

Supervisor: Ing. Jaroslav Křivánek, Ph.D.

Abstract: In modern 3D graphics, scenes made of triangles are usually used, combined with methods based on ray tracing. Hierarchical data structures, called accelerating trees, are often used to speed up the search for intersection between ray and the scene. When testing the best current methods with non-polygonal geometry (line segments), we have found out that those structures cannot build an effective tree in many cases.

The aim of this work is to formulate the problem mathematically. Thanks to this, the whole subject becomes more transparent and we can see the shortcomings of current methods, which have not yet been pointed out. At the result, we develop an algorithm which generalizes all current methods, which is not dependent on geometry and directly shows the space for improvement.

Keywords: Computer graphics, rendering, ray tracing, data structures.

Obsah

Úvod

Jedním z odvětví počítačové grafiky je fotorealistická grafika. Její cílem je automaticky generovat obrázky či videa, která budou svým obrazem odpovídat skutečnému světu. Lze se s ní setkat ve filmové produkci, v architektuře, v reklamách a na dalších místech. Běžné postupy jsou obvykle založeny na simulaci šíření paprsků světla ve scéně. Nejznámější jsou Ray tracing [1] či Path tracing [2].

Složité geometrické scény se většinou modelují pomocí jednodušších dílků, které se označují jako *primitiva*. Obvykle se používají trojúhelníky, avšak lze použít koule, válce, kvádry či jiné jednoduché objekty.

Vykreslovací algoritmy vyžadují pro *paprsek* (polopřímku) najít průsečík se scénou, který je nejblíže počátku paprsku, případně zjistit, že paprsek scénu neprotíná. Kvůli tomu je potřeba testovat průnik paprsku s jednotlivými primitivy a z případných průsečíků vybrat ten nejbližší.

Primitiv mohou být ve scéně tisíce, miliony, miliardy. Testování kolize s každým z nich, po jednom, by bylo časově velmi náročné. Pro efektivnější testování je užitečné primitiva určitým způsobem strukturovat, uspořádat.

V praxi se pro to nejvíce osvědčily stromové datové struktury, např. oktanové stromy, BVH stromy či KD stromy. Ve stromových strukturách každý uzel stromu odpovídá nějaké podmnožině z prohledáváného celku. Kořen stromu odpovídá celé množině. Při průchodu stromem vylučujeme části, ve kterých hledané řešení určitě neleží, a tím zmenšujeme prohledávanou množinu. Nakonec dojdeme do listu, ke kterému se váže jen malý počet prvků, a ty už můžeme otestovat po jednom.

Hledání prvků ve stromových strukturách obvykle mívá časovou složitost logaritmickou vzhledem k velikosti prohledávané množiny. Postavit efektivní strom však není tak jednoduché, obzvlášť v případě primitiv 3D scény, která mohou mít různé geometrické vlastnosti.

Tato práce byla motivovaná faktem, že současné metody stavby stromů si nedokáží poradit s nepravidelnou geometrií. Např. vlasy, ať už je modelujeme pomocí dlouhých úzkých válců či trojúhelníků, bývají rozmístěny v prostoru velmi hustě, navzájem se protínají a kříží. Podobné vlastnosti mají i modely chlupů zvířat, modely rostlin (vysoká tráva, jehličí) či architektonické modely s velkým počtem dlouhých rovných ploch.

Dalším případem nepravidelné geometrie jsou tzv. *photon beams* [3]. Jedná se o způsob simulace šíření světla v médiu, které není úplně průhledné. Do scény se přidají úsečeky, znázorňující cesty šíření světla ze svět. zdrojů. Pro paprsek z kamery je třeba rozhodnout, ke kterým „beamům“ (úsečkám) se příblíží na předem danou vzdálenost. Tyto úsečky rovněž bývají velmi dlouhé, husté a nepravidelné.

Právě s výše zmíněnými případy mají současné metody největší problémy. Nejen, že nedokáží postavit dost efektivní strom, v některých situacích vedou dokonce k jednouzlovým stromům s lineární časovou složitostí.

Na toto vše se důkladněji podíváme v první polovině práce. Ve té druhé si kromě teorie představíme i algoritmus, který umožní stavět efektivní stromy nejen pro nepravidelou geometrii, ale také vylepší kvalitu stromů pro běžné scény.

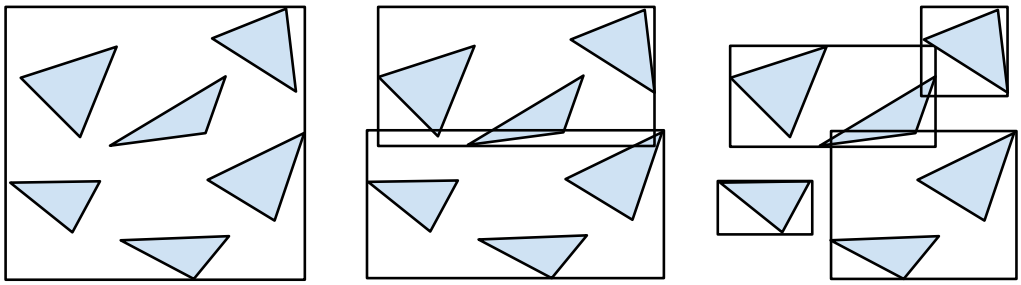
1. Současné techniky

V úvodu této práce se letmo podívejme na nejčastěji používané datové strutkury a metody jejich stavby. Detailnější informace lze najít např. v knize *Physically Based Rendering* [7].

Většina stromových struktur při rozdělování prostoru používá osově zarovnané ohraničující kvádry, dále jen *kvádr* nebo *AABB* (z anglického Axis-aligned Bounding Box).

Je zvykem stromové struktury dělit na ty, které rozdělují primitiva, a ty, které rozdělují prostor. Většina stromových struktur se staví "shora dolů"(top-down). Začíná se od kořene stromu, ke kterému se postupně přidávají další uzly. Stromy lze stavět i "zdola nahoru"(bottom-up), kdy se listy stromu postupně spojují do podstromů, až nakonec vznikne jeden strom. Současné bottom-up algoritmy však nejsou příliš efektivní.

1.1 BVH strom



Obrázek 1.1: Hladiny BVH stromu

BVH strom je většinou binární strom, kde každý uzel má buď dva potomky, nebo žádného. Každé primitivum patří do jednoho listu stromu. Každému listu je přiřazen kvádr, který ohraničuje všechna primitiva v něm. Každému vnitřnímu uzlu je přiřazen kvádr, který ohraničuje kvádry obou potomků. Takto popsaný BVH strom patří ke strukturám, které dělí objekty.

1.1.1 Stavba

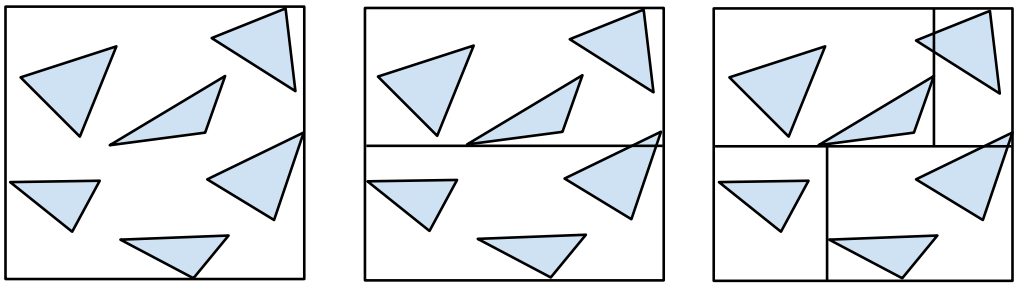
Lépe si BVH strom představíme při pohledu na algoritmus jeho stavby. Na začátku vyrobíme kořen stromu a k němu přiřadíme kvádr, který ohraničuje všechna primitiva. Následně primitiva rozdělíme do dvou skupin, které by měly mít co nejmenší ohraničující kvádry, a pro tyto dvě skupiny stejným způsobem postavíme pravý a levý podstrom.

1.1.2 Prohledávání

Při hledání průsečíku paprsku postupujeme rekurzivně. Nejdříve zkontrolujeme, zda paprsek protíná kvádr kořene. Pokud ano, podíváme se, zda protíná kvádry

synů. Pokud opět ano, hledáme průsečík stejným způsobem na nižších úrovních. Pokud jsme v listu, projdeme všechna primitiva, co k němu patří. Pokud při hledání najdeme průsečík s primitivem v jednom podstromu nějakého uzlu, stále může existovat průsečík ve druhém podstromu daného uzlu, který je bliž. Proto je obecně potřeba prohledat oba podstromy.

1.2 KD strom



Obrázek 1.2: Hladiny KD stromu

KD storm je rovněž binární strom, kde uzly mají buď dva potomky, nebo žádného. Ke každému uzlu se váže kvádr, který představuje nějaký kus prostoru, nezávisle na poloze primitiv. Dále se ke vnitřním uzlům váže rovina, která rozděluje kvádr uzlu na dva kvádry, které patří dvěma synům. K listům uzlu patří všechna primitiva, která protínají jeho kvádr. Narozdíl od BVH, KD strom může obsahovat listy, ke kterým nejsou přiřazena žádná primitiva. Primitivum také nemusí patřit pouze k jednomu listu, obvykle prochází více listy. KD strom patří ke strukturám, rozdělujícím prostor.

1.2.1 Stavba

Při stavbě KD stromu rekurzivně rozdělujeme prostor. Ke kořenu přiřadíme AABB primitiv, všechna primitiva a spustíme rekurzivní stavbu. Při stavbě podstromu pro nějaký uzel nejdříve zvolíme polohu dělící roviny. Z kvádru uzlu a dělící roviny spočítáme dva kvádry pro syny, dvě množiny primitiv, která tyto kvádry protínají, a s pustíme stavbu podstromů na synech.

1.2.2 Prohledávání

Při hledání průsečíku paprsku s primitivy v KD stromě také postupujeme rekurzivně. Nejdříve zkontrolujeme průsečík s kvádrem kořene. Pokud jsme ve vnitřním uzlu, zkontrolujeme průsečík s dělící rovinou a podle toho rozhodneme, zda paprsek protíná pravého či levého syna. Pokud protíná oba, navštívíme nejdřív toho, který je blíž počátku paprsku. Pokud jsme v listu, zkontrolujeme průsečík se všemi primitivy, která k němu patří. Zde si však musíme dát pozor, aby nalezený průsečík ležel uvnitř kvádru listu, jinak ho nepočítáme. Díky tomu však můžeme při nalezení průsečíku v jednom podstromu vynechat druhý podstrom, jelikož první podstrom se nacházel blíž a kvádry obou podstromů jsou disjunktní.

1.3 Několik implementačních detailů

Výhodou BVH stromů je, že se kvádry potomků rychle zmenšují. Pokud paprsek zasahuje kvádr uzlu, ale nezasahuje kvádry potomků, můžeme hledání ukončit. To neplatí o KD stromech, kde paprsek zasahující kvádr uzlu vždy zasahuje kvádr aspoň jednoho potomka - hledání musí pokračovat až do listu. Výhoda KD stromů je naopak ta, že při přechodu od uzlu k synům stačí otestovat průsečík s 1 rovinou, narozdíl od BVH, kde je potřeba otestovat celé 2 kvádry (12 rovin). Díky nízké ceně traverzace u KD stromů si můžeme dovolit stavět rozsáhle KD stromy, zatímco BVH stromy bývají kvůli drahé traverzaci mnohem mělčí.

1.4 Dělící strategie

Při stavbě výše popsaných struktur potřebujeme řešit dva problémy: jak rozdělit uzel na dvě části a kdy ukončit stavbu.

1.4.1 Efektivní rozdělení

U BVH stromů si lze primitiva seřadit podle středů jejich AABB, pro každou souřadnici zvlášť. Pak je můžeme rozdělit do dvou skupin na ty, které mají střed pod hranicí mediánu a na ty, které mají střed nad ní. To provedeme pro každou ze tří os a vybereme to rozdělení, ve kterém součet povrchů AABB pravé a levé skupiny primitiv je co nejmenší. Tento postup vede k dost vyváženým stromům.

Při rozdělování primitiv v KD stromu máme nekonečně mnoho možností, kam umístit dělící rovinu. Šikovným řešením je umístit ji uprostřed nejdelší strany kvádru. Užitečné může být také „ořezávání prázdna“, kdy jeden z výsledných kvádrů neprotínají žádná primitiva. Pro tento kvádr se vyrobí list bez primitiv, který umožní hledání rychle ukončit.

1.4.2 Efektivní ukončení stavby

Můžeme zvolit různé strategie i pro ukončení stavby podstromů.

Jednoduchým způsobem je omezení hloubky. Jakmile hloubka větve stromu přesáhne určitou mez, vyrobí se zde list. Tento postup vychází z intuice, že příliš hluboké stromy odpovídají zdlouhavému hledání. Mohou se však vytvářet listy s obrovským počtem primitiv.

Další možností je stanovit hranici počtu primitiv. Jakmile se má stavět podstrom s počtem primitiv pod touto hranicí, vyrobíme list. Tento postup se snaží zajistit, aby se při návštěvě listu nemuselo testovat hodně primitiv. Nebere však ohled na hloubku stromu.

Dalším postupem je stanovit hranici pro velikost kvádru u listu. Jakmile je kvádr uzlu dost malý, vyrobíme list. To nám může zajistit, že listy budou mít malé kvádry. Dle intuice, malé kvádry mají menší šanci zásahu paprskem. Tento postup nefunguje dobře u BVH stromů, kde primitiva mohou mít velké rozměry a musí být obsažena ve kvádru listů.

1.5 Povrchová heuristika

Malá revoluce ve stavbě akceleračních stromů nastala v roce 1990, kdy MacDonald a Booth zveřejnili svůj článek o povrchové heuristice [4].

Povrchová heuristika, dále SAH (z anglického Surface Area Heuristics) je způsob, jak stavět efektivní akcelerační stromy. Je celkem pěkně definovaná a my si ji teď zkusíme trochu přiblížit.

SAH vychází z matematické věty, která říká: máme-li dvě konvexní množiny $A, B \subseteq \mathbb{R}^3$, $B \subseteq A$, a náhodnou přímku, která protíná A, potom pravděpodobnost, že protíná i B, se rovná $povrch(B)/povrch(A)$. K této větě se ještě vrátíme v dalších kapitolách.

Označme si čas testu průsečíku s primitivem jako p a čas přechodu mezi dvěma uzly v našem stromě jako t.

Představme si situaci, kdy se při stavbě stromu nacházíme v uzlu, máme jeho kvádr q a množinu n primitiv. Pokud bychom zde vyrobili list, čas jeho průchodu bude $n \cdot p$. Řekněme, že primitiva rozdělíme na dva kvádry q_1, q_2, ke kterým by patřilo n_1 a n_2 primitiv. Pravděpodobnosti zásahu kvádrů jsou $povrch(q_1)/povrch(q)$ a $povrch(q_2)/povrch(q)$. Cena průchodu tohoto uzlu by byla

$$c = t + \frac{povrch(q_1)}{povrch(q)} \cdot n_1 \cdot p + \frac{povrch(q_2)}{povrch(q)} \cdot n_2 \cdot p$$

Dostali jsme se k jakési cenové funkci, která by měla ohodnotit kvalitu rozdělení. Nyní můžeme při rozdělování uzlu ohodnotit každé naše rozdělení a vybrat to, které má nejnižší cenu. To pak můžeme porovnat s cenou ponechání listu a rozhodnout, zda vyrobíme list.

Při hledání rozdělení u BVH stromů se primitiva obvykle seřadí dle středů jejich AABB. Pak se zkouší posouvat dělící mez mezi levou / pravou skupinou na všech $n - 1$ pozic. Pro každou skupinu se přepočítá AABB levé / pravé části a hodnota cenové funkce. To se provede pro každou osu a vybere se rozdělení s nejmenší cenou.

Při rozdělování v KD stromech je situace složitější, jelikož je nekonečně mnoho míst, kam umístit dělící rovinu. Lze však ukázat, že výše uvedená cenová funkce má u KD stromu extrémy v případech, kdy se dělící rovina „dotýká“ nějakého primitiva. Proto se vyzkouší a prozkoumá pouze těchto několik pozic dělící roviny ($2 \cdot n$ na každé ose). Tento postup automaticky zahrnuje i výše zmíněné *ořezávání prázdna*.

2. Problémové případy

Ačkoli se předchozí techniky v čele s povrchovou heuristikou mohou zdát ideální, není tomu tak. Je spousta případů, kdy se tyto metody chovají neefektivně.

Povrchová heuristika se snaží minimalizovat hodnotu cenové funkce v každém bodě stavby stromu. Zdá se, že se vždy vybírá to nejlepší rozdělení a stavba podstromů se ukončuje v těch nejlepších místech. Bohužel tomu tak není, jak si ukážeme v příkladech níže. Je také pravda, že povrchova heuristika dnes narazila na hranici efektivity a zdá se, že stromy již nelze stavět lépe. My si v příštích kapitolách tuto hranici odstraníme.

2.1 Několik ukázek

Ukažme si problém na trochu extrémním případě. Mějme n primitiv, jejichž ohraničující kvádry se rovnají.

Zkusme pro tato primitiva postavit BVH strom. Všimněme si, že ať strom stavíme jakkoli, AABB všech uzlů se vždy rovnají AABB celku. Kvádry se s rekurzí nezmenšují. Povrch syna ku povrchu předka je vždy 1.

Pokud vyrobíme list, jeho cena dle SAH bude $n \cdot p$. Pokud primitiva jakkoli rozdělíme do dvou stejně velkých skupin, pak bude cena uzlu

$$c = t + n/2 \cdot p + n/2 \cdot p$$

List by tedy znamenal kontrolu průsečíku se všemi primitivy, vyrobení synů by znamenalo to samé + navíc kontrolu průsečíku se dvěma kvádry (t). SAH nám tedy poradí vyrobit list.

Zkusme postavit pro primitiva KD strom, také dle SAH. Všimněme si, že ať umístíme dělící rovinu kamkoli, každé primitivum vždy zasahuje do obou částí. Pokud vyrobíme list, jeho cena bude opět $n \cdot p$. Pokud si označíme povrch celku jako sa a povrch dvou nových kvádrů jako sa_1, sa_2, potom je cena rozdělení

$$c = t + (sa_1/sa) \cdot n \cdot p + (sa_2/sa) \cdot n \cdot p$$

Pro povrchy kvádrů však platí, že $sa_1 + sa_2 \geq sa$. List by znamenal kontrolu průsečíku se všemi primitivy, vyrobení synů by znamenalo to samé jednou až dvakrát, jelikož všechna primitiva zasahují do obou synů. SAH nám tedy opět poradí vyrobit list.

Vidíme, že SAH v tomto extrémním případě vždy vyrobí jednouzlový strom, který má lineární časovou složitost hledání průsečíku.

Objevuje se i otázka, zda pro tento extrémní případ vůbec lze sestavit nějaký strom, který by se v průměru procházel rychleji. Vidíme že BHV strom určitě ne. Pro libovolný BVH strom, paprsek, protínající AABB kořene zároveň protíná AABB všech listů, což odpovídá kontrole průsečíků se všemi primitivy. U KD stromu by to ale možná mohlo jít, avšak nevíme, jak to provést.

S výše popsaným případem se v praxi nejspíše nesetkáme. Přesto se však můžeme setkat s případy, kdy primitiva mají stále dost velké ohraničující kvádry, které se intenzivně překrývají. Současné techniky, především SAH, pro ně staví velmi mělké a neefektivní stromy.

2.2 Řešení zjemněním

Vraťme se zpět k našemu extrémnímu případu. Zkusme každé naše primitivum nějak rozdělit na dvě primtiva. Tím se počet primitiv zdvojnásobí. Většinou to také způsobí, že se AABB každého primitiva výrazně zmenší a nebudou se tolik překrývat. Cena výroby listu se zdvojnásobí. Pokud primitiva šikovně rozdělíme do dvou skupin, budou AABB těchto skupin nejspíše menší, než před zjemněním. Je tedy velká šance, že rekurzivní stavba bude pokračovat a postaví se hlubší a efektivnější strom.

Není však jasné, jak moc a jakým způsobem máme primitiva zjemňovat. Malé zjemnění může vést k neefektivním stromům. Přílišné zjemnění může vést k velmi hlubokým a také neefektivním stromům. Tomuto problému se věnovalo několik článků z poslední doby.

Zjemňování primitiv se věnovali ve svém článku Manfred Ernst a Günther Greiner [5]. Jejich řešení spočívalo v tom, že se nejdříve zpracují primitiva tak, aby povrch jejich AABB byl pod nějakou hranicí sa_{max}. Z těchto drobných primitiv se potom staví BVH běžným způsobem. Není však úplně jasné, jakých hodnot má nabývat konstanta sa_{max}.

2.3 Řešení dělením prostoru v BVH

Dalších vylepšení pro BVH se snaží dosáhnout článek Spatial Splits in Bounding Volume Hierarchies [6]. Autoři se snaží vylepšit výkon BVH pro primitiva s velkými AABB, která se navzájem intenzivně překrývají.

Jak víme, standartní BVH dodržuje pravidlo, že každé primitivum se vyskytuje právě v jednom listu stromu. Autoři představují novou strukturu SBVH (Spatial BVH), která toto pravidlo vynechává. Primitva se mohou dále „dělit“. Předchozí BVH strom byl strukturou, která dělí objekty, toto nové SBVH dělí prostor. K této myšlence se v budoucnu několikrát vrátíme.

V listech SBVH jsou uloženy reference na primitiva, jedna reference může být ve více listech. Algoritmus funguje následovně. Při stavbě SBVH se v uzlu musí najít 2 kvádry, do kterých se primitiva rozdělí. Tyto 2 kvádry se najdou jednak způsobem běžným pro BVH (s použitím heuristiky, každé primitivum právě v jednom kvádru), jednak způsobem, kdy primitivum může patřit do obou (rozpesáno níže). Z těchto dvou rozdělení se vybere to s menší cenou.

Chceme-li efektivně rozdělit primitiva do dvou skupin a povolíme-li přítomnost jednoho primitiva v obou skupinách, je postup velmi podobný, jako při stavbě KD stromů. Autoři navrhují umísťovat dělící rovinu na hranice AABB primitiv a podle nich vybrat rozdělení s nejmenší cenou. Toto rozdělení má i jednu skvělou výhodu, a to tu, že dva nové kvádry se vůbec nepřekrývají.

Tato metoda pouze přidala BVH stromům kladné vlastnosti KD stromů. Pro náš extrémní případ bude stále vytvářet jednouzlový strom s lineární časovou složitostí, a to kvůli použití SAH.

2.4 Řešení chybováním

Zajímavé řešení výše popsaných situací bylo zmíněno v knize Physically Based Rendering [7]. Autoři zde navrhují, že pokud se dle SAH má vyrobit list a počet jeho primitiv je stále příliš velký, dovolme si „pochybit“, neposlechnout SAH a uzel i přesto rozdělit. Je totiž dost velká šance, že v dalších úrovních stavby se primitiva budou chovat lépe a chybovat již nebude potřeba.

Autoři navrhují při velkém počtu primitiv v listu ho rozdělit uprostřed nejdelší strany jeho AABB a syny dále rozvinout. Při tom by se mělo zaevidovat, že se provedlo pochybení. Počet pochybení na cestě od kořene k listu je také omezen nějakou konstantou (konkrétně autoři navrhují dovolit 3 pochybení).

2.5 Problém povrchové heuristiky

Výše zmíněné chybování již trochu naráží na fakt, že SAH není úplně dokonalá a v některých případech je lepší se jí neřídit. Podívejme se, co vlastně SAH zkoumá.

Povrchová heuristika umožňuje porovnat cenu vytvoření listu s cenou rozdělení uzlu. Mluví o vztahu mezi hodnotou $n \cdot p$ a

$$t + \frac{povrch(q_1)}{povrch(q)} \cdot n_1 \cdot p + \frac{povrch(q_2)}{povrch(q)} \cdot n_2 \cdot p$$

Hodnota vytvoření listu se zdá být v pořádku. Čas průchodu všech primitiv odpovídá jejich počtu krát čas testu jednoho primitiva. Zaměřme se tedy na druhou hodnotu. Ta říká, že po rozdělení primitiv bude čas průchodu roven pravděpodobnosti zásahu levého uzlu krát počet primitiv v levém uzlu + pravděpodobnost zásahu pravého krát počet primitiv v něm. Tato funkce předpokládá, že po rozdělení se vytvoří dva listy! Skutečně tedy říká, že vyrobit list hned je lepší, než vyrobit dva listy o úroveň níže. Funkce však vůbec nepočítá s tím, že oba nové podstromy budou dále rozdělovány a že čas jejich průchodu může být výrazně menší, než $n_1 \cdot p$ a $n_2 \cdot p$.

Dovolil bych si ještě zmínit článek *Fast Insertion-Based Optimization of Bounding Volume Hierarchies* [8]. Autoři zde popisují techniky, jak zlepšit kvalitu již hotového BVH stromu pouze tím, že v něm přeuspořádáme uzly (a přepočítáme AABB vnitřních uzlů). Drobného zlepšení lze dosáhnout i v případě, kdy se při stavbě stromu použila SAH. Popisované metody fungují právě díky tomu, že SAH není úplně dokonalá.

Všechen tento dosavadní výzkum na nás může působit jako pouhé zkoušení a testování jednotlivých postupů, kde každý z nich se chová trochu jinak a je vhodný pro trochu jiné vstupy. My bychom raději dostali nějaký obecný postup, který si poradí se všemi případy. K tomu by nám mohla pomoci řádná matematická formulace naší problematiky.

3. Matematická formulace

V předchozí kapitole jsme si ukázali, že dnes hojně využívaná povrchová heuristika není zcela ideální. Problémové případy lze řešit zjemňováním primitiv nebo porušováním heuristiky. U první metody nevíme, jak moc a jakým způsobem se má zjemňovat. U druhé metody nevíme, kdy a jak často můžeme heuristiku porušovat a nelze předem rozhodnout, zda to bude užitečné.

Algoritmus, který si popíšeme v další kapitole, nebude závislý na tvarech primitiv ani na jejich jemnosti. Nebude v něm ani potřeba „chybovat“. Přesto si dokáže snadno poradit s problémovými případy. Bude navržen obecně a tak, aby bylo zřejmé, že se snaží hledat ten nejlepší strom.

Pro účely algoritmu a lepšího pochopení problému by bylo dobré si celé téma řádně matematicky formulovat. Budeme tak mít větší jistotu v našich krocích a můžeme tím motivovat matematiky k zájmu o toto téma.

3.1 Základní pojmy

Definice. Primitivum *nechť je omezená uzavřená podmnožina* $\mathbb{R}^3$.

Všimněme si, že sjednocení konečně mnoha primitiv je také primitivum.

Definice. Kvádr *je množina trojic reálných čísel* (x, y, z), *splňujících* $a \leq x \leq b$, $c \leq y \leq d$ *a* $e \leq z \leq f$ *pro nějaká* $a, b, c, d, e, f \in \mathbb{R}$.

Definice. *Funkce* SA *: Kvádry* $\to \mathbb{R}$ *nechť přiřazuje každému kvádru velikost jeho povrchu. Konkrétně* $SA(q) = 2 \cdot ((b-a) \cdot (d-c) + (b-a) \cdot (f-e) + (d-c) \cdot (f-e))$ *pro* $a \leq b, c \leq d, e \leq f$, *jinak 0.*

Definice. *Funkci* AABB *: Primitiva* $\to$ *Kvádry si definujme tak, že* $AABB(p)$ *je povrchově nejmenší kvádr, obsahující* p.

3.2 Akcelerační stromy

Definice. Binární kvádrový akcelerační strom *pro neprázdnou konečnou množinu primitiv* P *(dále pouze Akcelerační strom) je dvojice* (V, E) *splňující následující podmínky:*

1. (V, E) je binární strom, každý uzel má buď žádného nebo 2 potomky
2. prvky V jsou dvojice (podmnožina P, kvádr)

$$x \in V \to x = (p, q) \& p \subseteq P \& q \in Kvadry$$

3. kořen je dvojice (P, AABB(P))

$$((p, q) \in V \& |\{w : (w, (p, q)) \in E\}| = 0) \to (p = P \& q = AABB(\cup P))$$

4. kvádry potomků leží v kvádru předka

$$((p,q),(p_1,q_1)) \in E \& ((p,q),(p_2,q_2)) \in E \& (p_1,q_1) \neq (p_2,q_2) \\ \rightarrow q_1 \cup q_2 \subseteq q$$

5. strom má „dobrou strukturu“

$$((p,q),(p_1,q_1)) \in E \& ((p,q),(p_2,q_2)) \in E \& (p_1,q_1) \neq (p_2,q_2) \\ \rightarrow (\cup p_1 \cap q_1) \cup (\cup p_2 \cap q_2) = \cup p \cap q$$

Definice. *O akceleračním stromě řekneme, že je* BVH*, pokud navíc platí:*

- kvádr uzlu je „co nejmenší“

$$(p,q) \in V \rightarrow q = AABB(\cup p \cap q)$$

Definice. *O akceleračním stromě řekneme, že je* KD*, pokud navíc platí:*

- kvádry potomků jsou rozdělením kvádru předka

$$((p,q),(p_1,q_1)) \in E \& ((p,q),(p_2,q_2)) \in E \& (p_1,q_1) \neq (p_2,q_2) \\ \rightarrow q_1 \cup q_2 = q, dim(q_1 \cap q_2) \leq 2$$

Všimněme si, že jsme si definovali BVH strom způsobem SBVH, kdy primitivum může ležet ve více listech. Klasická BVH jsou podmnožinou našich BVH.

Jako nejjednodušší akcelerační strom si můžeme představit jednouzlový strom:

$$(\{(P, AABB(\cup P))\}, \emptyset)$$

Tento strom je BVH a KD zároveň.

3.3 Cenová funkce

Již jsme si řekli, co budeme považovat za akcelerační strom. Nyní by se nám hodilo umět mluvit o efektivitě jednotlivých stromů pro primitiva P. Bylo by dobré si definovat nějakou cenovou funkci, podle které budeme moci jednotlivé stromy porovnávat.

Níže popsaná cenová funkce odpovídá funkci navržené MacDonaldem a Boothem [4]. My si ji však definujeme trochu elegantněji a budeme na ní dále stavět.

Nejdříve si zaveďme konstanty $c_p, c_t \in \mathbb{R}, c_p > 0, c_t > 0$. První bude představovat cenu testování průsečíku s primitivem (primitive test cost), druhá cenu přechodu mezi uzly stromu (traverse cost).

Cenová funkce by měla odpovídat době hledání průsečíků v akceleračním stromě. Zkusme si definovat cenovou funkci rekurzivně podle toho, jak algoritmus skutečně prohledává strom.

Pro akcelerační strom (V, E) a konstanty c_p, c_t definujme funkci $f_0 : V \rightarrow \mathbb{R}$ rekurzivně, podle toho, jak se strom doopravdy prochází.

- je-li (p, q) list, potom

$$f_0((p, q)) = |p| \cdot c_p$$

- je-li (p, q) vnitřní uzel, (p_1, q_1) a (p_2, q_2) jeho potomci, potom

$$f_0((p, q)) = c_t + \frac{SA(q_1)}{SA(q)} \cdot f_0((p_1, q_1)) + \frac{SA(q_2)}{SA(q)} \cdot f_0((p_2, q_2))$$

Připomeňme si, že $\frac{SA(q_1)}{SA(q)}$ odpovídá pravděpodobnosti zásahu kvádru q_1 náhodným paprskem, který zasahuje kvádr q. Defunujme si cenu pro celé stromy.

Definice. *Pro akcelerační strom T s kořenem k stanovme $f : Stromy \to \mathbb{R}$ takto:* $f(T) = f_0(k)$.

Nyní se na problém podívejme z poněkud jiného úhlu. Chceme zjistit všechny průsečíky náhodného paprsku s primitivy. Algoritmus určitě bude muset projít všechny vnitřní uzly i všechny listy stromu, jimiž paprsek prochází. Cenu bychom si mohli definovat jako součet cen přes všechny uzly s koeficienty, které odpovídají pravděpodobnosti zásahu daného uzlu.

Definice. *Pro akcelerační strom T s vnitřními uzly V_i, listy V_l a kořenem $k = (p_k, q_k)$ stanovme $g : Stromy \to \mathbb{R}$ takto:*

$$g(T) = \sum_{(p,q) \in V_i} \frac{SA(q)}{SA(q_k)} \cdot c_t + \sum_{(p,q) \in V_l} \frac{SA(q)}{SA(q_k)} \cdot |p| \cdot c_p$$

Nyní jsme si ukázali dva možné pohledy na věc. Oba se zdají být intuitivní a šikovně odhadovat efektivitu stromu. Teď by se mohly objevit otázky, která funkce to odhaduje lépe? Která je kvalitnější? Dovolte mi nyní vyslovit větu.

Věta 1. *Pro každý akcelerační strom T platí: $f(T) = g(T)$.*

Důkaz. Prozkoumejme funkci f_0

$$f(T) = f_0((p_k, q_k)) = c_t + \frac{SA(q_1)}{SA(q_k)} \cdot f_0((p_1, q_1)) + \frac{SA(q_2)}{SA(q_k)} \cdot f_0((p_2, q_2))$$

Zkusme naši rekurzivní funkci f_0 „rozvinout“ a roznásobit všechny závorky. Počet výskytů c_t bude odpovídat počtu vnitřních uzlů stromu. Počet výskytů c_p bude odpovídat počtu listů. Každý výskyt c_p bude vedle sebe mít počet primitiv v listu: $|p|$. U každého členu budou koeficienty, značící poměr ploch dvojic kvádrů, ležících na cestě od daného uzlu ke kořenu. Jsou-li kvádry na cestě ke kořenu ve tvaru $q_0, q_1, ...q_n$, kde q_0 je kvádr uzlu a q_n kvádr kořenu, pak koeficienty mají tvar

$$\frac{SA(q_0)}{SA(q_1)} \cdot \frac{SA(q_1)}{SA(q_2)} \cdot ... \cdot \frac{SA(q_{n-1})}{SA(q_n)}$$

To se však rovná

$$\frac{SA(q_0)}{SA(q_n)}$$

Výše zmíněnou hodnotu funkce přepíšeme na tvar

$$\sum_{(p,q)\in V_i} \frac{SA(q)}{SA(q_k)} \cdot c_t + \sum_{(p,q)\in V_l} \frac{SA(q)}{SA(q_k)} \cdot |p| \cdot c_p = g(T)$$

□

Ukázali jsme si, že při definici funkcí f, g jsme měli na mysli vždy tu samou funkci. Označme si ji nyní c a mluvme o ní jako o jedné funkci s více definicemi. Abychom byli přesnější, mluvíme o třídě funkcí, které se navzájem liší podle hodnoty konstant c_t, c_p. Abychom mohli mluvit o konkrétní funkci, značme si ji c_{c_t,c_p}.

Věta 2. *Pro akcelerační strom T a $t \in \mathbb{R}$: $c_{t\cdot c_t,t\cdot c_p}(T) = t \cdot c_{c_t,c_p}(T)$.*

Důkaz. Lze vidět z definice funkce g. □

Ve této větě jsme si ukázali, že pro správné cenové porovnání dvou stromů není důležitá velikost konstant c_t, c_p, ale spíše poměr mezi nimi.

Věta 3. *Nechť jsou stromy T_1, T_2 grafově isomorfní. Nechť obraz každého uzlu má stejný kvádr, jako jeho vzor, a pokud je to list, nechť má i stejnou množinu primitiv. Pak $c_{c_t,c_p}(T_1) = c_{c_t,c_p}(T_2)$.*

Důkaz. V definici funkce g vidíme, že není závislá na primitivech ve vnitřních uzlech. □

Zde jsme si ukázali, že pokud chceme pracovat se stromem určité kvality, nemusíme evidovat primitiva ve vnitřních uzlech, stačí pouze v listech. Celý strom můžeme zpět rekonstruovat např. tak, že primitiva ve vnitřních uzlech jsou sjednocením primitiv v obou potomcích.

3.4 Optimální stromy

Nyní, když jsme si definovali cenovou funkci, můžeme pořádně stanovit cíl této práce. Cílem práce je navrhnout algoritmy, které pro nějakou množinu primitiv P umožní nalézt akcelerační strom s co nejmenší hodnotou cenové funkce.

Definice. *Buď T akcelerační strom nad primitivy P a c_{c_t,c_p} cenová funkce. Pak o T řekneme, že je* optimální *dle c_{c_t,c_p}, je-li hodnota funkce pro tento strom menší nebo rovná hodnotě funkce na jakémkoli jiném stromě nad P.*

Věta 4. *Buď (p, q) list stromu T. Nechť $SA(q) > 0$ a nechť primitivum $r \in p$ neprotíná q. Pak T není optimální.*

Důkaz. Odstraněním r z p se neporuší žádný z axiomů akceleračního stromu, avšak hodnota cenové funkce klesne. □

Ukázali jsme si, že při hledání optimálních stromů nemá smysl uvažovat stromy, jejichž listy obsahují primitiva neprotínající kvádr.

4. Algoritmus stavby stromů

Podívejme se nyní na postup, jak najít efektivní akcelerační strom. Všimněme si, že pro jednu množinu primitiv existuje nekonečně mnoho stromů. Nelze tedy sestrojit všechny, ohodnotit je a vybrat ten nejlepší.

4.1 Schéma algoritmu

Náš algoritmus bude velmi podobný již existujícím algoritmům. Stromy stavějme shora. Podoba kořene je jasná. Ke kořenu budeme přidávat další uzly a strom rozšiřovat. Kvalitu stromu můžeme ovlivnit dvěma způsoby: velikostí kvádrů a počtem primitiv v uzlech.

Budeme vycházet z rekurzivní podoby cenové funkce (definovaná výše jako f_0). Budeme se snažit o její minimalizaci nejen v rámci celého stromu, ale v rámci každého uzlu a jeho podstromu.

Algoritmus bude stavět jednotlivé větve stromu, uzel po uzlu. V každém uzlu bude třeba rozhodnout, zda vyrobit list, nebo pokračovat ve stavbě. V případě pokračování bude třeba efektivně vyrobit syny, tj. 2 kvádry a dvě množiny primitiv.

Abychom se při tom dobře rozhodovali, prozkoumáme několik podstromů, které by šly v daném uzlu vyrobit, každý ohodnotíme a vybereme ten nejlepší. Tyto podstromy si dovolíme stavět jen do určité hloubky.

```
SplitInfo getBestSplit ( prm, box, depth )
{
   SplitInfo si (|prm| * cp);
   if (depth == 0) return si;

   for ((lprm, lb), (rprm, rb) in SPLITS)
      SplitInfo lsi = getBestSplit(lprm, lb, depth-1);
      SplitInfo rsi = getBestSplit(rprm, rb, depth-1);
      if (ct + (SA(lb)*lsi.cost + SA(rb)*rsi.cost)/SA(box) < si.cost)
         si.update();
   return si;
}
```

Funkce *getBestSplit* se snaží najít nejlepší rozdělení. V podstatě staví různé stromy do omezené hloubky a vrátí rozdělení, pro které existuje nejefektivnější strom. Všimněme si konstant *cp* (c_p) a *ct* (c_t), které ovlivní výběr rozdělení. Kvalita nalezeného rozdělení také záleží na množině *SPLITS*, ze které se rozdělení vybírají a testují.

```
Node BuildNode ( prm, box )
{
   Node node;
   SplitInfo si = getBestSplit (prims, box, depth);

   if(si.cost == |prm| * cp)
```

```
      node.makeLeaf(prms, box);
   else
      node.left  = BuildNode(si.lprm, si.lbox);
      node.right = BuildNode(si.rprm, si.rbox);
   return node;
}
```

Funkce *BuildNode* obdrží primitiva a kvádr a má pro ně postavit co nejlepší podstrom. Nejdříve najde nejlepší rozdělení do určité hloubky *depth*. Hodnota hloubky určuje kvalitu výsledného stromu. Poté se buď vyrobí list, nebo se strom rozvine. Strom pak postavíme velmi snadno.

```
Node root = BuildNode( P, AABB(P) );
```

4.2 SAH v kontextu algoritmu

Podívejme se nyní na povrchovou heuristiku v kontextu našeho algoritmu. Jak jsme si dříve naznačili, SAH zkoumá, zda je lepší vytvořit jeden list, či uzel se dvěma listy. To odpovídá hledání stromů do hloubky 1.

```
...
   SplitInfo si = getBestSplit ( prims, box, 1 );
...
```

Právě toto hledání do hloubky 1 způsobuje, že v určitých případech SAH vyrabí listy příliš brzy. V některých extrémních případech, např. když se ohraničující kvádry všech primitiv rovnají, SAH nemůže najít kvalitní rozdělení.

Při prvním pohledu na SAH se zdálo, že nám najde „absolutně nejlepší rozdělení“, tj. to s nejmenší cenou. Nenabízel se tedy žádný zřejmý způsob, jak SAH vylepšit. Až teď po důkladnější analýze vidíme, že SAH prohledává jen do hloubky 1 a jednoduchým vylepšením je např. zkoušet hledat do hloubky 2.

4.3 Možnosti konfigurace

Náš algoritmus je konfigurovatelný ve třech místech:

1. Konstantami c_p, c_t. Aby hodnota cenové funkce odpovídala času průchodu stromu náhodným paprskem v nějaké dobré implementaci, tyto dvě konstanty musí odpovídat času přechodu mezi dvěma listy a času testu paprsku s primitivem. Předpokládá se také, že doba testu průsečíků je u všech primitiv stejná.

2. Množinou *SPLITS*. Její prvky se budou lišit v závislosti na tom, zda se snažíme postavit BVH strom, KD strom či obecný strom. Počet prvků v této množině má vliv na časovou náročnost algoritmu.

3. Konstantou *depth*. Ta určuje, do jaké hloubky hledáme, a také má vliv na složitost algoritmu. Např. je-li její hodnota 0, hned se vyrobí list v kořenu a složitost algoritmu je konstantní.

Pro zajímavost jsem zkusil vyrobit malý test, na kterém si můžeme přiblížit vztah mezi hloubkou hledání, dobou hledání a kvalitou stromu. Test byl proveden se 100 úzkými válci délky 1, náhodně rozmístěnými uvnitř krychle se stranou 4. Stavěl se KD strom, dělící rovina se zkoušela umísťovat do pravidelné mřížky, 16 pozic na každé ose ($|SPLITS| = 48$), $c_p = 1, c_t = 1.2$.

Hloubka	Cena	Čas stavby
depth = 0	100.0	0.1 ms
depth = 1	13.0	2 ms
depth = 2	12.7	102 ms
depth = 3	12.5	8993 ms

Hloubka 0 odpovídá primitivům v jednom poli bez žádné akcelerační struktury. Hloubka 1 odpovídá SAH. Větší hloubky zatím nemají žádný vžitý název.

Nejen z tohoto testu, ale již z podoby algoritmu jsme vytušili, že prohledávání do větších hloubek je velmi drahé a je třeba s ním nakládat rozumně. Efektivním postupem je např. hledat do hloubky 1, a pokud nejlepší rozdělení navrhuje vyrobit list, potom zkusit hledat do hloubky 2. Užitečné může také být, když uděláme velikost množiny $SPLITS$ závislou na počtu rozdělovaných primitiv nebo na vzdálenosti uzlu od kořene. Jak konkrétně tyto věci realizovat si ukážeme při popisu a testování naší implementace.

5. Popis implementace

Součástí této práce je i implementace určená k zobrazování úsečkových modelů, která používá výše popsané metody. Hodí se především pro práci s photon beamy, zobrazování vlasů, chlupů a dalších modelů, které lze popsat úsečkami.

Na rozdíl od běžných implementací, naše struktury mají zabudované nalezení všech průsečíků paprsku s primitivy. To je obzvlášť užitečné právě u výše zmíněných photon beamů.

5.1 Primitiva

Vstupem programu jsou úsečky, tedy dvojice bodů ve trojrozměrném prostoru. Ty samozřejmě nemají žádný povrch a je nulová šance, že je náhodný paprsek protne.

Hledání průsečíku probíhá následovně. Při spuštění hledání se parametrem zadá hranice, na kterou se musí paprsek k úsečce přiblížit, aby byl započten průsečík. Výstupem je jednak vzdálenost, na které došlo k takovému průsečíku, jednak identifikátor dané úsečky, kterou paprsek zasáhl. Uživatel si pak může spočítat vlastní průsečík (včetně normály či dalších údajů), kde úsečka bude odpovídat třeba ose tenkého válce či *billboardu*.

Tento systém má i další výhodu, a to tu, že při změně „tloušťky úsečky“ není třeba přestavovat strom, což je obzvláště užitečné při práci s photon beamy.

5.2 BVH stromy

V implementaci máme 3 typy BVH stromů.

1. Klasické BVH - umísťuje celá primitiva do listů. Používá SAH.
2. SBVH - dělí uzly rovinou podle pravidelné mřížky, má povoleno „krájet“ primitiva do více listů. Také používá SAH.
3. SBVH depth2 - jako SBVH, avšak pokud nelze najít dobré rozdělení dle SAH, zkusí ho hledat do hloubky 2.

5.3 KD stromy

Implementace obsahuje 2 typy KD stromů.

1. Klasické KD - klasicky implementovaný KD strom. Nepoužívá „chybování“, řídí se čistě podle SAH.
2. KD depth2 - klasický KD strom, avšak, pokud nelze najít dobré rozdělení dle SAH, zkusí hledat do hloubky 2.

Struktury, které mají v názvu *depth2*, jsou pouze drobným vylepšením, které většinou nebude mít výrazný vliv na kvalitu stromu. Umožní nám to ale vyřešit problémové případy, kdy SAH navrhuje postavit jednouzlový strom. To vše uvidíme v kapitole o testování.

5.4 Stručná dokumentace

Implementace je napsaná v jazyce C++. Všechen kód nezbytný k použití stromů se nachází ve jmenném prostoru „fl“ (Fast Lines).

5.4.1 Základní struktury

Implementace obsahuje základní struktury pro 3D vektor, paprsek a průsečík. Jejich použití je celkem intuitivní, bližší informace lze najít v komentářích v kódu.

```
struct Vec
{
    float x,y,z;
    Vec(float x_, float y_, float z_) : x(x_), y(y_), z(z_) { }
    ...
};

struct Intersection
{
    float dist; int id;
    Intersection(float d, int i) : dist(d), id(i) {}
};

struct Ray
{
    Vec o, d, id;
    Ray(Vec o_, Vec d_) : o(o_), d(d_), id(1/d_.x, 1/d_.y, 1/d_.z) {}
};
```

5.4.2 Akcelerační stromy

Všechny stromové struktury dědí ze třídy *AccTree*. Ta má několik virtuálních metod a určuje rozhraní pro práci se stromy.

```
class AccTree
{
    int method;
public:
    AccTree(int m) : method(m) {};
    virtual void Build (float * lns, int num);
    virtual bool Intersect (Ray & r, Intersection * isc, float d);
    virtual bool IntersectAll (Ray & r,
                       std::vector<Intersection> * iscs, float d);
    ...
};
```

Konstruktor obsahuje jeden celočíselný parametr, který umožní škálování jednotlivých struktur. Funkce *Build* dostane ukazatel na pole hodnot typu *float*.

Každé 3 hodnoty určují vektor, každých 6 určuje úsečku. *num* je potom počet úseček.

Funkce *Intersect* resp. *IntersectAll* umožňují hledání prvního resp. všech průsečíků. Prvním parametrem je paprsek, druhý slouží k uložení informací o prvním resp. všech průsečících. Třetí parametr pak určuje minimální vzdálenost paprsku od úsečky, po jejíž dosažení se může průsečík započítat.

KD strom a BVH strom pouze rozšiřují tuto třídu a její virtuální metody. Navíc obsahují statické konstanty, které zle předat konstruktoru a specifikovat detaily daného stromu.

```
class BVH : public AccTree
{
public:
    static enum {CL, SS, D2};
    BVH(int m) : AccTree(m) {}
    ...
};
```

BVH::CL znamená klasické BVH, *BVH::SS* je verze s dělením prostoru (Spatial Splits), *BVH:D2* je verze výše označená jako *depth2*.

```
class KD : public AccTree
{
public:
    static enum {CL, D2};
    KD(int m) : AccTree(m)  {}
    ...
};
```

KD::CL je klasické KD, *KD::D2* je verze výše označená jako *depth2*.

5.4.3 Ukázka použití

Níže je malá ukázka použití, která dvakrát vypíše, že paprsek zasáhl úsečku.

```
float line[6] = {-1,-1,-1, 1,1,1};
Intersection isc(1e10, 0);
Ray r(Vec(0,0,-5),Vec(0,0,1));

BVH bvh(BVH::CL);  bvh.Build(line, 1);
KD  kd (KD ::D2);  kd .Build(line, 1);

if(bvh.Intersect(r, &isc, 0.5))
    std::cout << "Paprsek zasahl usecku.\n";

isc.dist = 1e10;
if(kd.Intersect(r, &isc, 0.5))
    std::cout << "Paprsek zasahl usecku.\n";
```

6. Testování

V této kapitole se podíváme na skutečnou efektivitu výše zmíněných postupů. Naši implementaci budeme zkoušet na několika různých modelech s použitím různě tlustých úseček.

Naši implementcaci budeme porovnávat s implementací *Embree*. Jedná se o velmi efektivní implementaci BVH stromů pro trojúhelníky. Implementuje prostorové rozdělování (Spatial Splits) a referencování primitiv z více listů. Používá SAH, avšak stejně jako všechny existující implementace, hledá rozdělení pouze do hloubky 1, což je její největší slabina.

Embree bylo vyvíjeno několik let a průchod stromem je mikrooptimalizovaný pro současné procesory. Optimalizace spočívá především v masivním použití SIMD instrukci pro vektorové operace. Dále v přizpůsobení velikostí struktur velikostem cache pamětí. Optimalizován je i test průsečíku paprsku s trojúhelníkem.

Zmiňme i to, že Embree používá kvaternární stromy, kde uzel má 4 potomky. Díky tomu je strom v průměru dvakrát mělčí. Jelikož jsou kvádry potomků uloženy v uzlu předka, výrazně se tím redukuje počet přístupů do paměti.

Díky těmto optimalizacím běží Embree v průměru dvakrát až třikrát rychleji, než jiné implementace s obdobně kvalitními stromy.

6.1 Vstupy

1. *Bun Hair* - jedná se o model vlasů s drdolem. Úsečky jsou poměrně jemně rozděleny.
2. *Straight Hair* - jedná se o model dlouhých vlasů. Úsečky jsou v průměru o trochu delší, než v předchozím modelu.
3. *Spaghetti* - zde jsou dlouhé úsečky, které spojují dva náhodné body na protilehlých stranách krychle.
4. *Spaghetti 2* - stejný model, jako předchozí, avšak s vyšším počtem úseček.
5. *Photon Beams* - zde úsečky představují cesty šíření paprsků světla z bodového světelného zdroje uvnitř zrcadlové krychle vyplněné homogénním poloprůhledným médiem.

6.2 Konfigurace

Níže je seznam stromových struktur a jejich konfigurací, se kterými byly testovány výše popsané sceny.

1. *BVH::CL* - klasické BVH, popsané výše.
2. *BVH::SS* - BVH se Spatial Splity.
3. *BVH::D2* - BVH, výše popsané jako depth2.
4. *KD::CL* - klasický KD strom, popsaný výše.

5. *KD::D2* - KD strom, výše popsaný jako depth2.

6. *Embree* - jelikož Embree umí pracovat pouze s trojúhelníky, každá usečka byla převedena na úzký obdélník (2 trojúhelníky), směřující ke kameře. Dva trojúhelníky mají téměř stejné AABB jako úsečka. Náročnost stavby a struktura stromů z úseček je velmi podobná stromům z odpovídajících trojúhelníků.

7. *Embree PS* - Embree pre-split. Tato konfigurace odpovídá výše zmíněnému Embree, avšak každý model byl zjemněn podle určité hranice délky, se kterou mělo Embree nejlepší výsledky. Toto ruční zjemnění by mělo odstranit nedostatky Embree plynoucí z použití SAH. Spolu s mikrooptimalizací Embree již teď můžeme očekávat, že tato umělá konfigurace bude nejrychlejší.

Stojí za to zmínit, že Embree nemá implementováno hledání všech průsečíků, to jsme ale nasimulovali několikanásobným hledáním prvního průsečíku.

Každý model a konfigurace byla testovaná pro první průsečík (značeno *First*) a všechny průsečíky (značeno *All*) se třemi různými tloušťkami (0.005 - W, 0.001 - M, 0.0005 - N). Pro lepší představu o podobě primitiv je ke každému testu připojena trojice obrázků (pro 3 různé tloušťky), kde barva pixelu odpovídá vzdálenosti mezi kamerou a nejbližším průsečíkem (světlejší místa jsou blíž ke kameře).

6.3 Výsledky

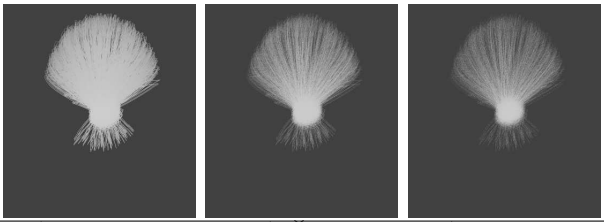
6.4 Hodnocení

Jak můžeme vidět z tabulek, s prvními dvěma modely si celkem dobře poradily všechny struktury. Tyto modely jsou poměrně jemné a SAH při stavbě stromů bohatě stačí. Drobný rozdíl lze pozorovat mezi BVH::CL a BVH:SS. Ukazuje, že i u jemných modelů nemusí být klasické BVH optimální a je lepší povolit dělení primitiv (Spatial Splits). Ostatní cenové rozdíly souvisí s implementací.

Zlom přišel u dalších dvou modelů. Zde efektivní stromy dokázaly postavit pouze struktury označené jako *depth2* a Embree s umělým zjemněním. SAH se v těchto případech „vzdává“ a vytváří velmi mělké, či dokonce jednouzlové stromy.

S pátým modelem si neporadilo pouze BVH::CL, které jako jediné nemůže dělit primitiva. Ostatní cenové rozdíly souvisí s implementací. Zajímavou je cena stromu Embree, která by měla odpovídat ceně BVH:SS, jelikož obě struktury implementují BVH se Spatial Splity. Rozdíl mezi nimi je dán nejspíše tím, že Embree obsahuje mechanismus, který nedovoluje stavět příliš rozsáhlé stromy. To lze vidět i na počtu uzlů u ubou struktur.

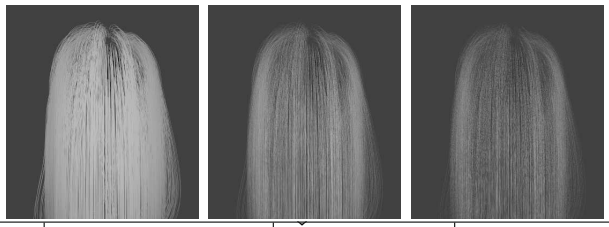
Za povšimnutí stojí také zjemnění, které vyžadují některé modely, aby měly co nejlepší výsledky s Embree (struktura *Embree PS*). Např. *Spaghetti 2* vyžaduje více než stonásobný nárust počtu primitiv. To se silně odráží na paměťových nárocích a času stavby stromu.



	Počet primitiv	Čas stavby	Počet uzlů	Cena
BVH::CL	82250	1445	22710	124.52
BVH::SS	82250	3236	121427	103.82
BVH::D2	82250	3445	125767	103.68
KD::CL	82250	3741	647015	58.20
KD::D2	82250	3773	650074	58.19
Embree	82250	725	84293	47.37
Embree PS	158090	1344	188920	37.97

	First W	First M	First N	All W	All M	All N
BVH::CL	786	1245	1419	1876	1844	1835
BVH::SS	663	1086	1219	1560	1526	1471
BVH::D2	652	1056	1194	1530	1472	1470
KD::CL	365	581	657	859	807	803
KD::D2	361	578	655	849	810	805
Embree	298	444	508	3118	1128	903
Embree PS	267	391	434	1843	800	680

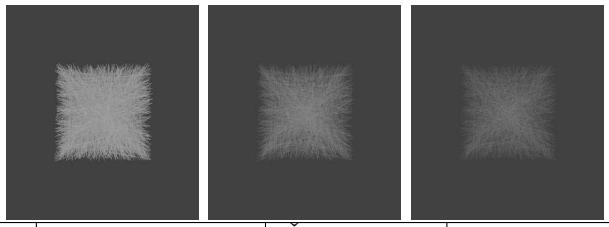
Tabulka 6.1: Bun Hair



	Počet primitiv	Čas stavby	Počet uzlů	Cena
BVH::CL	87460	1559	36243	149.75
BVH::SS	87460	1571	67669	136.24
BVH::D2	87460	1575	68138	136.15
KD::CL	87460	2673	563598	79.49
KD::D2	87460	2670	563804	79.49
Embree	87460	690	97761	47.87
Embree PS	486950	3377	530879	37.72

	First W	First M	First N	All W	All M	All N
BVH::CL	1488	2271	2580	3068	3016	3019
BVH::SS	1319	2012	2300	2735	2712	2691
BVH::D2	1319	2007	2341	2734	2692	2756
KD::CL	681	1135	1317	1643	1603	1590
KD::D2	677	1137	1309	1655	1604	1592
Embree	423	680	780	1863	1132	1044
Embree PS	365	545	606	1550	895	803

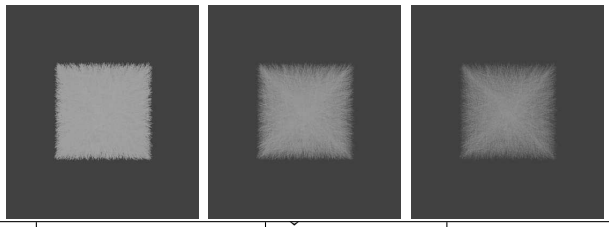
Tabulka 6.2: Straight Hair



	Počet primitiv	Čas stavby	Počet uzlů	Cena
BVH::CL	2000	17	179	1331.34
BVH::SS	2000	2	1	2000.00
BVH::D2	2000	436	14962	286.95
KD::CL	2000	2	1	2000.00
KD::D2	2000	366	76407	138.00
Embree	2000	91	1552	971.77
Embree PS	264741	1967	335726	94.52

	First W	First M	First N	All W	All M	All N
BVH::CL	5682	6072	6216	6570	6567	6572
BVH::SS	9463	9552	9450	10013	9945	9953
BVH::D2	461	842	969	1151	1140	1144
KD::CL	8920	9068	8898	9187	9163	9152
KD::D2	232	422	484	595	570	566
Embree	2889	3282	3455	13923	5522	4621
Embree PS	214	332	374	709	477	450

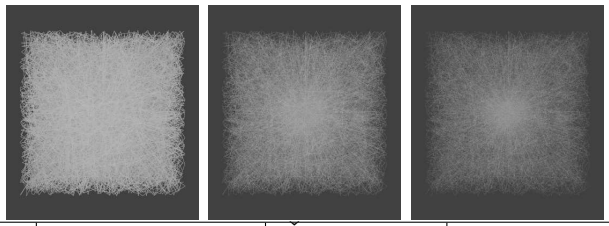
Tabulka 6.3: Spaghetti



	Počet primitiv	Čas stavby	Počet uzlů	Cena
BVH::CL	8000	85	608	4925.00
BVH::SS	8000	5	1	8000.00
BVH::D2	8000	3366	115754	579.99
KD::CL	8000	9	1	8000.00
KD::D2	8000	2908	602932	276.34
Embree	8000	143	5913	3724.86
Embree PS	1229361	10083	1624314	201.75

	First W	First M	First N	All W	All M	All N
BVH::CL	999999	999999	999999	999999	999999	999999
BVH::SS	999999	999999	999999	999999	999999	999999
BVH::D2	450	942	1346	2394	2345	2345
KD::CL	999999	999999	999999	999999	999999	999999
KD::D2	211	468	665	1289	1228	1205
Embree	999999	999999	999999	999999	999999	999999
Embree PS	221	434	653	2462	1417	1297

Tabulka 6.4: Spaghetti 2



	Počet primitiv	Čas stavby	Počet uzlů	Cena
BVH::CL	36294	597	9924	4399.96
BVH::SS	36294	4225	151943	589.37
BVH::D2	36294	4767	161798	587.16
KD::CL	36294	4106	783608	277.40
KD::D2	36294	4186	796709	276.70
Embree	36294	617	56946	911.01
Embree PS	552225	4836	696996	229.81

	First W	First M	First N	All W	All M	All N
BVH::CL	999999	999999	999999	999999	999999	999999
BVH::SS	1539	3305	4224	6247	6023	6012
BVH::D2	1543	3303	4179	6077	5977	5964
KD::CL	622	1508	1967	3021	2959	2867
KD::D2	620	1482	1929	3013	2876	2854
Embree	1712	4526	5662	27459	12227	10104
Embree PS	592	1291	1655	4704	2853	2611

Tabulka 6.5: Point Beams

Závěr

6.5 Zhodnocení nového přístupu

Zkusme si nyní shrnout látku naší práce. Podařily se nám dvě důležité věci.

Zaprvé jsme praktický problém řádně formalizovali a převedli na problém matematický. V rámci toho jsme také definovali rozdíl mezi KD stromy a BVH stromy. Ten spočívá pouze ve dvou různých axiomech a nese s sebou několik implementačních detailů. Ostatní vlastnosti a způsob přístupu je u obou typů stromů stejný a není třeba mezi nimi rozlišovat tolik, jako v minulosti.

Zadruhé se nám podařilo odstranit hranici kvality, kterou nabízí SAH, a převést tento problém na hledání kompromisu mezi časem stavby a kvalitou stromu. Pokud by se dnes v praxi pro scénu 10 minut stavěl ten nejlepší strom dle SAH, který by se poté 10 hodin vykresloval, náš přístup umožňuje věnovat několik dalších minut stavbě, díky kterým můžeme ušetřit několik hodin při vykreslování.

6.6 Budoucí výzkum

Byl bych velmi rád, kdyby se mnou navržený přístup k akceleračním strukturám v budoucnu uplatnil při jejich výuce. Vychází z jednoduché teorie, v rámci které lze snadno popsat většinu přístupů a metod, popsaných ve vědeckých článcích posledních doby. Nezatěžuje čtenáře spoustou detailů a nabízí větší prostor pro další zkoumání tématu.

V této práci jsme si představili *binární kvádrové akcelerační stromy*. Mohlo by být zajímavé zkoumat stromy, které nejsou binární či nejsou založeny na kvádrech. Tento směr a jeho praktická efektivita budou úzce spjaty s implementací.

Co se týče algoritmu, navrhli jsme si pouze základní podobu a neřekli přímo, jaké hloubky prohledávání se mají použít, ani jaká rozdělení (množina *SPLITS*) se mají prozkoumávat. Právě na těchto dvou věcech však záleží rychlost hledání kvalitních stromů.

Je také možné, že se podaří najít nový typ akceleračního stromu, který bude mít lepší praktické vlastnosti, než KD či BVH stromy. Bude však stále stejně snadno definovatelný.

Velký důraz by se měl klást i na matematickou část. Je možné, že se o stromech podaří dokázat několik významných vět, které budou mít pozitivní praktické důsledky, např. pomohou zúžit množinu *SPLITS* či umožní hledání dříve ukončit. Věty se mohou vztahovat na konkrétní BVH či KD strom, stejně tak mohou mluvit o konkrétních primitivech a využívat jejich pozitivních vlastností.

Seznam použité literatury

[1] Whitted, Turner *An Improved Illumination Model for Shaded Display.* ACM: Commun. ACM, 1980.

[2] Kajiya, James T. *The rendering equation.* ACM: SIGGRAPH Computer Graphics, 1986.

[3] Jarosz, Wojciech, Derek Nowrouzezahrai, Robert Thomas *Progressive Photon Beams.* ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2011), volume 30, 2011.

[4] MacDonald, David J., Booth, Kellogg S. *Heuristics for ray tracing using space subdivision.* Visual Computer, 6. vydání, stránky 153–166, Springer-Verlag New York, Inc., 1990.

[5] Manfred Ernst and Günther Greiner *Early split clipping for bounding volume hierarchies.* IN PROCEEDINGS OF THE 2007 IEEE/EG SYMPOSIUM ON INTERACTIVE RAY TRACING, 2007.

[6] Martin Stich and Heiko Friedrich and Andreas Dietrich *Spatial Splits in Bounding Volume Hierarchies.* Proc. High-Performance Graphics 2009

[7] Matt Pharr, Greg Humphreys *Physically Based Rendering, Second Edition: From Theory To Implementation.* stránka 239, ISBN 0123750792, 9780123750792, Morgan Kaufmann Publishers Inc. 2009

[8] Bittner, J., Hapala, M. and Havran, V. (2013), *Fast Insertion-Based Optimization of Bounding Volume Hierarchies.* Computer Graphics Forum, 32: 85–100. doi: 10.1111/cgf.12000

Přílohy

K práci je přiložené CD s implementací. Obsahuje 2 složky:

1. fl - složka s implementací výše popsaných struktur, určená pro začlenění do větších systémů.

2. test - obsahuje výše zmíněné struktury a pomocné nástroje pro práci s úsečkami. Obsah této složky umožňuje reprodukovat výše uvedené testy. Není však určen k budoucím úpravám či dalšímu použití. Součástí této složky je i knihovna Embree.

Vedle složek se dále nachází soubor *dokumentace.pdf* s podrobnějsími informacemi o implementaci.