Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Point Based Rendering

Plzeň, 2004 Martin Benda

Point Based Rendering

Point Based Rendering is a relatively new method of rendering which has lately received a lot of attention in computer graphics. The key principle of Point Based Rendering is the use of points as display primitives. Point Based Rendering is mainly focused on rendering very complex and large scenes.

The first part of this diploma thesis concerns with existing algorithms and methods of Point Based Rendering. The fundamental algorithms are explained and different approaches are discussed. However, this area of computer graphics is in a rapid development, therefore not all aspects of Point Based Rendering could be covered.

The second part of the thesis is an implementation of one of the presented Point Based Rendering methods—POP developed by Baoquan Chen and Minh Xuan Nguyen from the University of Minnesota at Twin Cities.

Obsah

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne ..... Martin Benda .....

Kapitola 1

Úvod

Point Based Rendering (neboli zobrazování použitím bodů) je poměrně mladá technologie, která používá jako zobrazovací prvky jednoduché body. Tím se liší od konvenčních metod používajících k zobrazování složitější primitiva (např. trojúhelníky).

Základním cílem Point Based Rendering technik je vytvořit systémy umožňující interaktivně zobrazovat velmi komplexní scény. Takovéto scény dnes vznikají např. jako výstup z prostorových skenerů, které produkují až několik stovek miliónů bodových vzorků. Dalším odvětvím, kde se pracuje s velmi komplexními scénami, je současná kinematografie využívající počítačovou grafiku pro ztvárnění scén, které by jiným způsobem buď nebylo možné natočit, nebo by to bylo velmi nákladné. Interaktivní zobrazení takovýchto obrovských scén není pomocí klasických ploškových zobrazovacích metod ani při použití moderního hardwaru možné.

Cílem práce je prozkoumat současné metody PBR, vytvořit o nich přehled a jednu zvolenou metodu implementovat.

1.1 Rozvržení dokumentu

Teoretická část práce se nachází v kapitole 2. Tato kapitola obsahuje stručný chronologický přehled vývoje Point Based Rendering a popisuje nejznámější a nejdůležitější práce v oblasti Point Based Rendering.

V kapitole 3 je dopodrobna rozebrána PBR metoda, která byla implementována jako součást této práce. V následující kapitole jsou uvedeny implementační detaily zvolené metody.

Kapitola 4 se zabývá implementačními detaily zvolené metody. Popisuje, v jakém prostředí byl program vytvořen, co je vstupem a výstupem programu a jakých možností hardwaru bylo využito.

Poslední část dokumentu je tvořena kapitolou 5, která obsahuje rozbor očekávaných a dosažených výsledků a srovnává experimentálně naměřené hodnoty.

V této kapitole se nacházejí tabulky a grafy, které srovnávají klasickou zobrazovací metodu s metodou PBR.

K dokumentu byly přidány tři dodatky — programátorská dokumentace, uživatelská dokumentace a ukázky grafických výstupů.

1.2 Seznam použitých zkratek

API — Application Interface
CPU — Central Processing Unit
EWA — Elliptical Weighted Average
FPS — Frames per Second
GPU — Graphical Processing Unit
LDC — Layered Depth Cube
LDI — Layered Depth Image
LOD — Level of Detail
MLS — Moving Least Squares
PBR — Point Based Rendering
POP — Point and Polygon

Kapitola 2

Point Based Rendering

Metody Point Based Rendering vychází z faktu, že trojúhelníky (či jiné plošky tvořící hraniční reprezentaci objektů) použité pro modelování velmi komplexních scén se při zobrazení často promítají do plochy menší než jeden pixel. Výhoda koherence v tradičních scan-line algoritmech je tím ztracena. Myšlenka PBR je taková, že v takto složitých scénách je zbytečná přesná geometrie a topologická informace a trojúhelníková síť je nahrazena mrakem bodů. Vědecké práce v oblasti PBR zkoumají, jak efektivně tyto bodové reprezentace zobrazovat, aniž by došlo ke značné újmě na kvalitě výstupu.

Základním problémem, který algoritmy PBR řeší, je zobrazení mraku bodů tak, aby výsledek vypadal jako souvislý povrch, ze kterého jsou body navzorkovány. Probíhá tedy rekonstrukce souvislého povrchu z množiny bodů, která nenese žádnou topologickou informaci. Většina PBR algoritmů tento problém řeší v obrazovém prostoru a rekunstrukci je pak nutné provádět po každé změně parametrů kamery. V posledních letech bylo pro rekonstrukci povrchu z bodů vyvinuto mnoho různých algoritmů. Přehled těch nejznámějších z nich je obsahem první poloviny této kapitoly.

Předmětem oblasti Point Based Rendering není však pouze rekonstrukce obrazu z bodových vzorků, ale také využití samotných množin bodů pro modelování komplexních objektů a scén. Proto také vzniklo velké množství různých datových struktur, které umožňují efektivně uložit navzorkované body. Tyto struktury jsou většinou hierarchické a dovolují jednoduše kontrolovat úroveň zobrazovaných detailů (*Level of Detail*). Jisté úsilí je také věnováno algoritmům, které z povrchových modelů efektivně získávají body vzorkováním. Mluví se pak o tzv. Point Based Modeling.

Další část je věnována metodám, které opouštějí myšlenku použití bodů jako *jediného* zobrazovacího primitiva a kombinují body s jinými prvky — jsou to tzv. hybridní metody. Mezi tyto metody patří i algoritmus POP, který se stal předlohou pro implementaci Point Based Rendering systému, jenž je součástí této práce.

Na konci kapitoly jsou diskutovány méně obvyklé techniky, jako např. sledování paprsku ve scénách tvořených body.

2.1 Stručný chronologický přehled

Za průkopnickou práci v oblasti Point Based Rendering je považována práce z roku 1985, jejímiž autory jsou Levoy a Whitted [9]. Argumentem pro použití bodů jako zobrazovacích primitiv bylo, že při zobrazování velmi složitých objektů nelze využít koherence ve scanline algoritmech a tudíž se stávají tyto algoritmy neefektivní. Myšlenkou bylo vytvořit univerzální algoritmus pro zobrazování bodů a pro převod libovolných objektů na bodovou reprezentaci před jejich zobrazením. Nicméně algoritmus a matematický aparát uvedený v této práci byl použitelný pouze pro diferencovatelné povrchové modely.

Po této publikaci se velmi dlouho nikdo k této problematice nevrátil, až v roce 1998 Grossman a Dally [8]. Cílem jejich práce bylo vyvinout algoritmus, který by zobrazoval komplexní objekty s dynamickým osvětlením.

Za skutečný rozmach Point Based Rendering lze považovat rok 2000, kdy byly publikovány dvě metody — *Surfels* [11], jejímž autorem je Pfister a *QSplat* [13], jejímiž autory jsou Rusinkiewicz a Levoy. Obě tyto metody se zaměřují na zobrazování velmi složitých scén, zejména množin bodů, které jsou získány jako výstup z prostorových skenerů.

V následujících letech lze sledovat bouřlivý vývoj point-based technologií. Jedná se zejména o *EWA splatting* [19], *MLS* povrchy [1], *raytracing* point-based modelů [2] a hybridní metody [4]. V současné době stále probíhá rozvoj těchto metod a algoritmů. Mezi hlavní směry vývoje Point Based Rendering patří využití hardwaru pro zobrazování [6].

2.2 Bodové vzorky

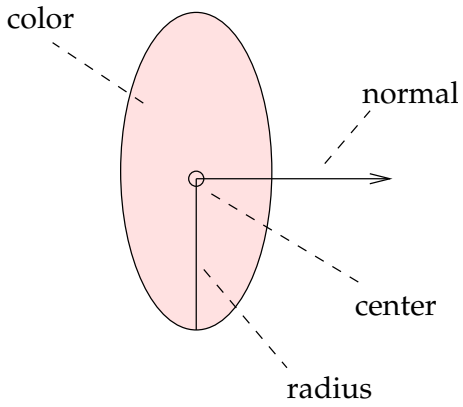
Bodové vzorky jsou základním stavebním kamenem metody Point Based Rendering. Každý bod má následující atributy:

- Polohu, která bývá nejčastěji vyjádřena kartézskými souřadnicemi v prostoru.
- Normálu, která odpovídá normále navzorkovaného povrchu v místě, kde se bod nachází.
- Barvu, souřadnice textury a jiné parametry pro stínování.

Pokud bodu přiřadíme určitou plochu, mluvíme pak o tzv. *surfelu*[1]. Ploška bývá často reprezentována jako kruh, jehož střed je pozice bodu a jehož poloměr je kolmý na normálu bodu. K ostatním parametrům bodu je pak navíc přidána velikost poloměru. Tyto surfely musí mít takový poloměr, aby dohromady zcela pokrývaly povrch navzorkovaného objektu a nevznikaly žádné díry. Dále v textu bude využíván jak termín surfel, tak bod ve stejném významu. Obrázek (2.1) představuje surfel s jeho atributy.

[1] *Surfel*, podobně jako *pixel*, vznikl jako zkratka z anglického termínu *surface element*.

V některých speciálních případech se k bodům přiřazují ještě další atributy jako např. hlavní směry a křivosti. Použití těchto informací vede ke kvalitnější rekonstrukci navzorkovaného povrchu, ale je pouze použitelné u diferencovatelných povrchů.



Obrázek 2.1: Surfel a jeho atributy.

2.3 Rekonstrukce obrazu z bodových vzorků

Hlavním cílem rekonstrukce obrazu z bodů je zobrazit hladký povrch, který je body reprezentován. Rekonstrukce musí být provedena tak, že mezi jednotlivými vzorky nevznikají díry a nedochází k aliasingu, přičemž eliminace děr je hlavním těžištěm většiny metod. Pokud bychom body zobrazili naivním způsobem tak, že každý bod by se zobrazil na jeden pixel, došlo by k tomu, že pouze část povrchu by byla zobrazena a mezi zobrazenými body by prosvítalo pozadí. Úkolem rekonstrukce je zaručit, že se tyto díry neobjeví a zobrazen bude celý hladký povrch.

Eliminace děr může být řešena několika základními způsoby:

1. **Ignorování** — Díry lze v některých případech ignorovat. Jedná se zejména o zobrazování nestrukturované geometrie (např. stromy, travnatý povrch), kde ignorování děr nehraje velkou roli pro výsledný vzhled.
2. **Převzorkování** — Díry lze zaplnit tak, že se vygeneruje tolik vzorků, aby mezi nimi žádné díry nevznikly při zobrazení v daném rozlišení.
3. **Detekce děr a následná rekonstrukce obrazu** — V prvním kroku se detekují pixely, které představují díry (rekonstrukce viditelnosti) a ve druhém kroku se tyto díry zaplní barvou získanou nejčastěji interpolací barev nejbližších sousedů (rekonstrukce obrazu).
4. **Splatting** — Jednotlivé vzorky se zobrazují tak, aby byly díry rovnou zakryté. V podstatě je tato metoda podobná té předchozí, ale oba kroky se provádějí najednou.

Všechny uvedené metody potřebují ke své správné funkčnosti znát prostorovou hustotu navzorkovaných bodů (vzdálenost sousedních vzorků).

2.3.1 Převzorkování

Převzorkování může být provedeno buď jako předzpracování, nebo může být adaptivní a dynamicky se počítat během zobrazování. Pokud se vzorkování provede v kroku předzpracování, musíme se smířit s tím, že výsledná množina bude bez chyby zobrazitelná jen z určité úzké množiny poloh kamery. Na druhou stranu adaptivní vzorkování se přizpůsobuje aktuální poloze kamery tak, že se vzorky generují až při zobrazování dle potřeby. Nevýhodou adaptivního vzorkování je větší výpočetní náročnost.

Mezi metody, které používají převzorkování pro rekonstrukci navzorkovaného povrchu patří zejména:

Randomizovaný z-buffer

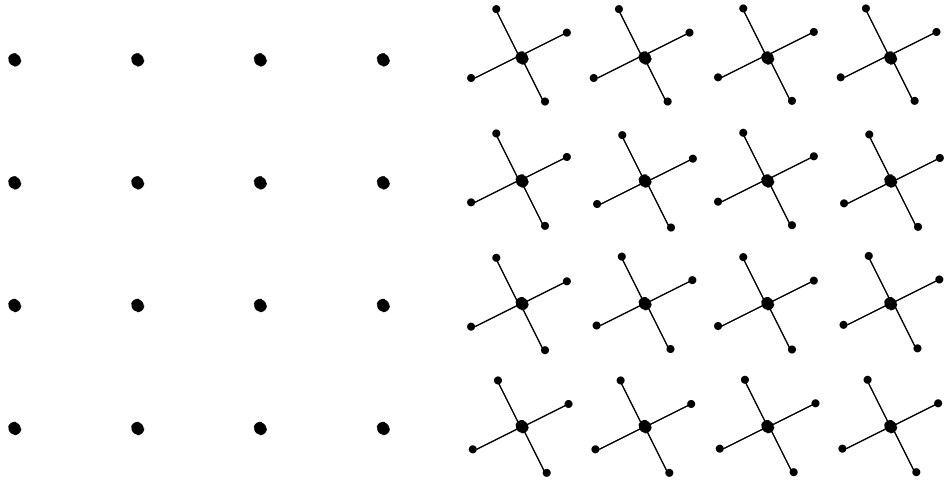
Wand a kolektiv [17] vytvořili metodu, která adaptivně vzorkuje počáteční množinu trojúhelníků. V předzpracování se vstupní množina trojúhelníků rozdělí do skupin, které sdružují trojúhelníky s podobným umístěním v prostoru. V průběhu zobrazování se pro každou skupinu určí hustota vzorkování tak, aby mezi vzorky nevznikaly díry. Tato hustota je odvozena od vzdálenosti skupiny od průmětny a celkové plochy trojúhelníků uvnitř skupiny. Trojúhelníky ve skupině se vzorkují náhodně, kde hustota pravděpodobnosti je proporcionální k ploše průmětu trojúhelníku (trojúhelníky, které více přispívají do výsledného obrazu mají vetší pravděpodobnost navzorkování). Výhodou tohoto algoritmu je jeho výpočetní náročnost — časová složitost je $O(a \log n)$, kde a je počet pixelů ve výstupním obraze a n je počet trojúhelníku ve vstupní scéně.

$\sqrt{5}$-vzorkování

Autorem této metody je Stamminger a Drettakis [16]. Vstupem je množina bodů, která vznikne pravidelným navzorkováním objektu v parametrickém prostoru plochy. Výchozí množina je dle potřeby dynamicky zjemňována podle $\sqrt{5}$ schématu. Krok vzorkování je znázorněn v obrázku (2.2).

MLS plochy

MLS je zkratkou pro *Moving Least Squares* (pohybující se nejmenší čtverce) a autorem této metody je Marc Alexa a kolektiv [1]. Vstupem této metody je množina navzorkovaných bodů (získaná například ze 3D skeneru). Pro každý bod je následně vypočtena lokální polynomiální aproximace pomocí matematického aparátu MLS. V průběhu zobrazování je tato aproximace dynamicky vzorkována tak, aby získané vzorky plně pokryly plochu a nevznikly žádné díry.



Obrázek 2.2: Dva následující kroky v $\sqrt{5}$ vzorkování.

2.3.2 Detekce děr a následná rekonstrukce obrazu

Tato technika zahrnuje dva kroky: V prvním kroku jsou identifikovány body pozadí, které jsou z obrazu odstraněny a označeny jako pixely děr. Ve druhém kroku, kdy obraz obsahuje pouze pixely popředí a díry, probíhá rekonstrukce. Jednotlivé techniky se liší zásadně v tom, jakým způsobem detekují pixely pozadí a jak je následně rekonstruují. Obecný algoritmus separace řešení viditelnosti od rekonstrukce obrazu je následující:

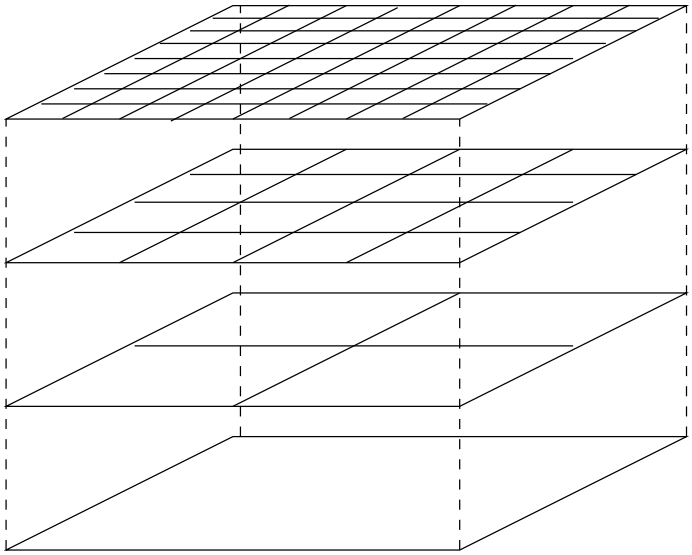
```
1. pro každý surfel:
    promítni surfel do obrazového prostoru
    vyřeš viditelnost pro okolí surfelu
    pokud je surfel viditelný
        ulož surfel do z-bufferu
2. pro každý viditelný surfel
    vykresli a vystínuj surfel
3. proved' rekonstrukci obrazu (zaplň díry)
```

Hierarchický z-buffer

Grossman a Dally [8] používají hierarchický z-buffer pro detekci děr a *pull-push* metodu pro jejich zaplnění.

Myšlenkou detekce děr je mít hierarchii z-bufferů s různým rozlišením. Z-buffer s největším rozlišením má stejné rozlišení jako cílový obraz a každý následující z-buffer v hierarchii má poloviční rozlišení než z-buffer předchozí — viz obrázek (2.3).

Pro každý bod, který je promítnut do obrazového prostoru je nejdříve proveden hloubkový test se z-bufferem s největším rozlišením. Pokud je teto test úspěšný, je proveden další test proti z-buffer s největším rozlišením, který ještě zajišťuje rekonstrukci bez děr (pro bližší objekty je vybrán z-buffer s nižším rozlišením).



Obrázek 2.3: Hierarchický z-buffer.

V případě, že oba testy jsou úspěšně splněny, je bod zapsán do obrazu a do příslušných z-bufferů. Díry se pak detekují podle rozdílu mezi hloubkami uloženými v z-bufferu s největším rozlišením a ostatními z-buffery v hierarchii. Je nutné si uvědomit, že pro výběr správného z-bufferu pro druhý test je potřebná znalost hustoty navzorkovaných bodů.

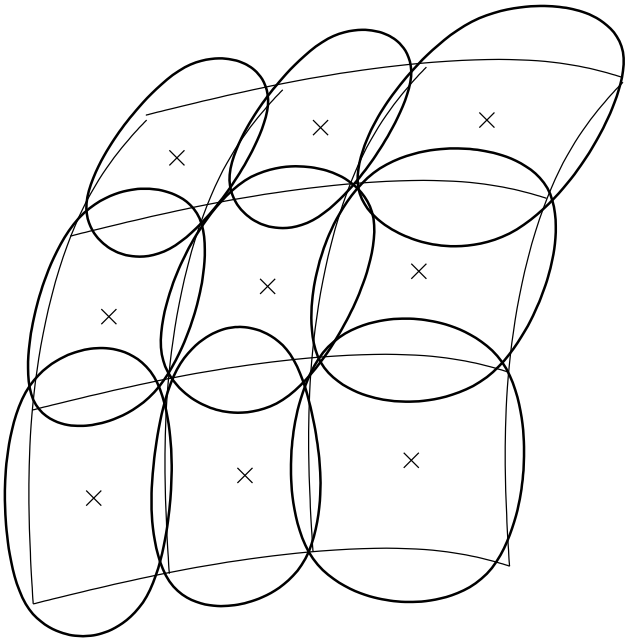
Pro rekonstrukci obrazu a zaplnění děr je použit tzv. *pull-push* algoritmus. Cílem je zaplnit pixely v děrách průměrnou barvou svých sousedů. Ve fázi *pull* se vygenerují obrázky s menším rozlišením, které jsou ve fázi *push* smíšeny pro získání aproximované hodnoty pixelů v děrách. Nicméně generování obrázků s menším rozlišením nemá již nic společného s hierarchií z-bufferů.

Surfels

V metodě *Surfels*, kterou publikoval Pfister a kolektiv [11], je pro detekci děr použito tzv. *visibility splatting*.

Při zobrazování se předpokládá, že každý navzorkovaný bod představuje malou část vzorkovaného povrchu a má tvar kruhu. Z této koncepce vznikl také termín *surfel* jako část povrchu. Při projekci do obrazu se surfel promítne do tvaru podobného elipse. Velikosti surfelů musí být zvoleny tak, aby zcela pokrývaly navzorkovaný povrch a při jejich projekci nevznikaly žádné díry. Každý pixel který vznikne při rasterizaci průmětu surfelu je testován vůči z-bufferu a pokud je test úspěšný, je příslušný pixel v obrazu označen za díru. Vystínovaný bod je do obrazu zapsán pouze na pozici středu elipsy (pokud byl hloubkový test úspěšný). Pokud je test neúspěšný, žádná akce se neprovede.

Z této procedury získáme obraz, který obsahuje pixely popředí, pozadí a pixely představující díry. Stejně jako v předchozím případě je nutné znát hustotu vzorků, aby mohla být zvolena správná velikost surfelů. Myšlenku metody visibility splatting zachycuje obrázek (2.4).



Obrázek 2.4: Surfely plně pokrývající navzorkovaný povrch.

Na první pohled se zdá tato metoda značně rozdílná od metody předchozí — hierarchie z-bufferů. Ale při bližším prozkoumáním zjistíme, že mají tyto metody mnoho společného: Body jsou do obrazu zapsány v takové velikosti, aby nedocházelo k tvorbě děr. V případě hierarchických z-bufferů je toho dosaženo volbou z-bufferu s příslušným rozlišením, visibility splatting toho dosahuje explicitní rasterizací promítnutých surfelů. Pokud by se ve visibility splatting metodě použily surfely, které by se promítaly do čtverců v zarovnané mřížce, dosáhli bychom stejných výsledků.

Rekonstrukce obrazu probíhá tak, že do viditelných bodů jsou umístěny Gaussovské filtry a barva pixelů nacházejících se v dírách je vypočtena jako vážený průměr blízkých viditelných pixelů. Poloměr Gaussovských filtrů se vypočítá z hustoty viditelných bodů v průmětně. Pro anti-aliasing je v této metodě použit supersampling.

2.3.3 Splatting

Další rekonstrukční technikou je tzv. *splatting*[2]. Na rozdíl od hierarchie z-bufferů a visibility splatting tato metoda řeší viditelnost a zaplnění děr najednou — neprovádí tyto činnosti ve dvou oddělených krocích.

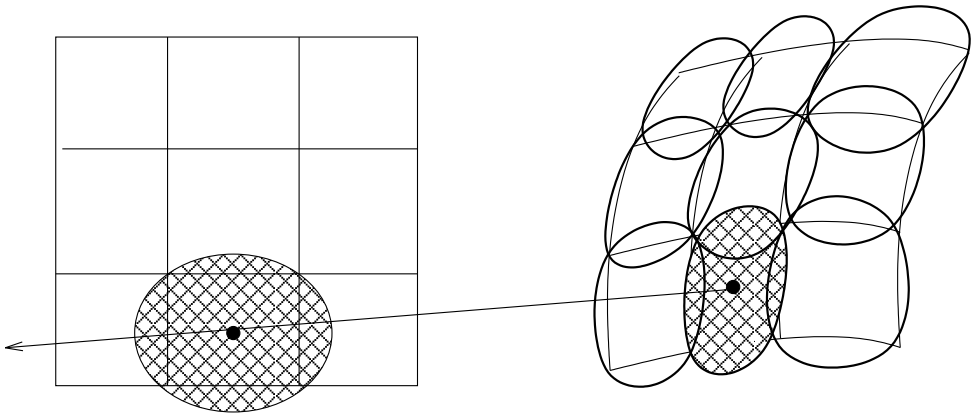
Stejně jako ve visibility splatting se předpokládá, že každému bodu je přiřazena ploška o určitém obsahu. Rekonstrukce výsledného obrazu spočívá ve vykreslení aproximací průmětů jednotlivých surfelů (která má přibližně tvar elipsy při použití surfelů ve tvaru kruhu). Průmět surfelu se nazývá *splat*. Ve splatting algo-

[2]Do češtiny lze přeložit jako „cáknout“ nebo „cákanec“. Dále v textu se budou používat anglické termíny *splatting* a *splat*.

ritmech bývá velikost jednotlivých splatů určena tak, aby byl pokryt celá plocha a mezi jednotlivými splaty nevnikaly díry. V metodě *surfels* byl splatting použit pouze pro detekci děr, zde je použit zároveň pro rekonstrukci výsledného obrazu. Obecný princip splattingu popisuje následující algoritmus:

```
1. pro každý surfel
    promítni surfel do obrazového prostoru
    vystínuj surfel
    vypočti atributy splatu
    do obrazu vykresli splat
2. pro každý pixel
    proved' postprocessing (stínování, normalizace)
```

Princip splattingu je uveden v obrázku (2.5).



Obrázek 2.5: Průmět surfelů do splatů, jejichž přibližný tvar je elipsa.

Kvalitu výsledného obrazu lze značně ovlivnit výběrem tvaru splatů. Tvar může být v nejjednodušším případě plný čtverec (tzv. *quad*), nebo lze použít splaty složitějších tvarů — kruh nebo elipsa. Případně je možné použít splaty s průhledností — fuzzy splatting a EWA splatting.

Čtvercové splaty

Splaty ve tvaru čtverce jsou tou nejhrubější aproximací přesného tvaru. Jejich výhoda je, že se dají velmi rychle vykreslit (jsou podporované většinou hardwarově — např. `GL_POINT` v knihovně OpenGL). Tvar splatu se neřídí orientací surfelu, což bývá zejména patrné na obrysech povrchových modelů [13]. Viditelnost je řešena pomocí běžného z-bufferu. Hloubka celého splatu je konstantní a je rovna hloubce bodu ve středu surfelu.

Kvalita výsledného obrazu je nízká, přesto se však tento tvar splatů často používá kvůli jednoduchosti a rychlosti vykreslování.

Obrázek 2.6: Smíšení fuzzy splatů.

Eliptické splaty

Elipsa je velmi dobrá aproximace skutečného tvaru průmětu surfelu. Použití elips vede k zvýšení kvality výsledného obrazu, zejména v blízkosti obrysů [13]. Výhodou je také to, že splat nemá ve všech pixelech stejnou hloubku, ale hloubka odpovídá průmětu tečné roviny surfelu. Nevýhodou je ovšem pomalejší vykreslení než v případě čtvercových splatů. Vykreslování elips nebývá běžně implementováno v současném grafickém hardwaru.

Fuzzy splatting

Nevýhodou použití jak čtvercových tak eliptických splatů je absence jakékoliv interpolace mezi barvami jednotlivých splatů. To vede ke vzniku artefaktů ve výsledném obraze (např. je patrný tvar splatů). Lepším řešením je použít tzv. *fuzzy splat*, který obsahuje navíc informaci o průhlednosti (alfa kanál). Průhlednost bývá ve středu splatu minimální a zvyšuje se směrem k okraji splatu — viz obrázek (2.6). Výsledná barva pixelu je poté vypočtena jako vážený součet barev všech přispívajících fuzzy splatů. Pokud je pozice pixelu (x, y), vypočte se jeho barva podle následujícího výrazu:

$$p(x,y) = \frac{\sum_i c_i w_i(x,y)}{\sum_i w_i(x,y)}, \tag{2.1}$$

kde $p(x, y)$ je barva pixelu na pozici (x, y), c_i je barva i-tého splatu a $w_i(x, y)$ je váha i-tého splatu v bodě (x, y).

Normalizace je nutná z toho důvodu, že splaty jsou na obrazu rozprostřeny nerovnoměrně a součet vah jednotlivých splatů není nutně roven jedné ve všech pixelech. Na současném grafickém hardwaru není možné provést normalizaci pro každý pixel (případně je tato operace příliš časově náročná), tak vznikla metoda, která částečně řeší normalizaci ve fázi preprocesingu [10]. Každému surfelu je přiřazen normalizační faktor, který je vypočten na základě projekce objektu z několika různých směrů během preprocesingu. Tento faktor poté zajišťuje, že součet vah splatů bude v libovolném místě blízko jedné.

Klasický z-buffer nám při testu zobrazování pixelu poskytne pouze dvě odpovědi — buď vykreslit, pokud se vykreslovaný pixel nachází v popředí, nebo za-

hodit, pokud není vidět. Při vykreslování fuzzy splatů je nutné, aby z-buffer uměl poskytnout ještě jednu odpověď — smísit, pokud vykreslovaný a již vykreslený pixel náleží sousedícím splatům. Vzhledem k tomu, že tvar povrchu a orientace surfelů může být v podstatě libovolná, žádná metoda, které tento problém řeší, se nevyhne určité dávce odhadování.

Nejjednodušší metodou, která řeší tento problém, je použití prahu při porovnávání hloubky zobrazovaného pixelu s hloubkou uloženou v z-bufferu [19]. Při zobrazování je určen práh ϵ. Pokud je absolutní hodnota rozdílu mezi hloubkami zobrazovaného pixelu a z-bufferu menší než tato hodnota, provede se smíšení barev, v opačném případě se postupuje podle pravidel běžného z-bufferu. Algoritmus hloubkového testu je následující:

```
if (pixel.z < z(x, y) - epsilon)
{
    // jsme vpopředí, pixel je normálně vykreslen
    z(x, y) := pixel.z
    color(x, y) := pixel.color
    weight(x, y) := pixel.weight
}
else if (pixel.z < z(x, y) + epsilon)
{
    // hloubka je přibližně stejná -> smíšení
    color(x, y) := color(x, y) + pixel.color
    weight(x, y) := weight(x, y) + pixel.weight
}
// jinak pixel není viditelný
```

Hardwarově se tato technika dá implementovat dvouprůchodovým zobrazením. Při prvním průchodu se pixely nevykreslují, pouze se zapisují hodnoty do z-bufferu zvýšené o dané ϵ. Při druhém průchodě se naopak nezapisuje nic do z-bufferu, ale všechny pixely, které projdou hloubkovým testem jsou smíšeny a vykresleny.

Prvním problémem této metody je, že uživatel musí vhodně zvolit velikost prahu. Druhým problémem je, že optimální velikost prahu není konstantní v celém obraze (blízko obrysů je vyšší než na plochách, které jsou kolmé k pozorovateli). Oba problémy lze částečně řešit tak, že práh je vypočítán jako poloměr splatu dělený kosinem úhlu mezi normálou a spojnicí mezi surfelem a pozorovatelem.

Další metoda se nazývá *z-ranges* [9, 12]. Pro každý pixel je vypočten interval hloubek a pokud se při hloubkovém testu interval nový a uložený překrývají, dojde ke smíšení barev a do z-bufferu je uložen interval vzniklý sloučením. Pokud jsou testované intervaly disjunktní, postupuje se podle pravidel klasického z-bufferu. Algoritmus hloubkového testu je následující:

```
if (pixel.zmax < zmin(x, y))
{
    // jsme v popředí, pixel je normálně vykreslen
    zmin(x, y) := pixel.zmin
```

```
        zmax(x, y) := pixel.zmax
        color(x, y) := pixel.color
        weight(x, y) := pixel.weight
    }
    else if (pixel.zmin < zmax(x, y))
    {
        // intervaly se překrývají -> smíšení
        zmin(x, y) := min (zmin(x, y), pixel.zmin)
        zmax(x, y) := max (zmax(x, y), pixel.zmax)
        color(x, y) := color(x, y) + pixel.color
        weight(x, y) := weight(x, y) + pixel.weight
    }
    // jinak pixel není viditelný
```

Hloubkový interval se pro splat vypočte jako minimum a maximum z-souřadnice příslušného surfelu po transformaci do prostoru kamery.

2.4 Eliminace neviditelných částí scén

Metody eliminace neviditelných částí scény nejsou nutně součástí Point Based Rendering technik, nicméně bývají implementovány ve většině existujících zobrazovacích systémech. Použití těchto metod vede ke zvýšení rychlosti renderování, což bývá často cílem PBR systémů. Existuje několik druhů eliminace neviditelných částí: ořezávání podle pohledové pyramidy (*view frustum culling*), eliminace částí orientovaných pryč od pozorovatele (*backface culling*) a eliminace částí zakrytých jinými částmi scény (*occlusion culling*).

Eliminace jednotlivých surfelů by byla neefektivní. Proto se provádí tyto operace s celými skupinami bodů. Tyto skupiny většinou přímo vyplývají z datových struktur, které jsou v PBR systémech použity (většinou se jedná o hierarchické, stromové struktury).

Ořezávání podle pohledové pyramidy

Pohledová pyramida představuje viditelnou část prostoru před scénou. Pokud se celá skupina (reprezentovaná např. ohraničujícím kvádrem) nachází mimo pohledovou pyramidu, všechny surfely v této skupině se při zobrazování neuvažují. Naopak, pokud se cela skupina nachází uvnitř, není potřeba už provádět test pro případné podskupiny v ní obsažené. Časová náročnost tohoto testu je nezávislá na počtu bodů ve skupině.

Eliminace částí otočených od pozorovatele

Pro určení, zda je daná skupina bodů zcela otočená od pozorovatele, je nutné nějakým efektivním způsobem zjistit, zda normály všech surfelů míří pryč od

pozorovatele. Standardní technika jsou tzv. normálové kužely. U každé skupiny je uložen kužel, který obsáhne normály všech surfelů. Při testování viditelnosti stačí zjistit, zda pozorovatel „vidí dovnitř kužele“. Časová náročnost tohoto testu je taktéž nezávislá na počtu bodů ve skupině.

Eliminace zakrytých částí

Eliminace zakrytých částí není součástí současných PBR systémů. Jedinou výjimkou jsou tzv. *visibility masks*, jejichž autorem je Grossman a Dally [8].

2.5 Sledování paprsku

Motivace pro zobrazování množin bodů metodou sledování paprsku je mít možnost výpočtu globálního osvětlení bez nutnosti převodu bodové reprezentace na reprezentaci jinou. Základním problémem při sledování paprsku je určení průsečíku paprsku s povrchem.

Autorem první publikované metody je Schaufler a Jensen [14]. Hledání průsečíku s plochou probíhá ve dvou krocích: detekování průsečíku a následný výpočet aproximace atributů tohoto průsečíku (poloha, normála, barva atd.).

Detekce průsečíku hledá průnik paprsku s množinou orientovaných disků, které jsou umístěny do každého bodu. Poloměr všech disků je globální parametr algoritmu. Pro urychlení nalezení průsečíku v množině bodů, která bývá zpravidla velmi rozsáhlá, je použit oktanový strom.

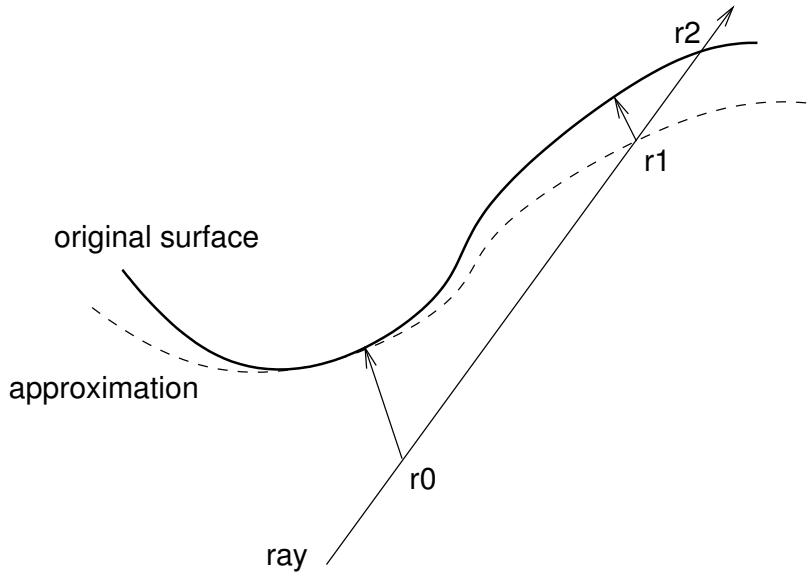
Pro každý detekovaný průsečík se nalezne podmnožina bodů, která leží v blízkosti detekovaného místa a parametry průsečíku se spočtou jako vážený průměr hodnot parametrů jednotlivých bodů v této podmnožině. Jako váha se bere přibližná vzdálenost disku od paprsku.

Druhá metoda pochází od autorů Adamsona a Alexy [2] a je založena na interpolaci technikou MLS. Tato metoda spočívá v iterativním výpočtu průsečíku paprsku s plochou. Aproximace průsečíku se iterativním procesem počítá tak dlouho, dokud není dosaženo požadované přesnosti. Iterativní krok je založen na projekci bodu ležícího na paprsku a následném výpočtu průsečíku s lokální polynomiální aproximaci povrchu:

1. Odhadem je získán počáteční bod r_0 nacházející se na paprsku.
2. Tento bod je pomocí matematického aparátu MLS promítnut na povrch, čímž získáme průmět bodu r'_0 a polynomiální aproximaci plochy v okolí r'_0.
3. Je vypočten průsečík paprsku s polynomiální aproximací r_1, který leží blíže hledanému průsečíku s plochou.

4. Pokud je dosaženo požadované přesnosti, je iterace ukončena a výsledkem je průsečík nalezený v bodě 3. Jinak je bod r_0 nahrazen bodem r_1 a jde se na krok 2.

Praxe ukazuje, že pro výpočet průsečíku s velkou přesností stačí většinou dvě až tři iterace. Proces iterace je znázorněn v obrázku (2.7).



Obrázek 2.7: Iterativní proces při hledání průsečíku paprsku s MLS plochou.

2.6 Modelování pomocí bodů

Některé techniky PBR používají bodové vzorky pouze jako přechodnou reprezentaci při zobrazování [9]. Nicméně většina ostatních PBR algoritmů používá bodovou reprezentaci scény místo klasické trojúhelníkové sítě. Jedná se pak o tzv. *Point Based Modelling*.

Téměř ve všech případech se jedná o hierarchické stromové datové struktury, které v jednotlivých úrovních hierarchie uchovávají informace o objektu v různých rozlišeních. Listy stromu obsahují nejvíce detailní reprezentaci geometrie, zatímco uzly blíže ke kořenu stromu obsahují hrubší aproximace objektu. Tato reprezentace objektů ve více rozlišeních je použita pro kontrolu úrovně zobrazovaných detailů (*Level of Detail*). Navíc toto hierarchické uložení geometrie může být použito pro kompresi dat, která spočívá v uložení pouhých rozdílů mezi jednotlivými úrovněmi stromu a vhodné kvantizaci těchto rozdílů.

V této části uvedeme dva příklady takovýchto hierarchických struktur, které ukazují dva hlavní směry, které se v PBR systémech používají: Hierarchický obálkový strom a LDC strom.

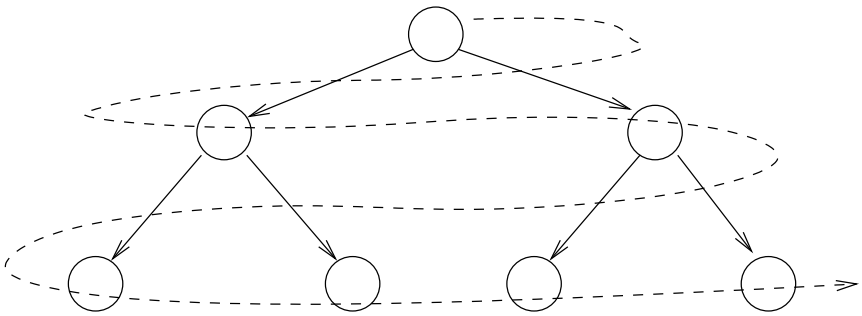
2.6.1 Hierarchická obálková struktura

Tato struktura je představena v systému QSplat, jehož autorem je Rusinkiewicz a Levoy [13]. Datová struktura je založena na hierarchii obálkových koulí. Body (nebo surfely) jsou v této struktuře reprezentovány jako koule.

Každý uzel stromu obsahuje:

- Střed a poloměr obálkové koule.
- Normálu.
- Šířku normálového kužele, který je použit k eliminaci uzlů, které jsou zcela otočeny pryč od pozorovatele.
- Další volitelné parametry jako např. barva.

Tento strom je vytvořen ze vstupní množiny bodů během preporcessingu a je zapsán na disk. Pro zápis na disk je použito průchod stromem do šířky — tím je zajištěno, že v souboru jsou uloženy uzly v pořadí se vzrůstající detailností. Toho lze využít k postupnému zobrazování objektu během načítání stromu (např. při načítání přes síť je uživateli již od počátku prezentována hrubá aproximace objektu, která postupně získává detaily).



Obrázek 2.8: Pořadí uložení uzlů stromu na disku.

Konstrukce stromu obálkových koulí probíhá zdola nahoru. Vstupem pro preprocessing je trojúhelníková síť. Vrcholy této trojúhelníkové sítě jsou použity jako středy výchozích koulí. Poloměry jsou zvoleny tak, aby se koule zcela překrývaly, tj. nevznikaly mezi koulemi žádné díry. Konstrukce uzlů vyšší úrovně probíhá rekurzivně. Každý interní uzel obsahuje přibližně čtyři uzly nižší úrovně.

Komprese probíhá tak, že se parametry uzlu kvantizují. Pozice středu a poloměr koule je vyjádřen relativně vůči parametrům nadřazeného uzlu. Tyto hodnoty jsou kvantizovány na 13 bitů. Normály jsou zakódovány do 14 bitů, šířka normálového kužele do dvou a barva do 16 bitů. Kompletní uzel se všemi informacemi včetně barvy zabírá v paměti 48 bitů.

Při zobrazování objektu se stromová struktura rekurzivně prochází. Pro každý uzel může nastat jeden z těchto případů:

1. Uzel není vidět (je zcela mimo pohledovou pyramidu, nebo je orientován pryč od pozorovatele). V tomto případě se uzel, ani uzly pod ním uložené, dále nezpracovávají.

2. Velikost projekce obálkové koule je větší než zvolený práh. Hierarchie se dále rekurzivně prochází.
3. Velikost uzlu v projekci je menší než práh, nebo je to uzel koncový. V tomto případě je vykreslen splat.

Práh, podle kterého se rozhoduje, zda bude rekurze ukončena, nebo se bude pokračovat, bývá zvolen buď ručně předem, nebo se adaptivně dopočítává podle aktuální rychlosti vykreslené scény. Algoritmus pro zobrazení stromové hierarchie je následující:

```
RenderHierarchy ( node )
{
    if (node.isVisible())
    {
        // tento uzel a celý podstrom se přeskočí
    }
    else if (node.isLeaf()
             or node.projectedSize() < threshold)
    {
        // vykreslí se splat
        DrawSplat(node)
    }
    else
    {
        // pokračuje se dále vrekurzi
        foreach child in node.children
        {
            RenderHierarchy(child)
        }
    }
}
```

Podobná datová struktura je také použita v systému POP [4], který se stal předlohou pro implementaci, jež je součástí této práce.

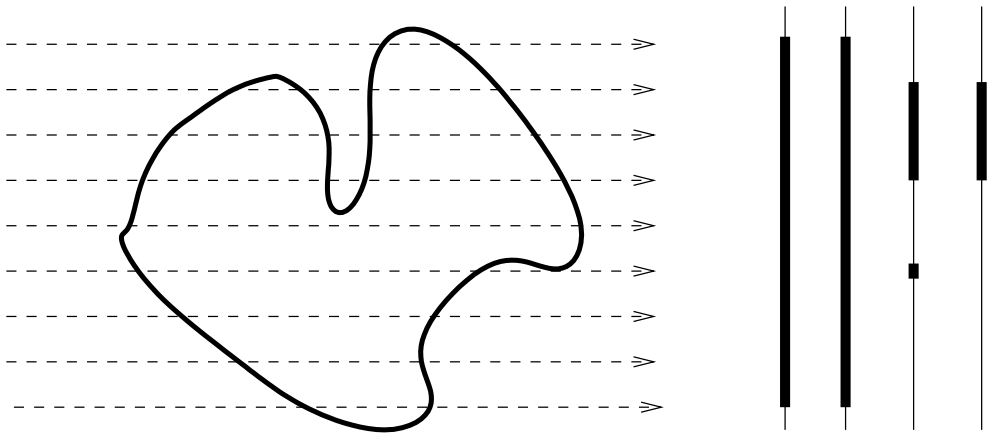
2.6.2 LDC strom

V systému Surfels, jehož autorem je Pfister a kolektiv [11], je pro reprezentaci scény použit tzv. *LDC strom*[3]. Před popisem samotného LDC stromu si ukážeme jakým způsobem je vstupní model navzorkován.

Objekt je vzorkován ze tří navzájem ortogonálních směrů. Vzorky jsou získány metodou sledováním rovnoběžných paprsků z každého ze tří směrů. Jako vzorky se ukládají všechny průsečíky podél paprsků, ne pouze ten první, jak bývá zvy-

[3] *LDC* je zkratka z anglického *Layered Depth Cube*, což je rozšíření tzv. *Layered Depth Image* (*LDI*), což je datová struktura používaná v technologii *image based rendering* — viz [5].

kem. Tím získáme tři navzájem ortogonální LDI (layered depth image), které tvoří tzv. layered depth cube (LDC).



Obrázek 2.9: Layered Depth Image.

LDC strom je vlastně oktanový strom obsahující ve svých uzlech LDC, které reprezentují části objektu. Kořenový uzel stromu obsahuje navzorkovaný celý objekt. Každý další uzel stromu obsahuje LDC navzorkovaný s polovičním krokem, než který je použit v uzlu nadřazeném. Počet vzorků v každém uzlu je tedy konstantní. LDC v každém uzlu je dále rozdělena na bloky o velikosti definované uživatelem (např. 8×8).

Zobrazovací algoritmus rekurzivně prochází oktanový strom a pro každý uzel provádí testy vůči pohledové pyramidě a orientaci vzhledem k pozorovateli (test normálového kuželu). Rekurzivní procházení je zastaveno, když se narazí na uzel s dostatečnou hustotou vzorkování. Vzorky uložené v uzlu jsou poté zobrazeny do obrazu jako splaty.

Nevýhodou LDC stromu oproti hierarchii použité v systému QSplat je to, že vstupní data musí být navzorkována s pravidelným krokem. Nelze ji tedy použít pro libovolnou vstupní množinu bodů, aniž by došlo k převzorkování. Tato metoda je velmi výhodná pro zobrazování dat, která jsou např. výstupem prostorových skenerů.

2.7 Vlastnosti PBR a další směry vývoje

Point Based Rendering je metoda, která je poměrně mladá a v současné době prochází bouřlivým vývojem. Zde si uvedeme výhody, které použití bodů jako zobrazovacích a modelovacích primitiv přináší. Dále si také nastíníme, jakými směry dnes vývoj PBR technologií kráčí.

Výhody bodů při zobrazování

- Bod je velmi jednoduchý, snadno se s ním pracuje.

- Velmi rychle zobrazování bodů (i softwarově).
- Interaktivní zobrazování velmi komplexních scén (až stovek miliónů trojúhelníků [18]).
- Velmi jednoduchá implementace LOD.
- Škálovatelný poměr mezi kvalitou a rychlostí.

Výhody bodů při modelování

- Jednoduchá práce s topologií (body žádné topologické informace nenesou).
- Snadné generování reprezentací modelů v různých rozlišeních.

Nevýhody bodové reprezentace

- Velmi neefektivní pro práci s modely, které obsahují velké plošky (není využito koherence tak jako ve scanline algoritmech).
- Nesnadná reprezentace nespojitých změn normál na ploše (ostré hrany).
- V současné době nedostatečná podpora v grafickém hardwaru.

Budoucí výzkum

Stručný přehled směrů, kterým se bude pravděpodobně výzkum PBR ubírat:

- PBR v animaci.
- Rychlé, kvalitní a robustní metody sledování paprsku.
- Efektivní implementace v hardware (zejména se jedná o splatting).
- Reprezentace a zobrazování nespojitostí (ostré hrany, okraje).
- Korektní antialiasing.
- Standardní API pro Point Based Rendering (tak jak existuje dnes pro polygonální zobrazování).
- Nové aplikace pro Point Based Rendering.

Kapitola 3

Popis použité metody

POP, je hybridní PBR technika, jejímž autorem je Baoquan Chen a Minh Xuan Nguyen [4]. Metoda se nazývá hybridní, protože pro zobrazování používá jak body, tak trojúhelníky. Kombinuje tyto dvě primitiva, aby bylo dosaženo co nejlepšího poměru mezi kvalitou výstupu a výpočetní náročností. Body se použijí v místech, kde by se trojúhelníky zobrazily na příliš malou plochu, trojúhelníky se naopak použijí pro zobrazení velkých rovinných ploch, kde by vzorkování na body bylo neefektivní.

Tato metoda je velmi podobná systému QSplat [13], ze kterého vychází. Používá velmi podobnou hierarchickou datovou strukturu pro uložení scény, ale liší se právě v rozšíření o použití trojúhelníků.

Přestože je tato metoda poměrně jednoduchá, je díky své hybridní podstatě velmi flexibilní a obecná. Toto byly hlavní argumenty pro její výběr.

3.1 Datové struktury

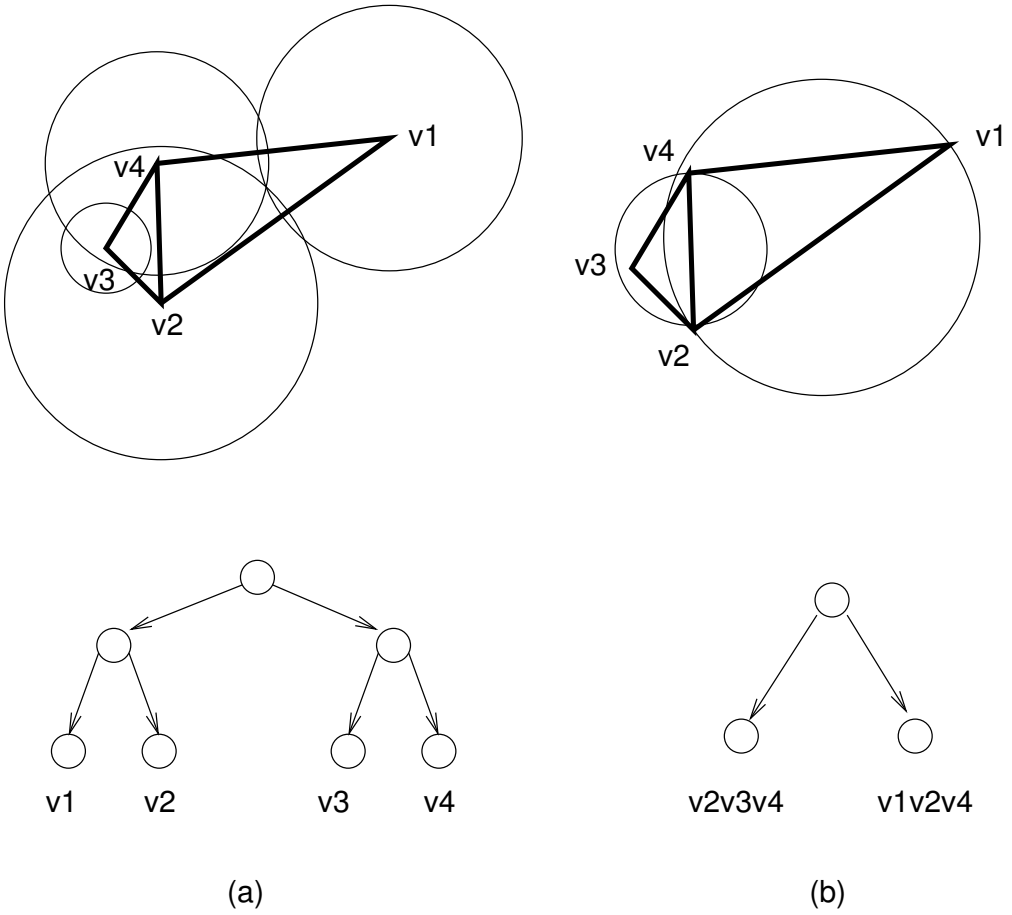
Vstupem metody je obecná trojúhelníková síť. Na rozdíl od metody QSplat nemusí být tato síť přibližně uniformní. Výsledná datová struktura je strom, jehož uzly jsou tvořeny obálkovými koulemi.

Trojúhelníky původní spojité sítě tvoří listy stromové hierarchické datové struktury. Každému trojúhelníku je přiřazena obálková kulová plocha včetně různých atributů, které zahrnují normálu, šířku normálového kužele, barvu apod. Normála přiřazená ke kouli je spočtena jako průměr normál ve vrcholech trojúhelníku. Podobně tak barva může být spočtena jako průměr barev v jednotlivých vrcholech. V případě texturování se barva získá konvolucí všech texelů[1] uvnitř trojúhelníku. Atributy každého dalšího uzlu výše v hierarchii jsou spočteny jako průměr hodnot dceřiných uzlů. Obálková koule je umístěna tak, aby co nejtěsněji obklopovala obálky uzlů v ní obsažených.

[1] *Texel* je — podobně jako pixel — zkratka z anglického *texture element*.

Prvním rozdílem mezi systémem POP a QSplat je to, ze POP obsahuje v listech stromové struktury trojúhelníky původní sítě, kdežto QSplat obsahuje pouze koule a výchozí trojúhelníky po konstrukci hierarchie koulí již dále neuvažuje. Druhým rozdílem je způsob výpočtu koulí v listech. QSplat umisťuje kouli do každého vrcholu trojúhelníkové sítě, zatímco POP umisťuje kouli kolem každého trojúhelníku. V obou případech vygenerované koule zcela pokrývají původní povrch a nevznikají žádné díry.

Rozdíl mezi strukturou použitou v metodě QSplat a POP je znázorněn v obrázku (3.1).



Obrázek 3.1: Stromová struktura obálkových koulí použitá v QSplat (a) a POP (b).

3.2 Předzpracování

Pro konstrukci (rekurzivní) datové struktury je použit jednoduchý rekurzivní algoritmus. Tento algoritmus probíhá pouze jednou během předzpracování. Výsledná datová struktura může být uložena na disk a sloužit jako efektivní forma uložení scény. Je možné použít stejné metody komprese, jako byly použity v technice QSplat [13, 4].

3.2.1 Algoritmus vytváření stromu

V prvním kroku algoritmu se vygeneruje množina koncových uzlů N_0 — pro každý trojúhelník ve vstupní množině jeden. V dalším kroku se spustí rekurzivní algoritmus, který rekurzivně dělí množinu N_0 vždy podle nejdelší osy ohraničující krychle dokud není dosaženo kritéria pro zastavení rekurze. Rekurze je ukončena v okamžiku, kdy pracovní množina rekurentní procedury obsahuje čtyři nebo méně uzlů. V tom okamžiku se vytvoří nový uzel ohraničující tyto čtyři (nebo méně) uzlů a zařadí se do množiny N_1. Tento postup se iterativně provádí pro posloupnost množin N_i tak dlouho, dokud se nevytvoří množina N_k obsahující právě jeden uzel — kořen stromu. Formální zápis algoritmu je následující:

```
1. Vytvoř výchozí množinu uzlů N(0)
2. Přiřad' i := 0
3. Rozděl množinu N(i) na skupiny uzlů o maximálně
   4 prvcích (zavolá se procedura
   Divide(i + 1, N(i)))
4. Pro tyto skupiny vytvoř nové uzly a přiřad' je
   do množiny N(i + 1)
5. Přiřad' i := i + 1
6. Pokud množina N(i) obsahuje pouze jeden uzel,
   tak konec, kořenem je uzel v množině N(i)
   Jinak jdi na krok 3
```

Rekurzivní algoritmus pro dělení množiny N_i na podmnožiny o čtyřech a méně prvcích:

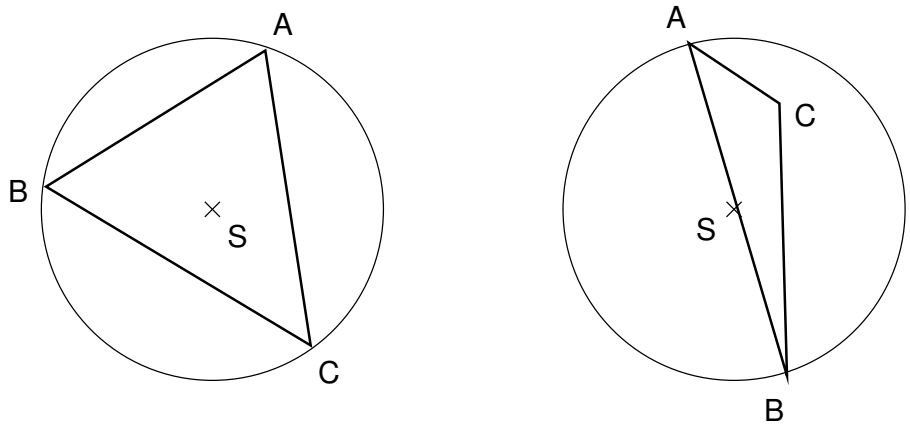
```
Divide(i, N)
{
    if (N.size <= 4)
    {
        vytvoř nový uzel Nn obsahující uzly z N
        N(i) := N(i) + Nn
    }
    else
    {
        rozděl N v místě mediánu podle nejdelší
        osy bounding-boxu na množiny Na a Nb
        Divide(i, Na)
        Divide(i, Nb)
    }
}
```

3.2.2 Hledání obálkové koule

Správný výpočet obálkových koulí je velmi důležitým krokem při vytváření stromové struktury. Obálková koule v uzlu musí obsahovat úplně všechny koule ve svém podstromu a zároveň je musí obalovat co nejtěsněji. V případě větších koulí, než je potřeba, dochází při zobrazování k tomu, že se zobrazuje scéna detailněji, než je požadováno. Pokud by obálková koule neobsahovala zcela všechny koule v podstromu, došlo by k degradaci kvality výstupu, protože by se rekurzivní proces vykreslování ukončil příliš brzy. V této části uvedeme matematické postupy výpočtu obálkových koulí jak pro koncové uzly obsahující pouze jeden trojúhelník, tak pro vnitřní uzly stromu.

Výpočet obálkové koule pro trojúhelník lze redukovat na rovinný problém — nalezení obálkové kružnice. Průměr obálkové kružnice je pak zároveň průměrem koule. Rozlišujeme dva případy:

1. Vnitřní úhly trojúhelníku jsou všechny menší než 90^o. V tomto případě je obálková kružnice rovna kružnici trojúhelníku opsané.
2. Jeden z vnitřních úhlů trojúhelníku je větší než 90^o. V tomto případě je průměr obálkové kružnice roven nejdelší straně trojúhelníku.



Obrázek 3.2: Výpočet obálkové koule pro koncové uzly.

Kružnice opsaná trojúhelníku tvořeným body $\mathbf{v_1}$, $\mathbf{v_2}$ a $\mathbf{v_3}$ v prostoru se vypočítá podle následujícího postupu:

1. Prostorový problém se převede na jednodušší rovinný tak, že zvolíme nový souřadný systém v rovině trojúhelníku a v tomto systému budeme úlohu řešit. Jako jedna báze nového prostoru se zvolí vektor

$$\mathbf{x} = \mathbf{v_2} - \mathbf{v_1}. \tag{3.1}$$

   Druhá báze se zvolí tak, aby byla kolmá na $\mathbf{x}$, měla stejnou velikost jako $\mathbf{x}$ a ležela v rovině trojúhelníku.

$$\mathbf{y} = \frac{\mathbf{x} \times (\mathbf{v_3} - \mathbf{v_1})}{|\mathbf{x} \times (\mathbf{v_3} - \mathbf{v_1})|} \times \mathbf{x}. \tag{3.2}$$

2. Vrcholy trojúhelníku v tomto novém prostoru mají souřadnice $\mathbf{v_1} = [0; 0]$, $\mathbf{v_2} = [1; 0]$ a $\mathbf{v_3} = [c_1; c_2]$, kde

$$c_1 = \frac{(\mathbf{v_3} - \mathbf{v_1}) \cdot \mathbf{x}}{|\mathbf{x}|^2},\ c_2 = \frac{(\mathbf{v_3} - \mathbf{v_1}) \cdot \mathbf{y}}{|\mathbf{y}|^2}. \tag{3.3}$$

   Podle [20] spočteme souřadnice středu kružnice opsané v našem dvourozměrném prostoru:

$$s_1 = \frac{1}{2},\ s_2 = \frac{c_1^2 + c_2^2 - c_1}{2c_2}. \tag{3.4}$$

3. Střed kružnice v původním trojrozměrném prostoru spočteme podle následujícího vztahu:

$$\mathbf{s} = \mathbf{v_1} + s_1\mathbf{x} + s_2\mathbf{y} \tag{3.5}$$

   a poloměr je roven

$$r = |\mathbf{s} - \mathbf{v_0}|. \tag{3.6}$$

Pro vnitřní uzly stromu je obálková koule vypočtena z obálkových koulí jejich dceřiných uzlů. V první řadě je nutné zkontrolovat, zda jedna z dceřiných koulí neobsahuje všechny ostatní dceřiné koule. V tomto případě se stává tato koule obálkovou koulí nového uzlu. Tento případ nastává velmi často, pokud ve ve vstupní síti nachází trojúhelníky různých velikostí.

Pokud není nalezena koule, která obklopuje všechny ostatní, je nutné novou obálkovou kouli sestrojit. Existují tři možnosti:

1. Dva dceřiné uzly. Střed nové koule leží na spojnici středů koulí dceřiných. Poloměr je spočten podle vzorce:

$$r = \frac{d + r_1 + r_2}{2}, \tag{3.7}$$

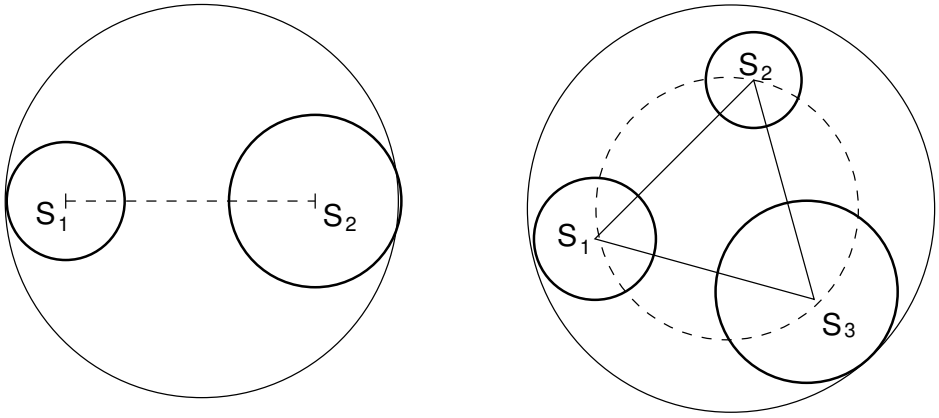
   kde r_1 a r_2 jsou poloměry dceřiných obálkových koulí a d je vzdálenost středů obálkových koulí. Střed nové obálkové koule se vypočítá podle následujícího vztahu:

$$S\ = S_1 + (r - r_1) \cdot \frac{S_2 - S_1}{d}, \tag{3.8}$$

   kde S_1 a S_2 jsou středy původních obálkových koulí.

2. Tři dceřiné uzly. Středy dceřiných obálek tvoří trojúhelník, pro který spočteme obálkovou kouli stejným způsobem jako pro koncové uzly. Výsledná obálková koule je tato koule s poloměrem zvětšeným o velikost poloměru největší dceřiné koule.

3. Čtyři dceřiné uzly. Středy koulí tvoří čtyřstěn — pro každou stěnu tohoto čtyřstěnu vypočteme jednoho kandidáta na obálkovou kouli. Nejprve pro stěnu vypočteme obálkovou kouli jako v případě tří koulí, ve druhém kroku vypočteme obálkovou kouli pro kouli takto získanou a obálkovou kouli ve čtvrtém vrcholu čtyřstěnu. Ze čtyř získaných obálkových koulí vybereme nakonec tu s nejmenším poloměrem.



Obrázek 3.3: Výpočet obálkové koule pro dva a tři dceřiné uzly.

3.3 Zobrazování

Zobrazování probíhá podobně jako v metodě QSplat. Algoritmus rekurzivně prochází strom a vybírá vhodné uzly, které zobrazí. Pokud má být zobrazen vnitřní uzel, vykreslí se splat, pokud se při průchodu stromu dostane algoritmus až do koncového uzlu a velikost průmětu uzlu je vetší než zvolený práh, je vykreslen trojúhelník. Během procházení stromu se samozřejmě používají techniky k eliminaci neviditelných částí stromu:

- Pokud se uzel nachází zcela mimo pohledovou pyramidu, dále se neprochází a nic není vykresleno. Naopak pokud se nachází celý uvnitř pohledové pyramidy, tento test se už neprovádí pro podstrom tohoto uzlu.
- Pokud všechny normály ukazují směrem pryč od pozorovatele, uzel se dále neprochází a nic není vykresleno. Naopak pokud všechny normály ukazují směrem k pozorovateli, není nutné tento testo provádět pro uzly ležící ve stromě pod uzlem testovaným.

Trojúhelníky lze vykreslit přímo, ale pro vykreslení bodů je potřeba použít jednu z rekonstrukčních technik uvedených výše — buď splatting jako v QSplat [13], nebo visibility splatting, který je použitý v algoritmu Surfels [11].

3.3.1 Jednoduchý rekurzivní algoritmus

Jednoduchý algoritmus pro zobrazování prochází strom od kořene směrem k listům. U každého uzlu mohou nastat tři možnosti:

1. Uzel není vůbec vidět. Nic se nevykreslí a rekurze zde končí.
2. Velikost splatu je menší než práh. Vykreslí se splat a rekurze končí.
3. Velikost splatu je větší než požadovaný práh, ale uzel je listem. Vykreslí se trojúhelník v něm obsažený.
4. Velikost splatu je větší než požadovaný práh a jedná se o vnitřní uzel. Strom se dále rekurzivně prochází.

Na volbě hodnoty prahu závisí výsledná kvalita a rychlost zobrazení. Čím menší je, tím kvalitnější obraz vznikne, ale proces zobrazení je pomalejší. Pro kvalitní výstup se používá práh velikosti jednoho pixelu. Formální zápis algoritmu:

```
Render(node)
{
    if (not node.isVisible())
    {
        // uzel je označen a přeskočen
        node.active := 1
    }
    else if (node.isLeaf())
    {
        // pro koncový uzel je vykreslen trojúhelník
        DrawTriangle(node)
        // uzel je též označen
        node.active := 1
    }
    else if (node.splatsize < threshold)
    {
        // vykreslí se splat
        DrawSplat(node)
        node.active := 1
    }
    else
    {
        // rekurze
        foreach (child in node.children)
        {
            Render(child)
        }
    }
}
```

Atribut *active* uložený v uzlu je použit pro využití koherence (viz níže).

3.3.2 Rekurzivní algoritmus s využitím koherence

Jednou z časově nejnáročnějších operací je procházení stromové struktury. Proto lze algoritmus zdokonalit tím, že budeme procházet menší množství uzlů. Nebudeme pokaždé začínat s procházením v kořenovém uzlu, ale v uzlech, o kterých bylo př předchozím zobrazení rozhodnuto, že budou buď zahozeny, nebo přímo zobrazeny (příznak *active* v předchozím algoritmu). V závislosti na velikosti splatů procházíme strom od těchto uzlů buď směrem nahoru nebo dolů. Tím je efektivně využito koherence mezi jednotlivými záběry animace.

Algoritmus pro zobrazování s využitím koherence je následující:

```
RenderCoherency(node)
{
    if (node.active = 1)
    {
        // uzel byl označen jako startovní
        // při posledním zobrazení
        if (node.splatsize > threshold
            and node.isVisible())
        {
            if (node.isLeaf())
            {
                DrawTriangle(node)
            }
            else
            {
                // splat je příliš velký -> rekurze
                node.active := 0
                Render(node)
            }
        }
        else
        {
            // splat je příliš malý
            // budeme se vracet nahoru, dokud nebude
            // mít splat správnou velikost
            node.active := 0
            n := node
            while ( n != NULL )
            {
                // cyklus se zastaví nejdále v~kořenu
                child := n
                n := n.parent
                if (n != NULL)
                {
                    if (n.stop)
                    {
                        break
                    }
                    if (n.splatsize > threshold
                        and n.isVisible())
                    {
                        n.stop := true
                        break
                    }
                    else
                    {
                        n.visited := 1
```

```
                        }
                    }
                }
                child.active := true
                if (child.isVisible())
                {
                    // vykreslíme splat
                    DrawSplat (child)
                }
            }
        }
        else
        {
            // uzel není označen jako startovní
            // postupujeme rekurzí níže
            node.visited := 0
            foreach (child in node.children)
            {
                RenderCoherency (child)
                if (node.visited = 1)
                {
                    // uzel byl již zpracován zpětným
                    // chodem -> konec rekurze
                    break
                }
            }
            node.stop := false
        }
    }
```

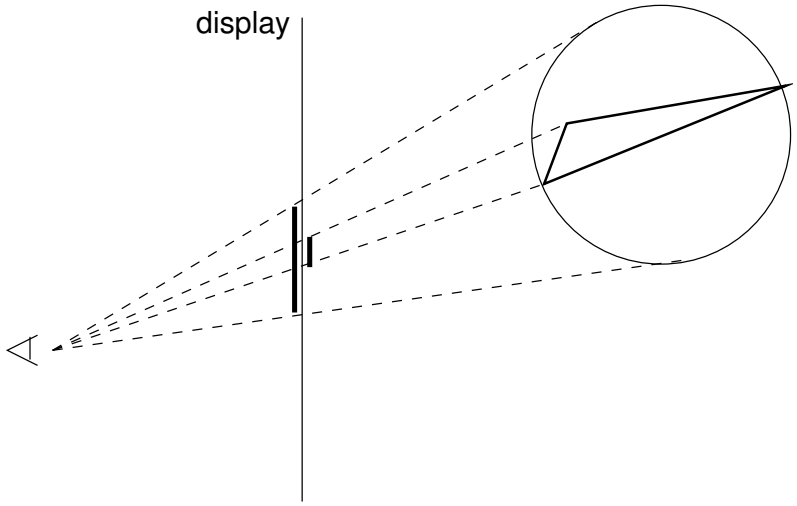
Koherentnní algoritmus nejprve rekurzivně prohledává strom od kořene, dokud nenarazí na uzel, u kterého došlo k rozhodnutí při minulém vykreslení (nastavený příznak *active*). Terpve od tohoto uzlu začne prohledávát strom s testováním velikosti průmětu a viditelnosti. V závislosti na aktualní velikosti průmětu uzlu začne prohledávat buď směrem dolů, nebo nahoru. Prohledávání dolů obstará jednoduchá funkce *Render*. Během prohledávání směrem nahoru se u uzlů nastavují příznaky *visited*, aby se zabránilo opakovanému prohledávání nekterých částí stromu. Příznak *stop* se nastaví v uzlu, u kterého dojde k rozhodnutí, zda bude vykreslen, nebo zahozen (nový *active* uzel) — tím se zabrání opakovanému testu uzlu při dalších zpětných chodech.

3.3.3 Výpočet velikosti splatů

Při průchodu stromu je nutné v každém uzlu spočítat velikost jeho splatu. Protože se tato operace provádí velmi často, je nutné najít optimální způsob, jak velmi rychle spočítat dobrou aproximaci velikosti průmětu surfelu. Přesný vý-

počet velikosti a tvaru splatu by byl velmi časově náročný a v podstatě ani není možný, protože již obálková koule je aproximací.

Na obrázku (3.4) je ilustrováno, k jaké chybě dochází při použití obálkových koulí. Tato chyba je zejména zřejmá u protáhlých trojúhelníků nebo trojúhelníků, jejichž normála je téměř kolmá k pohledovému paprsku. Důsledkem této skutečnosti je snížení kvality výsledného obrazu. Možným řešením by bylo použít obálkové elipsoidy místo kulových ploch, ale to by neúměrně zvýšilo výpočetní náročnost. Dalším řešením může být snížení prahu, čímž se dosáhne větší kvality za cenu pomalejšího zobrazení.



Obrázek 3.4: Chyba vznikající při průmětu koule může být značná.

Výpočet velikosti splatu jako přesný průmět obálkové koule je příliš časově náročný. Prvního urychlení lze dosáhnout tak, že se bude počítat velikost splatu pouze na základě hodnoty souřadnice z polohového vektoru koule. Pokud se koule nachází na souřadnici z, lze velikost průmětu koule aproximovat velikostí průmětu jejího průměru. Pro zjednodušení budeme dále pracovat pouze s poloměrem kolmým na pohledový paprsek. Myšlenka výpočtu je ilustrována na obrázku (3.5). Velikost průmětu poloměru koule A, jejíž střed leží na ose z, je rovna

$$r_A = \frac{dR}{z}, \tag{3.9}$$

kde d je vzdálenost pozorovatele od průmětny, R je velikost poloměru koule a z je vzdálenost koule od roviny pozorovatele. Výpočet průmětu poloměru koule, která neleží na ose z je složitější. Velikost úsečky a spočteme z trojúhelníku:

$$a = \frac{R}{\cos\phi}, \tag{3.10}$$

kde ϕ je velikost úhlu mezi spojnicí pozorovatele se středem koule a osou z. Z trojúhelníkové nerovnosti platí

$$\frac{r_A}{R} = \frac{r_B}{a}, \tag{3.11}$$

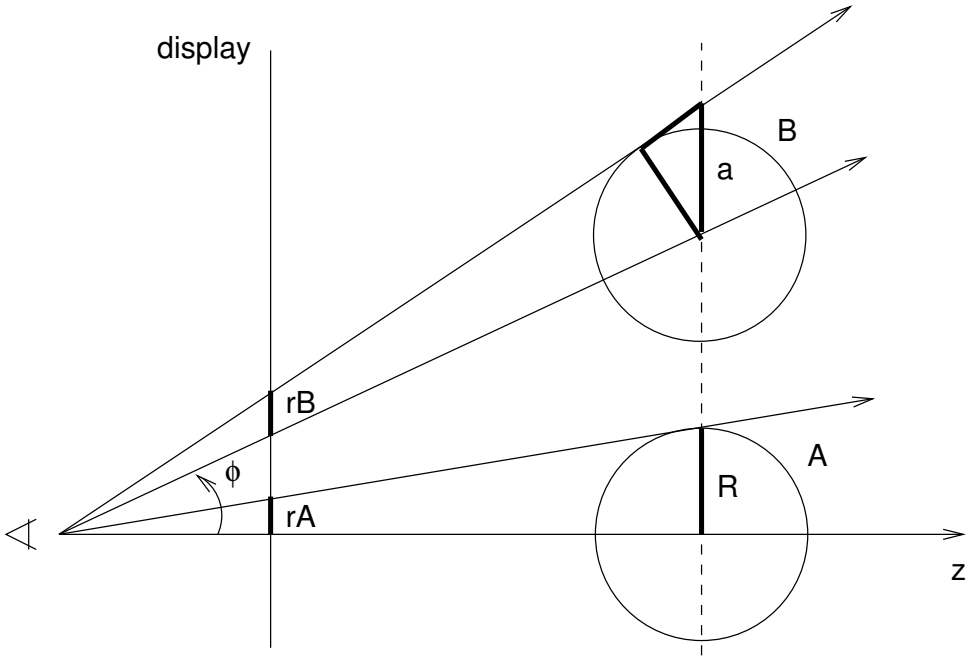
z čehož lze snadno vyvodit, že

$$r_B = \frac{r_A a}{R} = \frac{r_A}{\cos\phi}. \tag{3.12}$$

Při výpočtu průmětu lze zanedbat úhel ϕ a počítat velikost průmětu poloměru koule pouze ze vztahu

$$r' = r_A = \frac{dR}{z}. \tag{3.13}$$

Tato velikost je větší než přesněji spočtená velikost r_B, což znamená, že průchod stromem bude ukončen dříve. To sice vede ke zhoršení kvality výstupu, nicméně si tuto aproximaci můžeme dovolit, protože chyba vzrůstá až u okrajů obrazu, které nejsou pro pozorovatele tolik důležité.



Obrázek 3.5: Výpočet aproximace velikosti průmětu koule.

3.3.4 Vykreslování splatů

Pro zobrazení surfelů lze použít několik rekonstrukčních technik. Nejjednodušší je použít splatting s konstantní barvou podobně jako v systému QSplat [13]. To lze velmi jednoduše implementovat pomocí OpenGL primitiv `GL_POINT`. Nicméně použití této techniky vede k méně kvalitním výstupům, dobrých výstupu se dosahuje až pro práh roven velikosti jednoho pixelu.

Druhou možností je použít visibility splatting tak jak je použit v metodě Surfels [11]. Tato metoda poskytuje stejnou výstupní kvalitu jako obyčejný splatting při použití daleko většího prahu. Nevýhodou je, že neexistuje pro tuto techniku přímá podpora v hardwaru. Omezenou hardwarovou akceleraci lze získat dvouprůchodovým zobrazením. V prvním průchodu jsou splaty vykresleny pouze barvou pozadí, důležité je, že je hloubková informace zapsána do z-bufferu. Při druhém průchodu jsou vykresleny pouze pixely do středu splatů. Následně je provedena rekonstrukce obrazu Gaussovským filtrem, poloměr filtru je pro celý obraz konstantní a jeho velikost je rovna prahu pro velikost splatů.

Kapitola 4

Implementace

Implementace Point Based Rendering algoritmu byla provedena v jazyce *C++* v prostředí *Microsoft Visual Studio .NET* pod operačním systémem *Microsoft Windows XP Professional*. Pro veškeré grafické výstupy bylo použito standardní grafické API *OpenGL*[1] a knihovna *Glut*[2]. Toto prostředí bylo zvoleno pro své uživatelské pohodlí a širokou podporu a rozšíření.

4.1 Vstup

Vstupem aplikace je libovolná trojúhelníková síť. Tato síť je načtena ze souboru ve standardním formátu *TRI*[3]. Toto je textový formát, ve kterém je uložena pouze geometrie objektu (resp. scény). Povinné části TRI souboru jsou `Vertices` (souřadnice vrcholů), `Vertices' Normals` (normály ve vrcholech) a `Triangles` (které vrcholy tvoří trojúhelníky). Aplikace ignoruje veškeré další informace obsažené v souboru (např. normály trojúhelníků).

4.2 Výstup

Výstupem je zobrazení trojúhelníkové sítě v okně aplikace použitím různých zobrazovacích metod. Aplikace umožňuje zadat různé parametry pro zobrazení a pro srovnání také umožňuje zobrazit model třemi způsoby: klasicky (vykreslení celé trojúhelníkové sítě), pouze vrcholy sítě (pro srovnání s PBR) a metodou POP. Na požádání aplikace vypíše statistiky o zobrazení, které jsou použity pro srovnání Point Based Rendering a klasického zobrazování.

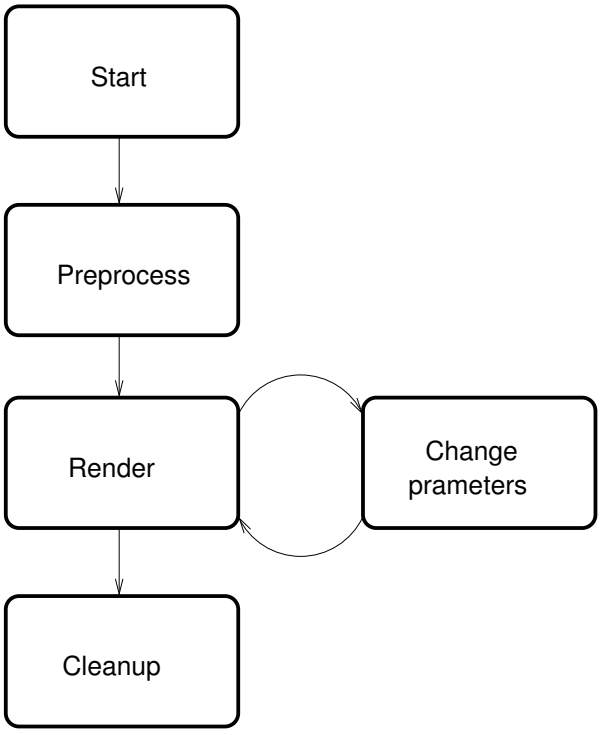
[1] Viz http://www.opengl.org.
[2] Viz http://www.opengl.org/resources/libraries/glut.html.
[3] Viz http://herakles.zcu.cz/research/mveruntime/mve/developer/tri10.php.

4.3 Životní cyklus aplikace

Životní cyklus aplikace lze přehledně shrnout do následujících kroků:

1. Načtení vstupního souboru. Vstupní trojúhelníková síť se načte celá do paměti. Zvlášť jsou uloženy souřadnice jednotlivých vrcholů, souřadnice normál ve vrcholech a topologické informace (které vrcholy spolu tvoří hrany a trojúhelníky).
2. Výpočet hierarchického LOD stromu. Ze vstupní trojúhelníkové sítě se vypočte stromová struktura tak jak je uvedeno výše. Celá tato datová struktura je uložena v operační paměti.
3. Zobrazení modelu. Model se zobrazí podle nastavených parametrů.
4. Změna parametrů zobrazení. Uživatel změní parametry zobrazení, nebo ukončí aplikaci. Pokud aplikaci neukončí, přechází se na krok *Zobrazení modelu*.
5. Ukončení. Datové struktury jsou uvolněny z paměti a aplikace je ukončena.

Životní cyklus je přehledně zobrazen v obrázku (4.1).



Obrázek 4.1: Životní cyklus aplikace.

4.4 Datové struktury

Z hlediska PBR metody je důležitou datovou strukturou pouze uzel LOD stromu — `POPNode`. Tato struktura obsahuje:

1. Střed a poloměr obálkové koule.
2. Kosinus úhlu, pod kterým je ještě uzel viditelný (úhel normálového kužele plus 90^o).
3. Ukazatel na rodičovský uzel.
4. Počet dceřiných uzlů a pole ukazatelů na ně.
5. Příznaky *active*, *visited* a *stop*.
6. Index do pole vrcholů středů obálkových koulí.
7. Tři indexy do pole vrcholů sítě tvořící trojúhelník (pokud je uzel listem).

Na paměťové nároky nebyl kladen příliš velký důraz, proto uzel zabírá poměrně dost paměti (64 bajtů, což je přibližně osmkrát více než v metodě QSplat [13]). Spoustu paměti by šlo ušetřit kvantizací hodnot a eliminací nadbytečných atributů — např. počet dceřiných uzlů nebo pole indexů vrcholu trojúhelníku, pokud uzel není listem.

4.5 Využití hardwarové akcelerace

Pro veškeré zobrazování je použita knihovna OpenGL. Při zobrazování není použito texturování, objekt je pouze vystínován jednoduchým osvětlovacím modelem. Pro výpočet barvy je použit standardní Phongův osvětlovací model a pro vystínování trojúhelníků Gouraudovo stínování. Pro odstranění neviditelných částí scén je použit klasický z-buffer.

Splatting je implementován pouze tou nejjednodušší metodou — splatting pomocí splatů čtvercového tvaru a konstantní barvy. Splaty tohoto typu lze velmi jednoduše vykreslovat přes grafickou knihovnu OpenGL, která přímo podporuje vykreslování takovýchto primitiv. Použije se OpenGL objekt `GL_POINT` a pomocí funkce `glPointSize` se nastaví potřebná velikost zobrazovaných bodů.

Protože se zobrazují velmi rozsáhlé množiny dat (až několik miliónů trojúhelníků, resp. bodů) je nutné brát zvláštní zřetel na způsob, jakým jsou zobrazovaná data poskytnuta grafickému hardwaru. OpenGL poskytuje tři tyto možnosti:

- Immediate Mode — Objekty jsou předávány po jednotlivých primitivech. Velmi nevhodný způsob pro zobrazování rozsáhlých množin primitiv.
- Vertex Array — Geometrická data jsou předána jedním voláním, ale jsou uložena v klientské paměti a proto není možné využít paralelismu mezi CPU a GPU.

- Display List — Model je „předkompilován“, může být tedy uložen v paměti grafického hardware a zobrazení je velmi rychlé. Nevýhodou je, že geometrie takto uložená nemůže být změněna.
- Vertex Buffer Object — Data jsou ukládána přímo do paměti grafického hardware a je umožněno dynamicky tyto data měnit. Tato metoda je dostupná pouze u moderního grafického hardwaru a v nejnovějších verzích OpenGL knihovny.

Vertex Buffer Objects by byly pro zobrazení nejvhodnější, ale nejsou ještě široce podporovány na běžném hardwaru. Navíc bychom byli pri použití VBO limitováni velikostí paměti na grafické kartě, což by představovalo problém pro modely, které mají řádově miliony a více trojúhelníků. Z běžné dostupných metod je nejrychlejší Display List, ale protože se body a trojúhelníky generují dynamicky, nelze Display List použít.

Pro přesun dat do grafického hardwaru byly tedy použity Vertex Arrays. To znamená, že při zobrazování hierarchické LOD struktury jsou zobrazena primitiva ukládána do pole a až nakonec je veškerá geometrie přesunuta do grafického hardwaru jedním příkazem.

Kapitola 5

Rozbor výsledků

V první části této kapitoly se budeme věnovat odhadu jak časové, tak paměťové složitosti. Tyto očekávané hodnoty porovnáme s experimentálně naměřenými hodnotami uvedenými ve druhé části.

Ve druhé části rozboru výsledků se zejména zaměříme na porovnání časové náročnosti konvenční polygonální zobrazovací metody a Point Based Renderingu. V neposlední řadě se také budeme věnovat porovnání kvality výstupů obou metod. Nicméně kvalita výstupů se nejlépe zhodnotí přímo při použití programu, který je součástí této práce. Rozebrána bude také samotná metoda PBR — budeme zkoumat kolik času potřebují jednotlivé fáze implementovaného PBR algoritmu.

Veškeré experimenty byly provedeny na osobním počítači s procesorem *AMD Athlon XP 1800+*, operační pamětí o velikosti 1 GB a grafickým akcelerátorem *Nvidia GeForce Ti 4200* se 128 MB video paměti.

5.1 Algoritmická složitost

Jelikož je algoritmus rozdělen na dvě části — předzpracovaní a zobrazování — musíme určit časovou složitost zvlášť pro obě části.

V předzpracování probíhá výpočet stromové datové struktury. Jedná se o iterativní proces, kdy v každém kroku iterace je vypočtena jedna úroveň stromu. Výpočet jedné úrovně je v podstatě velmi podobný algoritmu *quicksort*, takže jeho složitost je $O(n \log n)$. Počet iterací je roven hloubce stromu a ta odpovídá vztahu $\log n$. V každém iteračním kroku se však zpracovává pouze čtvrtinový počet uzlů než v kroku předchozím. Složitost celého procesu lze vyjádřit vzorcem:

$$O\left(n \log n + \frac{n}{4} \log \frac{n}{4} + \frac{n}{16} \log \frac{n}{16} + \ldots + \frac{n}{n} \log \frac{n}{n}\right). \tag{5.1}$$

Tento vztah lze omezit shora vztahem jednodušším — očekávaná časová složitost je tedy přibližně

$$O(\frac{4}{3} n \log n) \approx O(n \log n), \tag{5.2}$$

kde n je počet trojúhelníků vstupního modelu.

Průměrná časová složitost zobrazování se velmi špatně odhaduje, protože počet vykreslených primitiv silně závisí na poloze kamery a jiných parametrech zobrazení. V nejhorším případě se projde celý strom a vykreslí se všechny koncové uzly, kterých je n. Protože zpracování jednoho uzlu vyžaduje konstantní počet operací, je horní odhad časové složitosti vykreslení scény $O(n)$, kde n je počet trojúhelníků vstupního modelu.

5.2 Paměťová složitost

Podobně jako při určování algoritmické složitosti je potřeba uvést zvlášť paměťovou složitost jak pro předzpracování, tak pro samotné vykreslování.

Výsledná datová struktura je strom, který má přibližně $\log_4 n$ úrovní (n je počet trojúhelníků na vstupu). Jelikož paměťové náklady na uložení jednoho uzlu jsou konstantní a při předzpracování se nepoužívají žádné rozsáhlé pomocné datové struktury, je průměrná paměťová náročnost rovna $O(n)$, kde n je počet trojúhelníků vstupního modelu. Je dobré si uvědomit, že komprimační techniky uvedené v teoretické části v žádném případě nesnižují tuto paměťovou náročnost.

Když pomineme přítomnost LOD stromu v paměti, je paměťová náročnost algoritmu vykreslování dána velikostí použitého zásobníku pro rekurzi. Tato velikost je shora omezena počtem úrovní stromu, kterých je $\log_4 n$. Horní odhad paměťové náročnosti zobrazovacího algoritmu je tedy $O(\log n)$, kde n je počet trojúhelníků vstupního modelu.

5.3 Použité modely

Tabulka (5.1) obsahuje přehled modelů použitých pro experimenty a měření. V tabulce se nachází jméno modelu, počet vrcholů, počet trojúhelníků a náhled.

Model `horse.tri` reprezentuje síť s malým počtem trojúhelníků. Trojúhelníkové sítě `hand.tri` a `dragon.tri` se skládají s poměrně velkého množství trojúhelníků a představují data z reálných aplikací. Největší počet trojúhelníků mají modely `sphere.tri` a `torus.tri`, které vznikly navzorkováním parametrických funkcí.

5.4 Měření časů

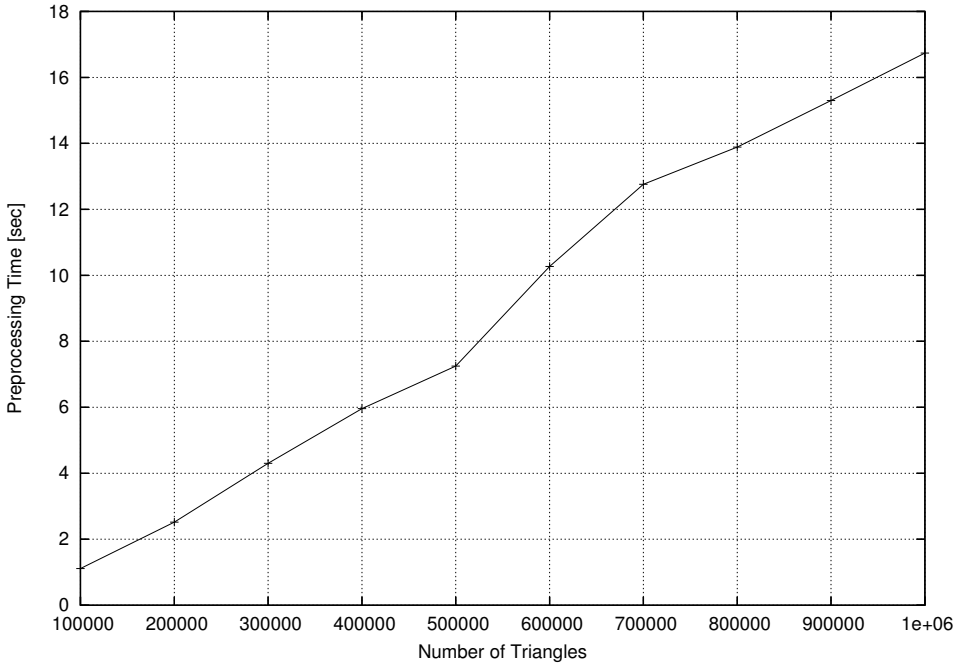
Pokud není uvedeno jinak, platí ve všech následujících měřeních tyto hodnoty:

- Rozlišení okna je 500×500 pixelů.
- Velikost splatů (a práh pro jejich vykreslení) je rovna třem.

- Je použit algoritmus eliminace částí scény odvrácených od pozorovatele.

5.4.1 Předzpracování

Tabulka (5.2) ukazuje dobu predzpracování (vytváření LOD stromu) u jednotlivých modelů a graf (5.1) závislost doby předzpracování na počtu trojúhelníků vstupní sítě. Měření závislosti uvedené v grafu probíhalo na jednotkovém čtverci navzorkovaném postupně na 100000, 200000 až 1000000 trojúhelníků.



Obrázek 5.1: Graf závislosti doby předzpracování na počtu trojúhelníků.

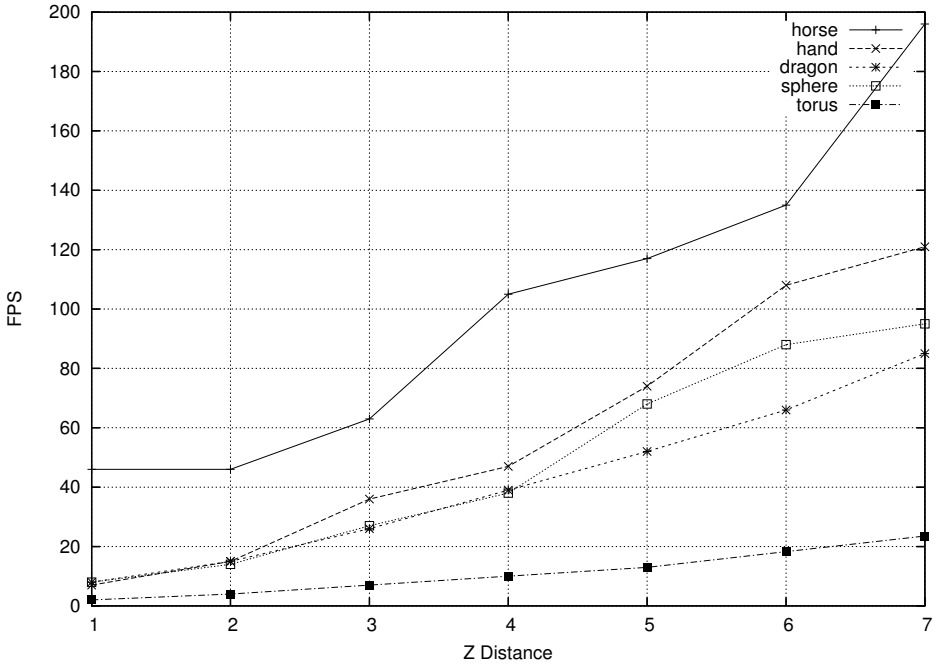
5.4.2 Rychlost vykreslování

Protože předzpracování se provádí pouze jednou, jeho časová náročnost není tak důležitá. Daleko důležitější je rychlost vykreslování již předzpracovaných modelů. V tabulce (5.3) je uveden počet snímků za sekundu a počet zobrazených trojúhelníků a bodů pro jednotlivé modely v závislosti na vzdálenosti od kamery.

Pro porovnání je uvedena v tabulce (5.4) rychlost vykreslení konvenční metodou. Počet snímků za sekundu se neměnil se vzdáleností od kamery, proto je u každého modelu uvedena pouze jedna hodnota.

Pro přehlednost jsou uvedeny grafy zobrazující data z výše uvedené tabulky. Graf (5.2) ukazuje závislost rychlosti vykreslování na vzdálenosti od pozorovatele. Grafy (5.3) a (5.4) pak počet vykreslených trojúhelníků a bodů.

V tabulce (5.5) jsou uvedeny hodnoty počtu snímků za sekundu a počtu vykreslených primitiv v závislosti na na velikosti prahu pro vykreslení splatů. Hodnoty z tabulky jsou zobrazeny v grafech (5.5), (5.6) a (5.7).



Obrázek 5.2: Graf závislosti počtu snímků za sekundu na vzdálenosti od pozorovatele.

5.5 Měření paměťové náročnosti

Tabulka (5.6) ukazuje velikost potřebé paměti pro jednotlivé modely. V grafu (5.8) je pak vynesena závislost velikosti spotřebované paměti na počtu trojúhelníků vstupního modelu. Paměťová náročnnnost je vyjádřena v počtu uzlů LOD stromu. Měření závislosti vynesené v grafu probíhalo na jednotkovém čtverci navzorkovaném postupně na 100000, 200000 až 1000000 trojúhelníků.

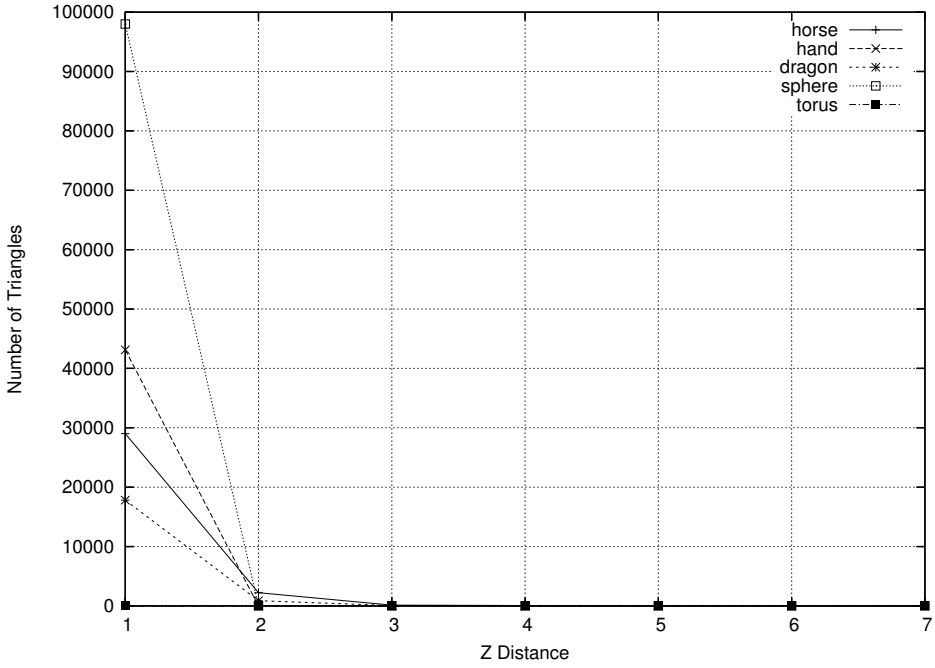
5.6 Diskuze výsledků

5.6.1 Předzpracování

Doba předzpracování je přímo úměrná počtu trojúhelníků vstupního modelu — nejlépe je to patrné z grafu (5.1). Naměřené výsledky tedy odpovídají očekávané teoretické složitosti $O(n)$.

5.6.2 Rychlost vykreslování

Rychlost vykreslování klasickou metodou je nezávislá na poloze kamery — nemění se při oddalování a přibližování modelu. To je dáno tím, že časově nejnákladnější částí vykreslování je přesun dat z operační paměti do grafického akcelerátoru. Samotný výpočet osvětlení a následná rasterizace probíhá pralelně během



Obrázek 5.3: Graf závislosti počtu zobrazených trojúhelníků na vzdálenosti od pozorovatele.

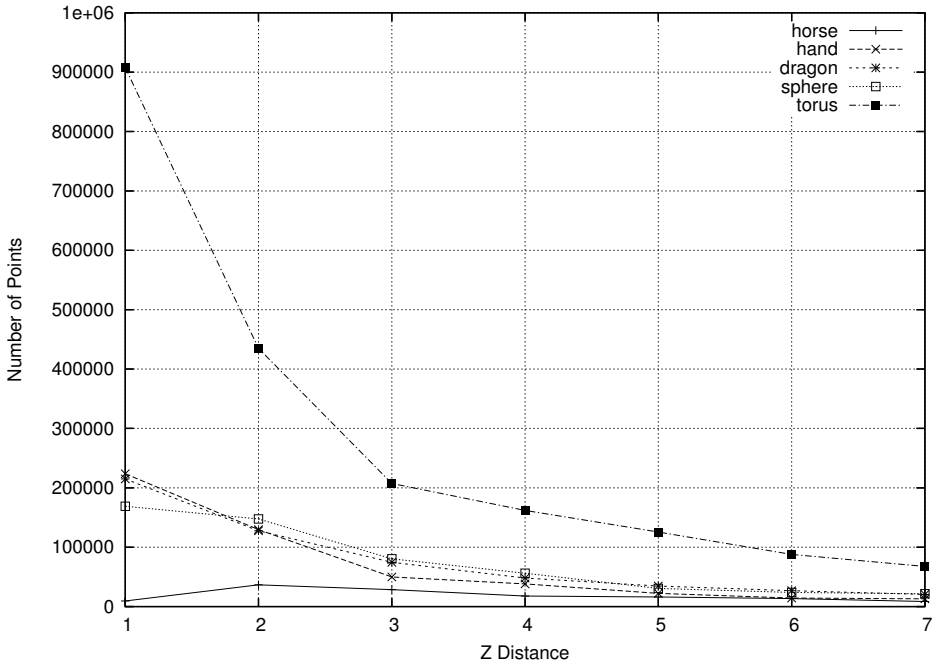
přenosu dat.

Na druhou stranu vykreslování metodou POP je silně závislé na poloze kamery — ta totiž určuje, kolik primitiv bude vykresleno. Počet snímků za sekundu je přímo úměrný vzdálenosti objektu od pozorovatele a nepřímo úměrný počtu vykreslených primitiv. K nejvýraznějšímu urychlení oproti klasické metodě dochází u modelů s velikým počtem trojúhelníků. Horší výsledky u modelu `torus` jsou dány tím, že tento model obsahuje mnoho trojúhelníků s velkým tupým úhlem u jednoho z vrcholů a výsledný LOD strom není zdaleka optimální.

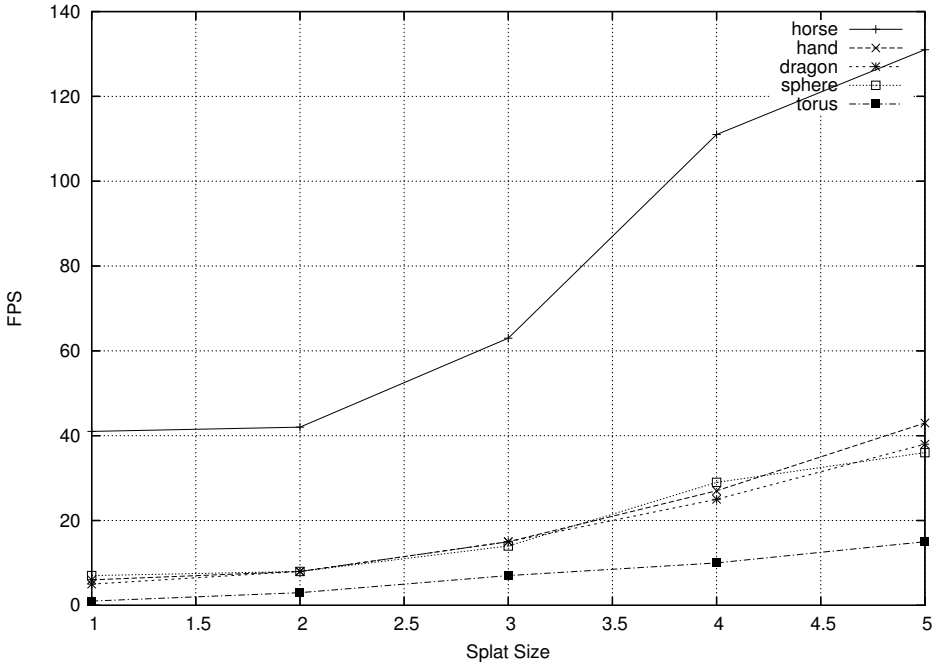
Vliv zvyšování velikosti prahu má prakticky stejný význam jako oddalování modelu — v obou případech dochází k vykreslování modelu v menším rozlišení. Proto jsou odpovídající tabulky a grafy velmi podobné grafům zobrazujícím závislost rychlosti zobrazenní na vzdálenosti od pozorovatele.

5.6.3 Paměťové nároky

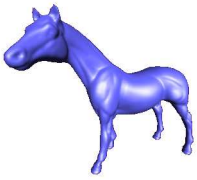
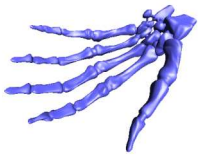
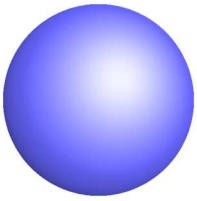
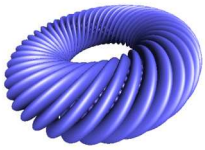
V tabulce (5.6) a grafu (5.8) je vynesen počet uzlů LOD stromu v závislosti na počtu trojúhelníků výchozího modelu. Paměťová náročnost je přímo úměrná počtu uzlů, protože jeden uzel zabírá konstantní množství paměti. Z tabulky i grafu vychází, že paměťové nároky jsou přímo úměrné velikosti sítě, což potvrzuje teoretický předpoklad — paměťová náročnost $O(n)$.



Obrázek 5.4: Graf závislosti počtu zobrazených bodů na vzdálenosti od pozorovatele.



Obrázek 5.5: Graf závislosti počtu snímků za sekundu na velikosti splatů.

#	Název	Počet vrcholů	Počet trojúhelníků	Náhled
1	horse.tri	48485	96966	
2	hand.tri	327323	654666	
3	dragon.tri	437645	871414	
4	sphere.tri	501501	1000000	
5	torus.tri	2020101	4000000	

Tabulka 5.1: Modely použité pro experimenty a měření.

Model	Doba předzpracování [s]
1	1,05
2	11,71
3	16,37
4	16,58
5	104,8

Tabulka 5.2: Čas potřebný pro předzpracování a zobrazení.

	z	1	2	3	4	5	6	7
Model 1	FPS	46	46	63	105	117	135	196
	△	29014	2229	146	76	55	47	37
	•	9507	36644	28447	17670	16061	13580	8476
Model 2	FPS	7,27	15,84	36,63	47	74	108	121
	△	43118	0	0	0	0	0	0
	•	223709	129706	49826	38095	22243	14339	13108
Model 3	FPS	8,33	15,53	26,21	39	52,48	66	85
	△	17822	882	58	3	0	0	0
	•	215049	127486	74497	48007	34533	26762	20382
Model 4	FPS	8,82	14,85	27,72	38,24	68	88	95
	△	98001	0	0	0	0	0	0
	•	169007	147646	80378	56204	30542	23865	21669
Model 5	FPS	2,24	4,17	7,84	10	13	18.27	23.53
	△	0	0	0	0	0	0	0
	•	907676	434868	207231	161845	125540	87903	67515

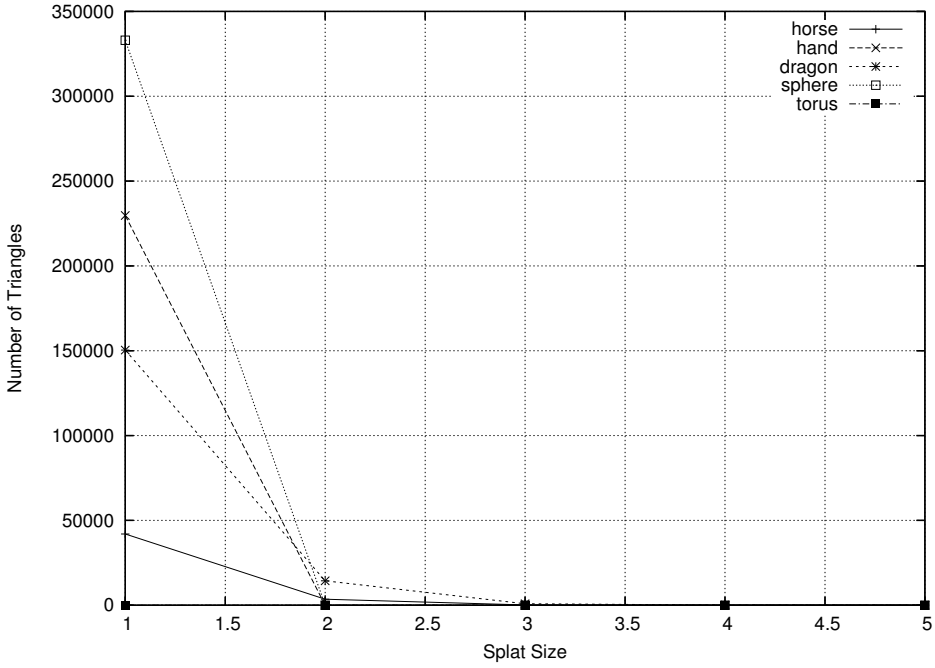
Tabulka 5.3: Počet snímků za sekundu a počet vykreslených primitiv pro různé vzdálenosti od kamery.

Model	FPS
1	70
2	10,9
3	8,25
4	7,1
5	1,8

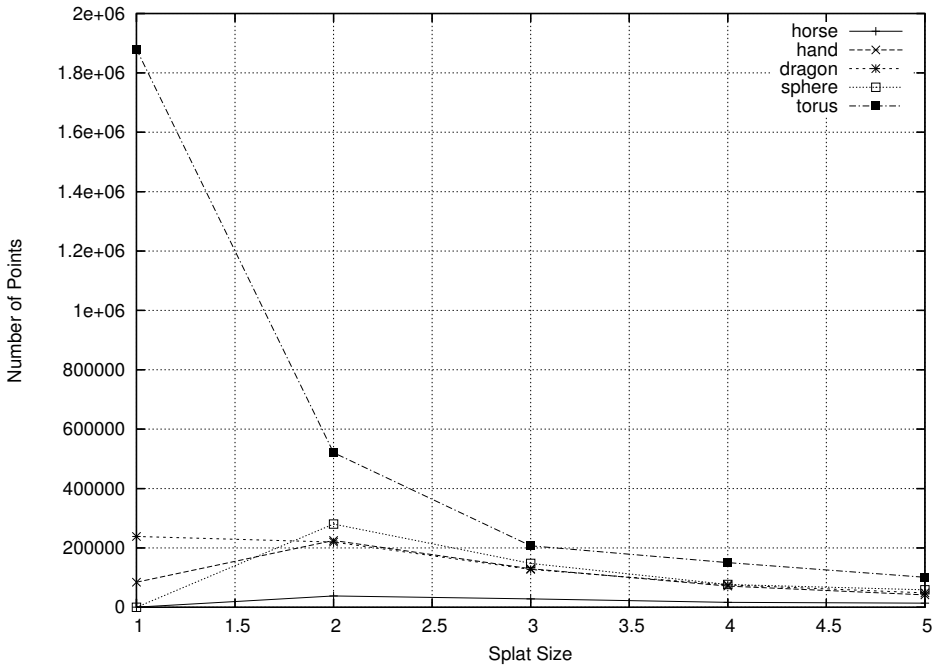
Tabulka 5.4: Počet snímků za sekundu při použití klasické metody.

	Práh	1	2	3	4	5
Model 1	FPS	41,58	42,16	63	111	131
	△	41959	3564	146	64	44
	•	476	38383	28447	16735	13967
Model 2	FPS	6,09	8,65	15,84	27,18	43,56
	△	229706	32	0	0	0
	•	84268	225187	129706	71514	41560
Model 3	FPS	5,04	8,41	15,53	25,74	38,24
	△	150355	14382	882	64	7
	•	238495	219683	127486	75285	48586
Model 4	FPS	7,08	8,26	14,85	29	36,63
	△	332996	0	0	0	0
	•	49	280443	147646	76935	58697
Model 5	FPS	1,2	3,54	7,84	10,78	15,84
	△	0	0	0	0	0
	•	1879977	520579	207231	150517	101069

Tabulka 5.5: Počet snímků za sekundu a počet vykreslených primitiv pro různé velikosti prahu pro ukončení procházení stromu.



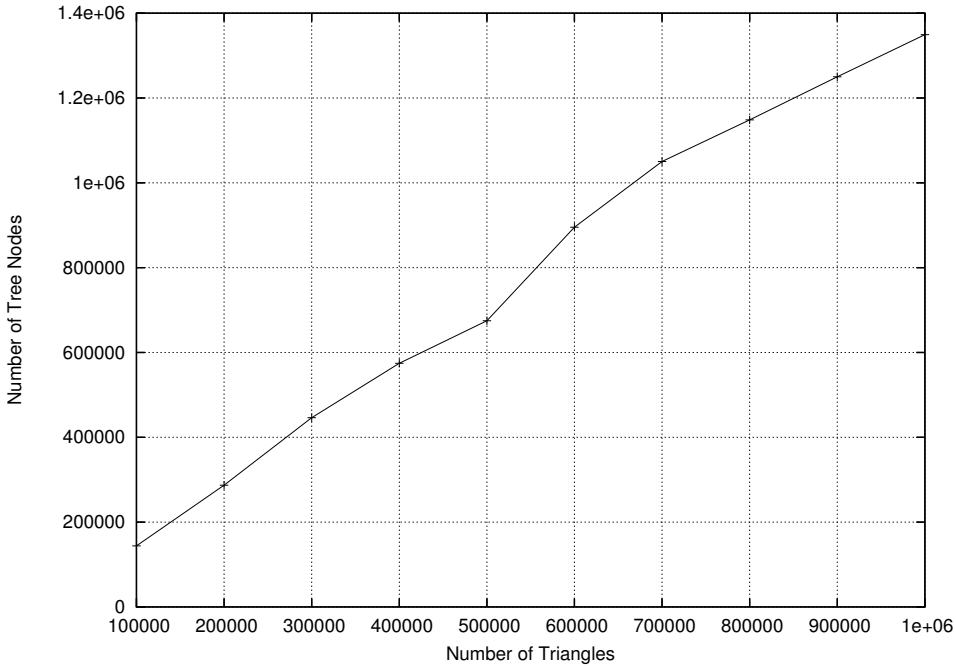
Obrázek 5.6: Graf závislosti počtu zobrazených trojúhelníků na velikosti splatů.



Obrázek 5.7: Graf závislosti počtu zobrazených bodů na velikosti splatů.

Model	Počet uzlů stromu
1	140657
2	1003497
3	1220939
4	1349525
5	5398101

Tabulka 5.6: Počet uzlů LOD stromu.



Obrázek 5.8: Graf závislosti počtu uzlů LOD stromu na velikosti sítě.

Kapitola 6

Závěr

V diplomové práci byla představena moderní zobrazovací metoda Point Based Rendering. Byly popsány a vysvětleny základní algoritmy, které se staly stavebním kamenem dalšího vývoje této metody. Práce obsahuje porovnání nejznámějších metod a diskutuje jejich přednosti a slabosti. Na konci první části práce byly nastíněny další směry vývoje v této oblasti.

Point Based Rendering je oblast velmi mladá a prochází bouřlivým vývojem. Stávající algoritmy jsou zlepšovány, nové algoritmy vznikají a celá oblast je značně nepřehledná. Cílem této práce bylo danou problematiku zmapovat a zpřehlednit, což bylo snad naplněno v první části dokumentu.

Součástí diplomové práce je program, který implementuje jednu zvolenou PBR techniku. Tento demonstrační program dokáže zobrazovat rozsáhlé trojúhelníkové sítě rychleji než klasické zobrazovací metody při zachování srovnatelné kvality. Program je dostatečně rychlý, ale pouze nastiňuje možnosti Point Based Renderingu. Pro širší použití by bylo potřeba program rozšířit o použití textur, možnost uložení předzpracovaného LOD stromu a celkově zlepšit jeho paměťovou náročnost.

6.1 Osobní zhodnocení

Point Based Rendering je téma zajímavé zejména díky své aktuálnosti. Zpracování přehledu existujících metod bylo pro mě velkým přínosem. Velkým lákadlem byly také možnosti PBR metod — vykreslování velmi rozsáhlých scén při zachování interaktivnosti.

Vytváření programu bylo zejména pracné, ale místy i zábavné a poučné. Hlavním těžištěm při psaní programu byly netriviální algoritmy (rekurzivní vytváření a vykreslování LOD stromu) a řešení matematických problémů (výpočet obálkových koulí, rychlý výpočet velikosti průmětu a test viditelnosti). Program neobsahuje žádné sofistikované uživatelské rozhraní, ale myslím, že poskytuje všechny potřebné funkce.

Zpočátku mi kladlo značné potíže velké množství článků zabývající se problematikou Point Based Rendering. PBR je v současnosti velmi moderní a jednotlivé metody se bouřlivě vyvíjejí a celá situace je nepřehledná. Navíc zatím neexistuje žádný ucelený souhrn zabývající se metodami PBR.

Literatura

[1] Alexa. M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C. T.: Point Set Surfaces. In Proceedings of IEEE Visualization, 2001.

[2] Adamson, A., Alexa, M.: Ray Tracing Point Set Surfaces. In Proceedings of Shape Modeling International 2003, 2003.

[3] Carpenter, L.: The A-buffer: An Antialiased Hidden Surface Method. In Siggraph '84 Proceedings, 1984.

[4] Chen, B., Minh Xuan Nguyen: POP: A Hybrid Point and Polygon Rendering System for Large Data. In Proceedings of IEEE Visualization, 2001.

[5] Chun-Fa Chang, Bishop, G., Lastra, A.: LDI Tree: A Hierarchical Representation for Image-based Rendering. The University of North Carolina at Chapel Hill, Department of Computer Science, 1999.

[6] Coconu, L., Hege, H.: Hardware-Oriented Point-Based Rendering of Complex Scenes. In Proceedings of the 13th Eurographics Workshop on Rendering, 2002.

[7] Fleishman, S., Cohen-Or, D., Alexa. M., Silva, C. T.: Progressive Point Set Surfaces. ACM Transaction on Graphics, 2002.

[8] Grossman, J. P.: Point Sample Rendering. Department of Electrical Engineering and Computer Science, MIT, 1998.

[9] Levoy, M., Whitted, T.: The Use of Points as Display Primitives. The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[10] Liu Ren, Pfister, H., Zwicker, M.: Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. In Eurographics 2002 Proceedings, 2002.

[11] Pfister, H., Zwicker, M., Baar, J., Gross, M.: Surfels: Surface Elements as Rendering Primitives. In Siggraph 2000 Proceedings, 2000.

[12] Räsänen, J.: Surface Splatting: Theory, Extensions and Implementation. Helsiki University of Technology, Department of Computer Science, 2002.

[13] Rusinkiewicz, S., Levoy, M.: QSplat: A Mutliresolution Point Rendering System for Large Meshes. In Siggraph 2000 Proceedings, 2000.

[14] Schaufler, G., Jensen, H. W.: Ray Tracing Point Sampled Geometry. In Proceedings of the 11th Eurographics Workshop on Rendering, 2000.

[15] Shade, J., Grotler, S., Li-Wei He, Szeliski, R.: Layered Depth Images. In Siggraph '98 Proceedings, 1998.

[16] Stamminger, M., Drerrakis, G.: Interactive Sampling and Rendering for Complex and Procedural Geometry. In Proceedings of the 12th Eurographics Workshop on Rendering, 2001.

[17] Wand, M., Fischer, M., Peter, I., Friedhelm Meyer auf der Heide, Straßer, W.: The Randomized Z-buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In Siggraph '01 Proceedings, 2001.

[18] Wand, M., Straßer, W.: Multi-Resolution Rendering of Complex Animated Scenes. In Eurographics 2002 Proceedings, 2002.

[19] Zwicker, M., Pfister, H., Baar, J., Gross, M.: Surface Splatting. In Siggraph '01 Proceedings, 2001.

[20] Žára, J., Beneš, B., Felkel, P.: Moderní počítačová grafika. Computer Press, Praha, 1998.

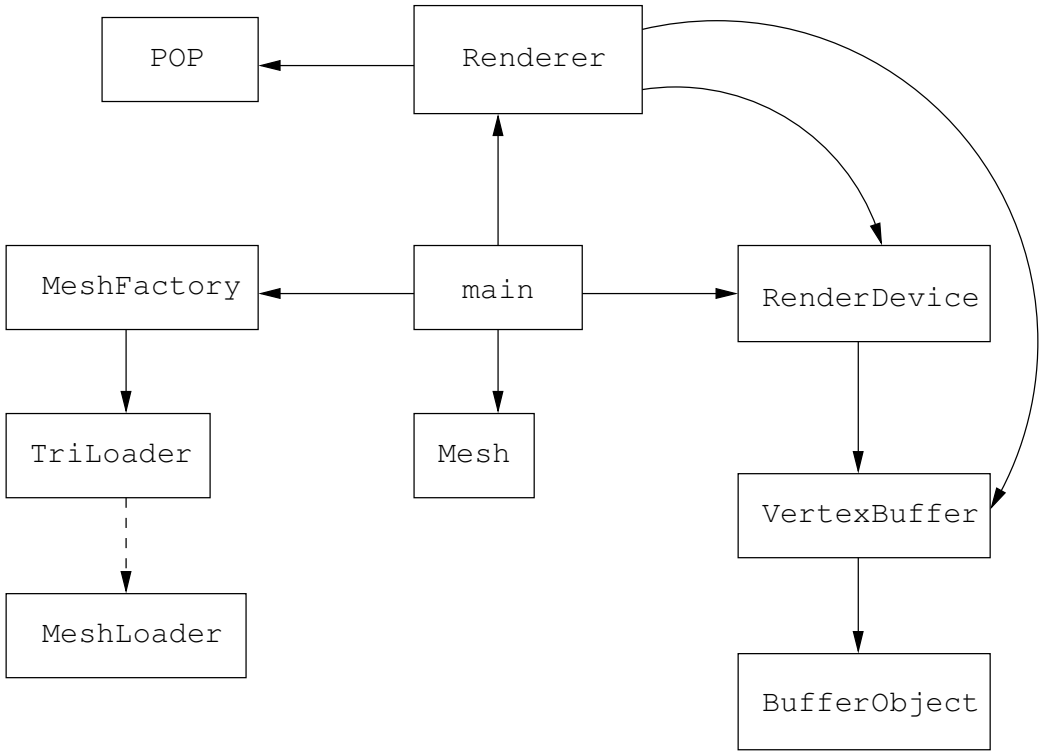
Dodatek A

Programová dokumentace

V tomto dodatku bude popsána struktura aplikace z hlediska programátora. Nejprve budou popsány jednotlivé moduly a pak bude graficky znázorněno provázání mezi nimi.

- `BufferObject` — Třída pro alokaci paměti s možností alokace přímo z paměti grafické karty.
- `Mesh` — Třída reprezentující trojúhelníkovou síť.
- `MeshFactory` — Třída pro načítání objektů `Mesh` z disku. Umožňuje zaregistrovat instance třídy `MeshLoader` pro různé přípony souborů.
- `MeshLoader` — Abstraktní rozhraní pro třídy načítající `Mesh` ze souboru.
- `POP` — Tento modul obsahuje definici uzlu POP stromu a metody pro vytváření těchto uzlů jak z trojúhelníků, tak z množiny jiných uzlů.
- `RenderDevice` — Třída sloužící k vykreslování na obrazovku. Obsahuje metody usnadňující práci s OpenGL API.
- `Renderer` — Hlavní výkonná třída obsahující kód pro vykreslování modelu a pro předzpracování.
- `TriLoader` — Implementace abstraktního rozhraní `MeshLoader` načítající soubory TRI.
- `VertexBuffer` — Třída reprezentující pole souřadnic vrcholů a jejich normál. Předává se třídě `RenderDevice` při zobrazování.
- `main` — Hlavní modul, obsahuje funkci `main` a stará se o inicializaci aplikace a uživatelské rozhraní.
- `vecmath` — Struktury, funkce a operátory pro vektorovou matematiku ve 3D. Obsahuje zejména definici tříd `vec3f` a `mat4f`.

Na obrázku (A.1) je znázorněno provázání mezi jednotlivými třídami a moduly aplikace.



Obrázek A.1: Struktura programu.

Dodatek B

Uživatelská dokumentace

B.1 Instalace a spuštění programu

Aplikace nevyžaduje žádný speciální instalační proces. Výkonná část programu je tvořena pouze binarním souborem `pbr.exe`, který je možné spustit z libovolného místa. Program ke své funkci potřebuje knihovnu *Glut*, jejíž nejnovější verzi lze získat na adrese `http://www.xmission.com/~nate/glut.html`. Na CD je u spustitelného souboru přiložena její aktuální verze, se kterou byl program testován.

Program se spouští z příkazové řádky s jedním povinným parametrem — jménem souboru, který obsahuje vstupní trojúhelníkovou síť. Jediným podporovaným formátem je TRI a jméno souboru musí končit příponou `.tri`, jinak je soubor aplikací odmítnut.

B.2 Uživatelské rozhraní

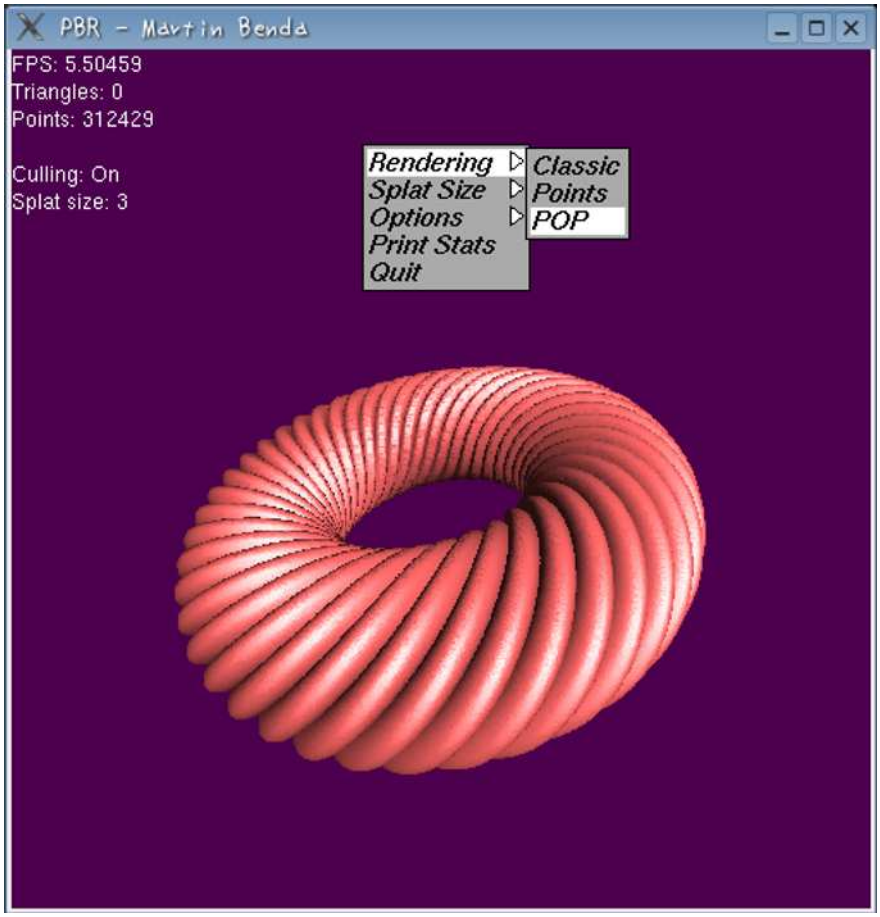
Uživatelské rozhraní aplikace je tvořené pouze jedním oknem, ve kterém je zobrazen načtený model. Aplikace se ovládá zejména myší:

- Tažením levého tlačítka myši se provádí intuitivní rotace modelu kolem jeho středu.
- Tažením pravého tlačítka myši je model posouván podél osy z (přibližování a oddalování).
- Tažením při stisku levého i pravého tlačítka se model posouvá rovnoběžně s rovinou xy.
- Stiskem prostředního tlačítka myši se vyvolá menu aplikace.

Na obrázku (B.1) je zobrazena aplikace s vyvolaným menu.

V levém horním rohu okna se vypisují aktualní statistiky:

- **FPS** — Aktuální počet snímků za sekundu.



Obrázek B.1: Uživatelské rozhraní aplikace.

- **Triangles** — Počet aktuálně zobrazených trojúhelníků.
- **Points** — Počet aktuálně zobrazených bodů.
- **Culling** — Ukazatel, zda je aktivní algoritmus pro odstraňování těch částí scény, které jsou odvrácené od uživatele.
- **Splat size** — Velikost prahu pro vykreslování splatů (uplatní se při vykreslování bodů a při vykreslování metodou POP).

Menu obsahuje tři podmenu `Render`, `Splat Size` a `Options` a volby `Print Stats` a `Quit`. Volba `Print Stats` vypíše statistiky na konzoli a volba `Quit` slouží k ukončení programu.

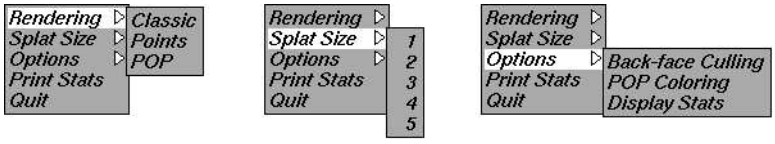
V menu `Render` se nastavuje metoda, kterou bude model vykreslen:

1. **Classic** — Klasické vykreslení celého modelu jako trojúhelníkové sítě.
2. **Points** — Vykreslení vrcholů trojúhelníkové sítě jako bodů.
3. **POP** — Vykreslení metodou POP.

V menu `Splat Size` se nastavuje velikost prahu pro vykreslení bodů v rozmezí jedna až pět.

Menu `Options` obsahuje položky sloužící k zapínaní a vypínání různých voleb aplikace:

1. **Back-face Culling** — Použití algoritmu pro eliminaci odvrácených částí scény.
2. **POP Coloring** — Zobrazení modelu metodou POP tak, že trojúhelníky jsou zobrazeny modře a body červeně.
3. **Display Stats** — Zobrazení statistik v levém horním rohu okna.



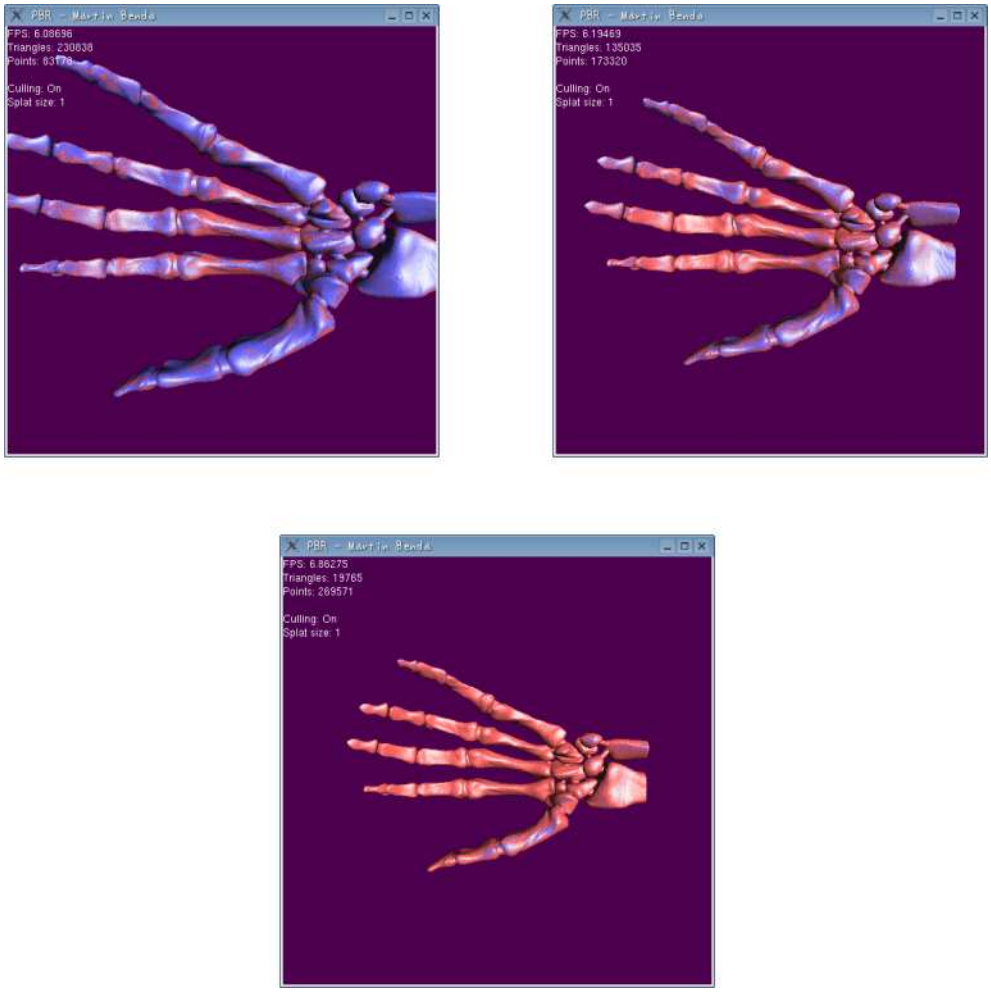
Obrázek B.2: Menu aplikace.

Veškeré položky menu jsou přístupné přímo z klávesnice — to umožňuje rychlejší a pohodlnější ovládání aplikace.

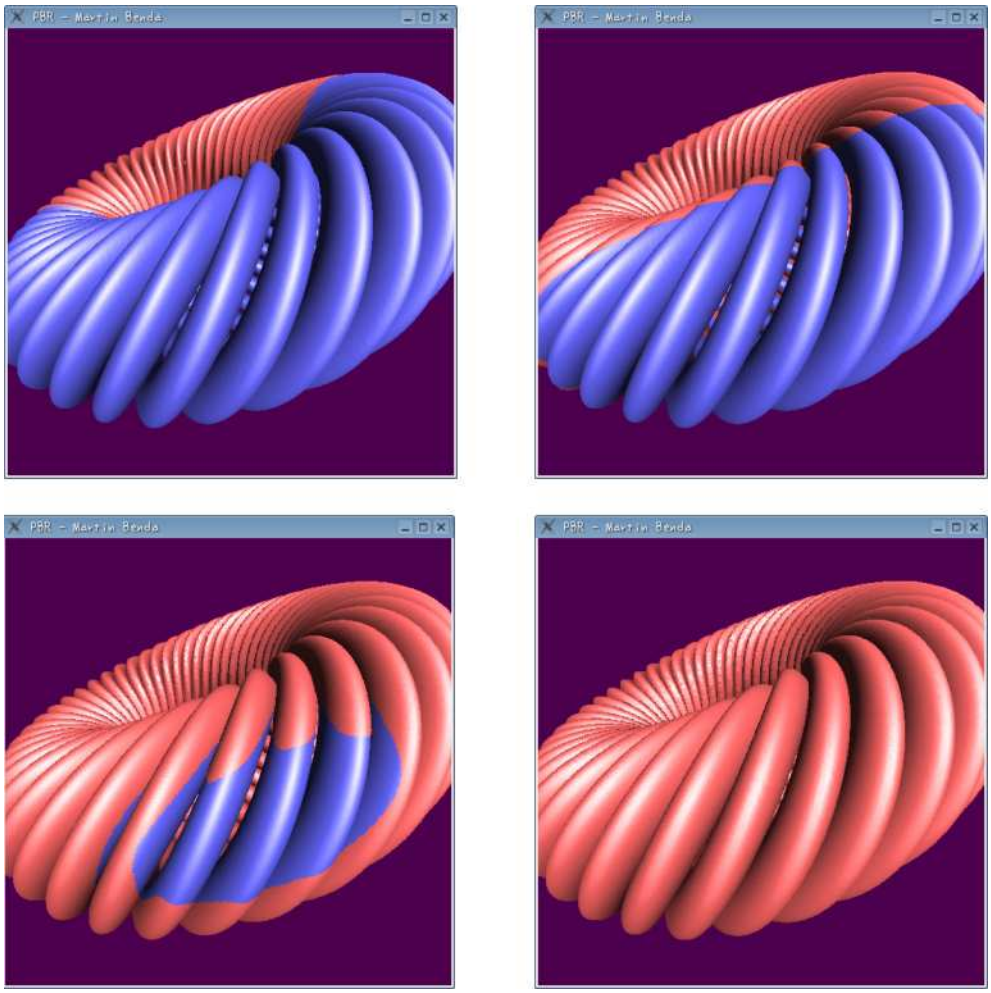
Klávesa	Odpovídající položka v menu
Esc, Q	`Quit`
Enter	`Print Stats`
A	`Render -> Classic`
S	`Render -> Points`
D	`Render -> POP`
1 – 5	`Splat Size -> 1 - 5`
B	`Options -> Back-face Culling`
C	`Options -> POP Coloring`
Space	`Options -> Display Stats`

Dodatek C

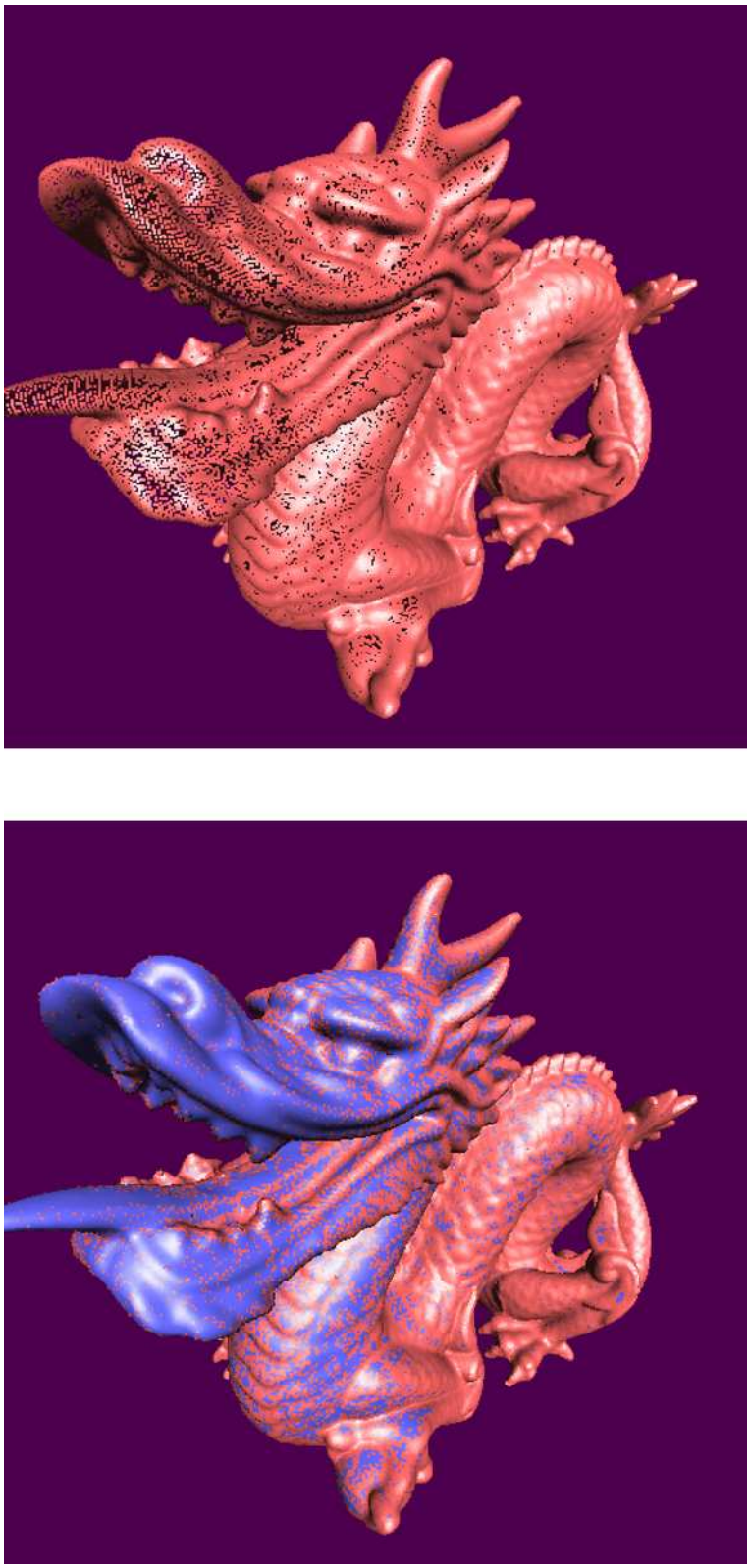
Ukázky výstupů



Obrázek C.1: Algoritmus používá dle potřeby buď trojúhelníky (modrá), nebo body (červená).



Obrázek C.2: Čím větší je práh pro vykreslení splatů, tím méně trojúhelníků se použije.



Obrázek C.3: Tam, kde by mezi body vznikaly díry, se použijí trojúhelníky.