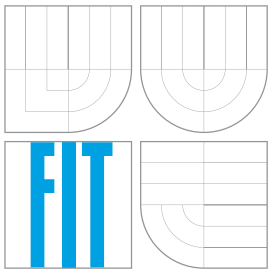
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

OPTIMALIZOVANÉ SLEDOVÁNÍ PAPRSKU

OPTIMIZED RAY TRACING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK BRICH

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Dr. Ing. PAVEL ZEMČÍK

BRNO 2008

Abstrakt

Cílem této práce je vytvořit optimalizovaný program pro zobrazování 3D scény metodou sledování paprsku. Nejprve je stručně vysvětlena teorie a jednotlivé techniky. Další část práce se věnuje možnostem urychlení algoritmu. Jsou to zejména techniky dělení prostoru, algoritmus pro rychlé nalezení průsečíku paprsku s trojúhelníkem a různé možnosti paralelizace celého zobrazovacího algoritmu. Samostatná kapitola je věnována návrhu a realizaci programu.

Klíčová slova

Sledování paprsku, 3D scéna, optimalizace, paralelizace, SIMD, SSE, svazky paprsků, dělení prostoru, oktalový strom, kd-strom, barycentrické souřadnice.

Abstract

Goal of this work is to write an optimized program for visualization of 3D scenes using ray tracing method. First, the theory of ray tracing together with particular techniques are presented. Next part focuses on different approaches to accelerate the algorithm. These are space partitioning structures, fast ray-triangle intersection technique and possibilities to parallelize the whole ray tracing method. A standalone chapter addresses the design and implementation of the ray tracing program.

Keywords

Ray tracing, 3D scene, optimization, parallelization, SIMD, SSE, ray bundles, ray packets, space partitioning, octree, kd-tree, barycentric coordinates.

Citace

Radek Brich: Optimalizované sledování paprsku, diplomová práce, Brno, FIT VUT v Brně, 2008

Optimalizované sledování paprsku

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Pavla Zemčíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Radek Brich
19. května 2008

Obsah

Kapitola 1

Úvod

Počítačová grafika je obor informatiky zabývající se syntetickým vytvářením obrazu a zpracováním rastrových, vektorových a prostorových informací. Oblast zobrazování trojrozměrných scén (rendering) má široké uplatnění ve vědě i zábavním průmyslu. Vstupem rendereru je popis scény skládající se z objektů rozmístěných v prostoru (většinou polygony v kartézském souřadném systému), světelných zdrojů a dalších podpůrných informací. Cílem je scénu efektivně zobrazit.

Používané metody lze rozdělit do dvou obecných skupin – rasterizační metody a sledování paprsku (ray tracing). Rasterizér zpracovává postupně jednotlivé trojrozměrné objekty a převádí je do dvojrozměrné reprezentace v rovině obrazovky. K řešení problému viditelnosti objektů se většinou používá tzv. Z-buffer, do kterého se ukládá vzdálenost objektu od obrazovky.

Ray tracing zobrazuje scénu po jednotlivých bodech, z nichž vysílá paprsky a hledá průsečíky s objekty ve scéně. Tato metoda narozdíl od rasterizace umožňuje interakci mezi objekty a tedy zobrazovat reálné stíny či odrazy. Obě metody lze různě rozširovat a dosahovat tak pokročilých vizuálních efektů. Metoda sledování paprsku však umožňuje daleko vyšší míru fotorealismu právě díky možnosti interakcí mezi objekty.

Technika rasterizace našla své uplatnění především v oblasti interaktivní grafiky, kde je potřeba rychlé zobrazování posloupnosti snímků. Rasterizační pipeline je dobře realizovatelná v hardware. Naopak ray tracing se kvůli své výpočetní náročnosti využívá spíše pro neinteraktivní tvorbu obrázků a animací (offline rendering) a díky rekurzivní podstatě algoritmu je jeho hardwarová implementace daleko obtížnější.

Požadavky na kvalitu zobrazování však stále rostou a tak lze v budoucnu očekávat uplatňování ray tracingu v interaktivní grafice. Napomůže tomu i logaritmická složitost algoritmu, díky které je ray tracing od určité komplexity zobrazované scény rychlejší než rasterizační technika, u které výpočetní náročnost roste lineárně.

Mezi přednosti metody sledování paprsku patří také snadné využití více výpočetních jednotek, tedy možnost paralelizace. Zobrazování jednotlivých bodů obrazovky je vzájemně nezávislé a tak jej lze snadno rozdělit mezi více procesorů či počítačů v síti. Jedinou podmínkou je sdílení či zkopírování zobrazovaných dat. Algoritmus sledování paprsku také dokáže dobře využít schopností dnešních procesorů, především pak jejich SIMD vlastností.

Na základně metody sledování paprsku vznikla řada dalších odvozených technik. Distribuovaný ray tracing nabízí měkké stíny a odrazy, hloubku ostrosti a další neostré efekty. Ještě pokročilejší (a výpočetně náročnější) je photon tracing, umožňující globální interakci objektů s využitím metody sledování paprsků.

Tato práce pojednává o principech algoritmu sledování paprsku, se zaměřením na mož-

nosti jeho urychlení. Dále se zabývá návrhem a implementací takto optimalizovaného ray traceru.

Následující druhá kapitola popisuje teorii sledování paprsku obecně, třetí kapitola se zaměřuje na možnosti urychlení této metody, čtvrtá kapitola pak podrobně rozebírá jeden z důležitých způsobů urychlení ray tracingu a to paralelizaci tohoto algoritmu. V páté kapitole se podíváme na mojí implementaci optimalizovaného ray traceru. V závěrečné kapitole shrnu výsledky práce, stav implementace a další možnosti postupu práce na programu.

Tato diplomová práce úzce navazuje na semestrální projekt, v jehož rámci jsem nastudoval teorii metody sledování paprsku a napsal převážnou část druhé a třetí kapitoly. V implementaci ray traceru jsem v této diplomové práci dále značně pokročil, program je nyní ucelenější, lépe dokumentovaný a podporuje další techniky a algoritmy. Hlavním přínosem po semestrálním projektu je zvláště kapitola o paralelizaci sledování paprsku a využití SIMD instrukcí. Tuto technologii jsem rovněž využil v implementaci.

Kapitola 2

Zobrazování scény metodou sledování paprsku

Sledování paprsku je matematická metoda vykreslování trojrozměrné grafiky. Je inspirovaná fyzikálními principy geometrické optiky. Předpokládá se přímé šíření světla ve formě paprsků, které se při interakci s povrchem objektů mohou odrážet a lámat. Paprsek je sledován od oka do scény skrze stínítko představující matici bodů obrazovky. Objekty ve scéně i simulované paprsky jsou určeny parametrickými rovnicemi. Hledání průsečíku paprsku s jednotlivými objekty spočívá v řešení soustavy rovnic objektu a paprsku.

V přírodě se světlo šíří od zdroje, například slunce, ozařuje pozorovanou scénu a některé odražené paprsky jsou pak zachyceny okem či fotoaparátem, čímž vznikne pozorovaný obrázek. Není v silách dnešních počítačů simulovat chování světla dokonale, proto musíme přistoupit k různým zjednodušením. Paprsků, které se dostanou až k pozorovateli je jen velmi malý zlomek, proto je vhodné paprsky vysílat od pozorovatele do scény, zde je nechat odrážet a lámat a po cestě sbírat světelné příspěvky. Takto funguje zobrazovací metoda sledování paprsku.

Výstupem algoritmu bude rastrový obrázek o předem známém počtu bodů. Bude nám tedy stačit vyslat jeden paprsek do scény skrze každý bod (ponechme zatím stranou možnost nadvzorkování). Tyto paprsky nazýváme primární, paprsky vznikající při odrazu a lomu ve scéně nazýváme sekundární. Speciálním případem jsou stínové paprsky.

Narozdíl od oka či fotoaparátu, kde je sítnice či film umístěna za optickým středem, zde postavíme do optického středu přímo pozorovatele a stínítko umístíme mezi něj a scénu. Každý primární paprsek sestrojíme tak, aby měl počátek v oku pozorovatele a procházel jedním bodem stínítka. Změnou polohy pozorovatele a stínítka pak můžeme libovolně měnit pohled na scénu, aniž bychom museli paprsek pokaždé transformovat. Vzdálenost a velikost stínítka určuje šíři záběru.

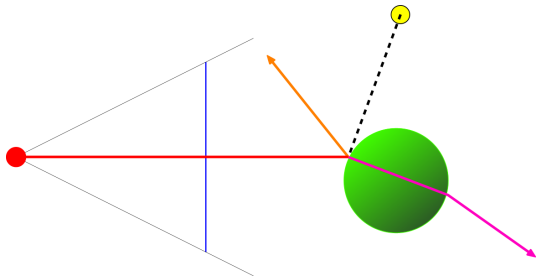
Paprsky necháme procházet scénou a hledáme průsečíky s přítomnými objekty. Většinou nás zajímá pouze nejbližší z nalezených průsečíků. V něm bychom mohli jednoduchou stínovací funkcí přímo spočítat barvu bodu a pixel zobrazit – takové technice se říká ray casting. K dosažení vyšší kvality obrazu ale budeme potřebovat další paprsky.

Stínový paprsek slouží k zjištění, zda je bod vzhledem k danému světelnému zdroji ve stínu. Paprsek nasměrujeme z bodu přímo do světelného zdroje a hledáme průsečík s jiným objektem – zde nás v případě neprůhledných předmětů zajímá, zda existuje nějaký průsečík nebo je zdroj světla přímo viditelný. Není nutné hledat průsečík nejbližší.

Odražené a lomené paprsky počítáme rekurzivně stejným způsobem jako paprsky pri-

mární. Všechny získané hodnosty nakonec předáme stínovací funkci, která z nich vypočítá barvu v daném bodě. Na odraz a lom se podíváme podrobněji dále.

K sestrojení odražených a lomených paprsků budeme rovněž potřebovat normálu v každém průsečíku s objektem. Pro trojúhelníkové modely se často používají falešné normály, které vyhladí spoje jednotlivých trojúhelníků a vytváří tak dojem detailnějšího modelu.



Obrázek 2.1: Znázornění interakce paprsku s objekty ve scéně

Na obrázku 2.1 jsou naznačeny interakce paprsku ve scéně – odražený, lomený a stínový paprsek.

Základní algoritmus sledování paprsku je poměrně standardní (výpis 2.1).

Výpis kódu 2.1: Pseudokód základního algoritmu sledování paprsku

```
Pro každý pixel v obrázku
{
  ray = paprsek od pozorovatele skrze tento pixel do scény
  raytrace(ray, 0);
}

function raytrace(ray, depth)
{
  object = NULL
  distance = nekonečno

  // najít nejbližší objekt, který paprsek protíná
  pro každý objekt ve scéně
  {
     zkontrolovat, jestli paprsek protíná tento objekt
     {
        t = vzdálenost průsečíku paprsku s objektem
        pokud je t < distance
        {
           distance = t
           object = tento objekt
        }
     }
  }

  pokud object == NULL
  {
     nastavit pixel na barvu pozadí
  }
  jinak
  {
     // vypočítat barvu pixelu pomocí rekurze
     vyslat stínové paprsky ke všem zdrojům světla
     pokud je povrch reflektivní
     {
       vytvořit odražený paprsek reflection_ray
       raytrace(reflection_ray, depth+1);
```

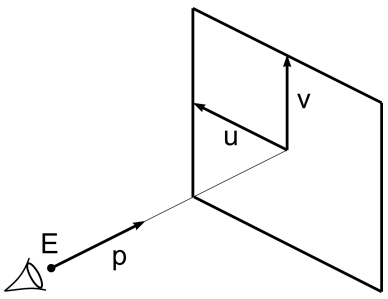
```
      }
      pokud je povrch transparentní
      {
        vytvořit lomený paprsek refraction_ray
        raytrace(refraction_ray, depth+1);
      }
      získané hodnoty předat stínovací funkci
      nastavit barvu pixelu na výsledek stínovací funkce
  }
}
```

2.1 Kamera a transformace

Sledování paprsku nevyžaduje žádnou projekční a perspektivní transformaci – správná projekce je zajištěna již způsobem, jakým jsou paprsky vysílány. Paprsky jsou vysílany z *kamery*, jejíž poloha a natočení určuje pohled do scény.

Podívejme se, jak lze vytvořit tuto kameru pro účely sledování paprsku. Kamera bude určovat polohu pozorovatele, směr pohledu a šíři záběru. Polohou pozorovatele uvažujeme bod, který je společným počátkem paprsků vysílaných do scény. Neuvažujeme žádnou optickou soustavu, přestože i tu lze simulovat. Kamera je v podstatě obdobou štěrbinového fotoaparátu, pouze *film* se zde nenachazí za *štěrbinou*, ale před ní (a výsledný obraz tudíž není převrácený).

Následující vztahy vycházejí z [4]. Vektor určující polohu pozorovatele, též nazývanou oko, budeme označovat $\mathbf{E}$. Normalizovaný vektor směru pohledu (neboli optické osy) označíme $\mathbf{p}$. Ještě je nutný jeden vektor, který určí natočení obrazu kolem směru pohledu. Tento vektor většinou určuje směr „nahoru“, tedy ukazuje ve směru od středu obrazu k jeho horní hraně (vertikálně). Pro sledování paprsku je však výhodnější zavést i vektor ve směru horizontální osy obrazu. Horizontální a vertikální směrové vektory nazveme $\mathbf{u}$ a $\mathbf{v}$. Vektory $\mathbf{p}$, $\mathbf{u}$, $\mathbf{v}$ by měly být vzájemně kolmé, jinak bude obraz deformovaný. Vzájemný vztah vektorů $\mathbf{p}$, $\mathbf{u}$, $\mathbf{v}$ je znázorněn na obrázku 2.2.



Obrázek 2.2: Určení směru pohledu kamery pomocí vektorů $\mathbf{p}, \mathbf{u}, \mathbf{v}$

Známe-li $\mathbf{E}$, $\mathbf{p}$, $\mathbf{u}$ a $\mathbf{v}$, můžeme již snadno konstruovat primární paprsky. Počátkem paprsku je vždy oko $\mathbf{E}$ a bod na stínítku, skrze který paprsek prochází, označíme $\mathbf{S}$. Z těchto vektorů získáme parametrickou rovnici paprsku 2.1.

$$\mathbf{r} = \mathbf{E} + t \cdot (\mathbf{S} - \mathbf{E}) \tag{2.1}$$

Bod $\mathbf{S}$ můžeme snadno určit jako $\mathbf{S} = u\mathbf{u} + v\mathbf{v} + f\mathbf{p}$. Skalární parametry u, v určují

bod na stínítku vzhledem k jeho středu a f souvisí se šíří záběru. S tímto již lze jednoduše naprogramovat vysílání paprsků dle parametricky určené kamery.

Nyní nastává otázka, jak vlastně vektory $\mathbf{p}$, $\mathbf{u}$ a $\mathbf{v}$ určit pro požadované nastavení kamery. Tři vzájemně kolmé vektory libovolně umístěné a natočené v prostoru nelze přímo zadat číselně, ani není jednoduché již známou soustavu vektorů natáčet. V případě otáčení takové soustavy pomůžou buď klasické transformační matice nebo kvaterniony.

Look-at kamera

Více intuitivní způsob zadávání parametrů kamery je pomocí oka $\mathbf{E}$ a cíle $\mathbf{T}$. $\mathbf{T}$ je bod, na který kamera kouká a který se má nacházet uprostřed obrazu. Opět je nutné ještě určit směr „nahoru“, tentokrát však volněji, bude stačit například jedna z hlavních os. Tento třetí parametr označíme jako vektor $\mathbf{up}$. Kamera zadaná těmito parametry se nazývá *look-at* kamera.

Přepočet těchto parametrů $\mathbf{E}$, $\mathbf{T}$, $\mathbf{up}$ na pro ray tracing vhodné parametry $\mathbf{p}$, $\mathbf{u}$ a $\mathbf{v}$ ukazují vztahy 2.2.

$$\begin{aligned} \mathbf{p} &= \mathbf{T} - \mathbf{E} \\ \mathbf{u} &= \mathbf{up} \times \mathbf{p} \\ \mathbf{v} &= \mathbf{p} \times \mathbf{u} \end{aligned} \tag{2.2}$$

Transformace

Bylo by zbytečné zde rozepisovat jednotlivé formy matic pro afinní transformace a způsob jejich kombinace, tyto základy jsou dobře popsané v dostupné literatuře (např. [1]). Podívejme se pouze, k čemu jsou nám matice dobré v metodě zobrazování sledováním paprsku.

V některých případech budeme potřebovat souřadnice v kártézském osovém systému nějak transformovat. Například libovolnou kameru lze určit pevnými parametry $\mathbf{p}$, $\mathbf{u}$ a $\mathbf{v}$ (viz výše) a transformační maticí, která tuto soustavu natočí a posune do požadovaného bodu.

Transformace se mohou hodit i pro manipulaci s objekty, které mají například jinou orientaci v prostoru než požadujeme. To může být případ jak trojúhelníkových modelů importovaných ze souboru tak i parametricky definovaných objektů.

Manipulace s objekty může být provedena dvěma způsopby – buď transformujeme přímo souřadnice objektů a nebo transformujeme paprsek do souřadného prostoru objektu, spočítáme průsečík a transformujeme zpět. První způsob může být výhodný v případě trojúhelníkových modelů – transformovat stačí jednou a při vlastním sledování paprsku již transformace nezpomalují. Druhý způsob je nutností pro některé objekty, které lze těžko orientovat libovolným způsobem (například parametrický válec).

Transformační matice jsou nejpoužívanějším způsobem polohování jak kamery, tak i objektů ve scéně. Jejich výhodou je univerzálnost – jednou maticí lze změnit polohu, velikost i natočení objektu. Vzhledem k velkému rozšíření transformací pomocí matic (obzvláště v grafických akcelerátorech) je vhodné je i v ray traceru implementovat. Díky možnosti skládání transformačních matic jsou výhodné pro použití k transformacím v grafu scény.

Rotace pomocí matic však trpí problémem známým jako *gimbal lock*. Pokud rotujeme objekt postupně podle jednotlivých os X, Y, Z, dochází k narušení ortonormality osového systému a druhá rotace může způsobit splynutí dvou os, přičemž poslední rotace pak nemá předpokládaný účinek. Tomuto problému lze předejít nahrazením tří rotačních matic pro

jednotlivé základní osy jedinou maticí rotující kolem libovolné obecné osy. Taková matice je obdobou kvaternionu, na který se podíváme v následující části.

Kvaterniony

Kvaterniony jsou rozšíření komplexních čísel do čtyřrozměrného prostoru a tvoří nekomutativní kruh s dělením. Kvaternion lze zapsat $q = a + bi + cj + dk$, přičemž i, j, k jsou jednotky a platí pro ně $i^2 = j^2 = k^2 = ijk = -1$.

Kvaterniony jsou vhodné k definování rotací v trojrozměrném prostoru. Umožňují snadno vyjádřit rotaci kolem libovolné obecné osy a tedy i jakoukoliv rotaci v prostoru. Narozdíl od transformačních matic kavaterniony nemohou měnit polohu objektu. Kvaternionová rotace netrpí problémem *gimbal locku* a jsou numericky stabilnější než matice. Rotace kolem osy $\mathbf{A}$ o úhel ρ je reprezentována kvaternionem $q = cos(\rho/2) + sin(\rho/2)(A_x i + A_y j + A_z k)$.

Vektor $\mathbf{v}'$, který vznikne rotací vektoru $\mathbf{v}$ kvaternionem q spočítáme $\mathbf{v}' = q\mathbf{v}q^{-1}$. Vektor $\mathbf{v}$ musí být v kvaternionové formě (s reálnou složkou rovnou nule). Kvaternion $q^{-1} = a_q - b_q i - b_q j - d_q k$ je inverzní prvek násobení. Způsob násobení kvaternionů ukazuje rovnice 2.3.

$$\begin{aligned} q_1 q_2 =& a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2 \\ &+ (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2) i \\ &+ (a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2) j \\ &+ (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2) k \end{aligned} \tag{2.3}$$

Podobně jako u matic lze i kvaternionové rotace kombinovat, celkovou rotaci získáme násobením dvou kvaternionů s původními rotacemi.

Rotace vyjádřené maticemi a kvaterniony lze vzájemně převádět.

2.2 Průsečík paprsku se základními tvary

Průsečíky s jednotlivými objekty počítáme řešením soustavy rovnic paprsku a objektu. V nalezeném průsečíku nás většinou zajímá i směr normály povrchu v daném bodě. Pro implicitní matematické tvary je nalezení takové normály snadné. Trojúhelník, stejně jako plocha, není objemový tvar a má tedy v každém bodě dvě možné normály. Normálu tedy bude třeba nějak zadat.

Paprsek lze parametricky zapsat rovnicí 2.4, kde $\mathbf{A}$ je výchozí bod paprsku a vektor $\mathbf{d}$ udává směr. S konvexními objekty může mít paprsek maximálně dva průsečíky t_1 a t_2. Složitějšími objekty se zde zabývat nebudeme.

$$\mathbf{X}(t) = \mathbf{A} + t\mathbf{d},\ t \geq 0 \tag{2.4}$$

Rovina

Rovinu určuje bod $\mathbf{P}$ a normála $\mathbf{n}$. Z těchto parametrů lze utvořit rovnici roviny 2.5.

$$\mathbf{n} \cdot (\mathbf{X} - \mathbf{P}) = 0 \tag{2.5}$$

Dosazením rovnice paprsku 2.4 do rovnice 2.5 získáme vztahy 2.6 a 2.7.

$$\mathbf{n} \cdot (\mathbf{A} + t\mathbf{d} - \mathbf{P}) = 0 \tag{2.6}$$

$$t = \frac{\mathbf{n} \cdot (\mathbf{P} - \mathbf{A})}{\mathbf{n} \cdot \mathbf{d}} \tag{2.7}$$

Pokud je $t \geq 0$, pak paprsek protíná rovinu a hledaný průsečík je $\mathbf{A} + t\mathbf{d}$.

Alternativně můžeme rovinu určit normálou $\mathbf{n}$ a skalárním parametrem d (rovnice 2.8, 2.9).

$$d = -\mathbf{n} \cdot \mathbf{P} \tag{2.8}$$

$$t = -\frac{\mathbf{n} \cdot \mathbf{A} + d}{\mathbf{n} \cdot \mathbf{d}} \tag{2.9}$$

Parametr d zde určuje posunutí roviny od počátku ve směru normály.

Koule

Koule má rovnici 2.10, kde $\mathbf{C}$ je střed koule, $\mathbf{X}$ bod na jejím povrchu a r poloměr.

$$(\mathbf{X} - \mathbf{C})^2 = r^2, \tag{2.10}$$

Dosazením rovnice paprsku 2.4 do rovnice 2.10 získáme vztah 2.11.

$$(\mathbf{A} + t\mathbf{d} - \mathbf{C})^2 - r^2 = 0, \tag{2.11}$$

Pro zjednodušení si určíme substituci $\mathbf{V} = \mathbf{A} - \mathbf{C}$ a hledáme vzdálenost průsečíku t (vztahy 2.12, 2.13).

$$\mathbf{d}^2 t^2 + 2\mathbf{V} \cdot \mathbf{d}t + \mathbf{V}^2 - r^2 = 0, \tag{2.12}$$

$$t = \frac{-2\mathbf{V} \cdot \mathbf{d} \pm \sqrt{(2\mathbf{V} \cdot \mathbf{d})^2 - 4\mathbf{d}^2(\mathbf{V}^2 - r^2)}}{2\mathbf{d}^2}, \tag{2.13}$$

Průsečíky $\mathbf{X}$ získáme dosazením t zpět do rovnice paprsku.

Normálu k povrchu v bodě $\mathbf{X}$ vypočítáme jako $\mathbf{N} = (\mathbf{X} - \mathbf{C})/r$.

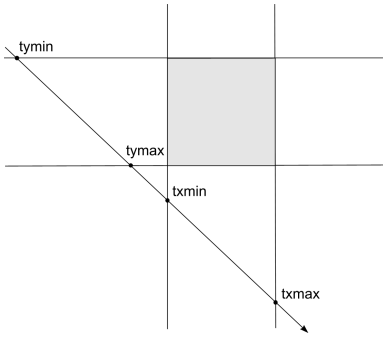
Kvádr

Kvádr se skládá ze šesti polygonů. Nejjednodušší metoda zjištění průsečíku spočívá v hledání průsečíků s rovinami jednotlivých polygonů a testování, zda je průsečík uvnitř daného obdélníku.

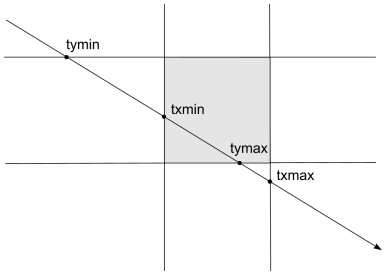
Jsou-li však strany souběžné s osami souřadného systému, lze průsečík hledat efektivněji. Spočítáme průsečíky paprsku s plochami všech šesti stran kvádru. Průsečíky vždy označíme jako bližší a vzdálenější a následně je porovnáváme. Pokud největší z bližších průsečíků je větší než nejmenší ze vzdálených průsečíků, paprsek minul kvádr. Není-li tato podmínka splněna, paprsek kvádrem prochází a jako nejbližší průsečík s kvádrem můžeme použít největší z blízkých průsečíků s jeho plochami. Obrázky 2.3 a 2.4 zobrazují průsečíky s plochami kvádru pro dva z možných případů.

Průsečík s plochou kolmou k jedné z os lze vypočítat s využitím pouze jediné souřadnice (rovnice 2.14).

$$t = (P_x - O_x)/D_x \tag{2.14}$$



Obrázek 2.3: Průsečíky paprsku s rovinami stran pokud paprsek minul kvádr



Obrázek 2.4: Průsečíky paprsku s rovinami stran pokud paprsek zasáhl kvádr

P_x je poloha plochy na dané ose, O_x a D_x jsou souřadnice počátku a směru paprsku na stejné ose.

Efektivnější metoda hledání průsečíku s osově zarovnaným kvádrem byla popsána v [10]. Tato metoda využívá Plückerovy souřadnice.

Trojúhelník

Trojúhelník je určen třemi body v prostoru. Průsečík hledáme nejprve s plochou určenou těmito body. Pokud takový průsečík existuje, otestujeme zda se nachází uvnitř trojúhelníku nebo vně. Pro zjednodušení a urychlení výpočtu je vhodné převést trojúhelník i bod do vhodnějšího souřadného systému a počítat v něm. Rychlou intersekcí paprsku s trojúhelníkem se budeme dále zabývat ve třetí kapitole.

Normálu lze vypočítat vektorovým součinem vektorů dvou hran trojúhelníku (rovnice 2.15) a následnou normalizací.

$$\mathbf{N} = (\mathbf{B} - \mathbf{A}) \times (\mathbf{C} - \mathbf{A}) \tag{2.15}$$

Směrem normály je určena viditelná strana trojúhelníku, nevyužijeme-li speciální techniky k oboustrannému zobrazování. Je zřejmé, že takto vznikají dvě možnosti zadávání trojúhelníku, lišící se pořadím bodů a zvolenými hranami pro výpočet normály.

Interpolace normál trojúhelníku

Součástí Phongova návrhu osvětlovacího modelu (tomu se budeme věnovat dále) byla i metoda interpolace normál, pomocí které lze vizuálně zvýšit jemnost trojúhelníkového modelu.

Tato metoda vyžaduje mít definovány normály ve všech vrcholech trojúhelníku. V bodech uvnitř trojúhelníku se potom vypočítá normála lineární interpolací tří normál vrcholů.

Normály ve vrcholech lze počítat automaticky, ale je to nákladná operace, neboť vyžaduje znalost všech trojúhelníků sousedících s daným bodem. Proto se tyto normály obvykle počítají při načítání modelu nebo ještě dříve při jeho tvorbě, pak jsou uloženy v datovém souboru spolu s modelem.

K interpolaci lze s výhodou využít barycentrycké souřadnice (viz 3.1).

2.3 Stínovací funkce, odraz a lom

V této části se budeme zabývat výpočtem barvy objektu v bodě, kde jej protnul paprsek. Pokud bychom se nezabývali odrazy a lomy, stačil by nám Phongův (či jiný) osvětlovací model, který dle normály a světelného vektoru počítá barvu v daném bodě včetně odrazu světelného zdroje. Přidáme-li skutečné odrazy a lomy, dostaneme tři různé barvy, které bude nutné nějakým způsobem sloučit.

Funkci počítající barvu osvětleného bodu budeme nazývat stínovací funkce. V jednodušších ray tracerech bývá tato funkce pevně naprogramována a uživatel může osvětlování měnit pouze několika parametry (Phongovy parametry, odrazivost, průhlednost, index lomu, apod.). Komplexnější zobrazovací systémy stínovací funkci více generalizují. Programovatelná může být jen osvětlovací funkce, nebo i rekurzivní vysílání sekundárních a stínových paprsků.

Nejznámějším systémem s propracovanými stínovacími funkcemi (nazývanými shadery) je RenderMan od společnosti Pixar [2]. V takovém systému uživatel může upravovat celý proces stínování, přičemž má k dispozici běžně používané funkce pro počítání s vektory, vysílání sekundárních paprsků, zjišťování zastínění bodu (occlussion) a další. Takový systém je velmi silný, umožňuje vytváření spousty různých efektů, simulaci materiálů i chování světla na povrchu objektů.

Nyní se podívejme na základní prostředky sledování paprsku, lokální osvětlovací model – věnovat se budu pouze známému Phongovu modelu – a na způsob generování odražených a lomených paprsků.

Phongův osvětlovací model

Phongův model určuje množství světla v daném bodě podle směru dopadajícího paprsku, povrchové normály v daném bodě a směru ke zdroji světla. Model popisuje tři složky – okolní světlo (konstantní), rozptýlené světlo (dle normály povrchu) a odražené světlo (odraz světelného zdroje). Funkce je vypočítána pro všechny světelné zdroje a jednotlivé příspěvky jsou posléze sečteny.

Phongův osvětlovací model pro jeden světelný zdroj vyjadřuje rovnice 2.16. Konstanty $\kappa_a, \kappa_d, \kappa_s$ se vztahují k materiálu objektu a konstanty i_a, i_d, i_s k světelnému zdroji. Konstanta α vyjadřuje lesklost materiálu.

$$I_p = \kappa_a i_a + \kappa_d i_d (\mathbf{L} \cdot \mathbf{N}) + \kappa_s i_s (\mathbf{R} \cdot \mathbf{V})^\alpha \tag{2.16}$$

Vektor $\mathbf{L}$ udává směr ke zdroji světla, $\mathbf{N}$ je normálový vektor v daném bodě povrchu, $\mathbf{V}$ je vektor směru k pozorovateli (obrácený směr příchozího paprsku) a $\mathbf{R}$ je směr dokonale odraženého paprsku ze světelného zdroje.

Phongův osvětlovací model lze dobře kombinovat s dalšími technikami.

Odraz paprsku

Geometrická fyzika o odraženém paprsku říká, že se nachází vždy ve stejné rovině s paprskem dopadajícím a s normálou povrchu svírá úhel o stejné velikosti, ale opačném znaménku. Ve vektorovém počtu lze odražený paprsek vyjádřit rovnicí 2.17:

$$\mathbf{R} = \mathbf{I} - 2\mathbf{N}(\mathbf{I} \cdot \mathbf{N}) \tag{2.17}$$

Zde je $\mathbf{I}$ příchozí paprsek a $\mathbf{N}$ normála povrchu.

Takto spočítáné odrazy jsou vždy dokonalé. Ve skutečném světě však nebývají povrchy dokonale hladké, ale vyskytují se na nich drobné nerovnosti. Odraz je pak více rozptýlený. Podobného efektu lze dosáhnout využitím distribuovaného sledování paprsku. Jednodušší možností je odražený paprsek pouze lehce odvrátit z dokonalé dráhy s použitím určité míry náhody.

Lom paprsku

Lom světla je další z efektů snadno dosažitelných sledováním paprsku. Paprsek se při přechodu mezi dvěma prostředími láme úměrně k poměru indexů lomu těchto prostředí. Vztah vyjadřuje Snellův zákon 2.18.

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \tag{2.18}$$

Pro potřeby sledování paprsku je tuto rovnici třeba přepsat do vektorové formy 2.19.

$$\mathbf{T} = \frac{n_1}{n_2} \cdot \mathbf{I} + \left(\frac{n_1}{n_2} \cos\theta_1 - \cos\theta_2 \right) \cdot \mathbf{N} \tag{2.19}$$

Vektor $\mathbf{I}$ je opět příchozí paprsek a $\mathbf{N}$ normála povrchu. Odvození tohoto vztahu je rozepsáno v [18].

Nutné je rozeznat přechod dovnitř objektu a ven z objektu, kvůli správným indexům lomu. U trojúhelníkových modelů lze využít směr povrchové normály – ukazuje-li normála ve směru přicházejícího paprsku (skalární součin je kladný), vstupuje paprsek do objektu, jinak vychází ven z objektu. Index lomu okolí lze předpokládat roven jedné.

Světlo procházející průhledným objektem obyčejně ztrácí svou intenzitu. Toto chování popisuje Lambertův-Beerův zákon [17]. K výpočtu tohoto efektu je navíc třeba znát vzdálenost, jakou urazil paprsek vnitřkem objektu, což není v případě sledování paprsku nijak složité.

Takto získaný lom světla odpovídá chování skla a podobných materiálů. Složitějších efektů může být dosaženo metodami distribuovaného ray tracingu.

Distribuovaný ray tracing

Konvenční metoda sledování paprsku vytváří ostré stíny, odrazy a lomy. Reálné scény tak ale nevypadají, zdroje světla nebývají bodové a povrchy absolutně hladké. Abychom dosáhli měkkých stínů, můžeme tvořit plošné zdroje světla a k nim vysílat více stínových paprsků. Podobně můžeme vyslat více odražených paprsků rozptýlených v malém prostorovém úhlu a získáme měkčí odraz. Paprsky je však třeba distribuovat náhodně, ne v pravidelné mřížce – došlo by k aliasingu, například stíny by se skládály ze zřetelných pruhů. Náhodnou distribucí paprsků vzniká naopak šum, zrnité stíny – to je pro lidské oko přijatelnější. Metoda s náhodnou distribucí paprsků se nazývá také stochastický ray tracing.

2.4 Textury

Textury popisují vzhled materiálu, ze kterého je daný objekt vyroben, či kresbu nebo tapetu na povrchu objektu. Také je možné je využít jako náhradu příliš složité geometrie objektu.

Běžně se používají trojrozměrné a dvourozměrné textury. Trojrozměrná textura je definována v každém bodu prostoru (nebo alespoň jednotkové krychle) a je vhodná k vyjádření materiálu, z kterého je objekt například vyřezán. Dvourozměrné textury slouží k zobrazení obrázku na povrchu tělesa. Obrázek je třeba na povrch tělesa nějakým způsobem namapovat, tj. přiřadit každému bodu na povrchu tělesa bod v rovině obrázku. Toto přiřazení se nazývá mapovací funkce.

Textura může být definována buď explicitně, diskrétními daty (obrázkem), nebo implicitně nějakou funkcí. Textury popsané funkcí nazýváme procedurální. Barva procedurální textury v každém bodě se v ray traceru obvykle počítá až ve chvíli, kdy je potřeba. To znamená, že takové textury vyžadují výpočetní čas, oplátkou za značnou úsporu paměti a za neomezeně detailní rysy.

U všech druhů textur může docházet k *aliasingu*, čili optickým vadám způsobeným vzorkování vysokých frekvencí na frekvencích nižších. V ray traceru lze tento problém řešit tak, že je sledována vzdálenost sousedních paprsků a podle ní odhadnuta frekvence vzorkování a přizpůsobena textura tak, aby obsahovala jen frekvence nižší. Tato technika *diferenciálů paprsků* je popsána v [11].

Mapování 2D textur

Souřadnice určující bod textury označíme u, v. Mapovací funkce je taková funkce $f(x, y, z)$, která každému bodu na povrchu tělesa přiřadí určitou kombinaci u, v.

Nejjednodušší mapování je planární. Dvě ze souřadnic x, y, z jsou namapovány na u, v, třetí není brána v ohled (textura bude na této ose konstantní).

Kubické mapování rozšiřuje planární mapování do všech tří rozměrů. Textura je uložena v jedné až šesti pixmapách, mapovací souřadnice a správná pixmapa jsou zvoleny podle největší složky polohového vektoru daného bodu (nebo normály).

Sférické mapování využívá převodu souřadnic do sférického souřadného systému. Sférické souřadnice Φ, ρ jsou mapovány na u, v souřadnice textury. Převod z původního souřadného systému na texturovací souřadnice je vyjádřen rovnicemi 2.20. Rozsah texturovacích souřadnic je samozřejmě vhodné upravit na $\langle 0, 1)$.

$$u_{sphere} = \arctan\left(\frac{y}{x}\right) \qquad -\pi \leq u_{sphere} \leq \pi \tag{2.20}$$

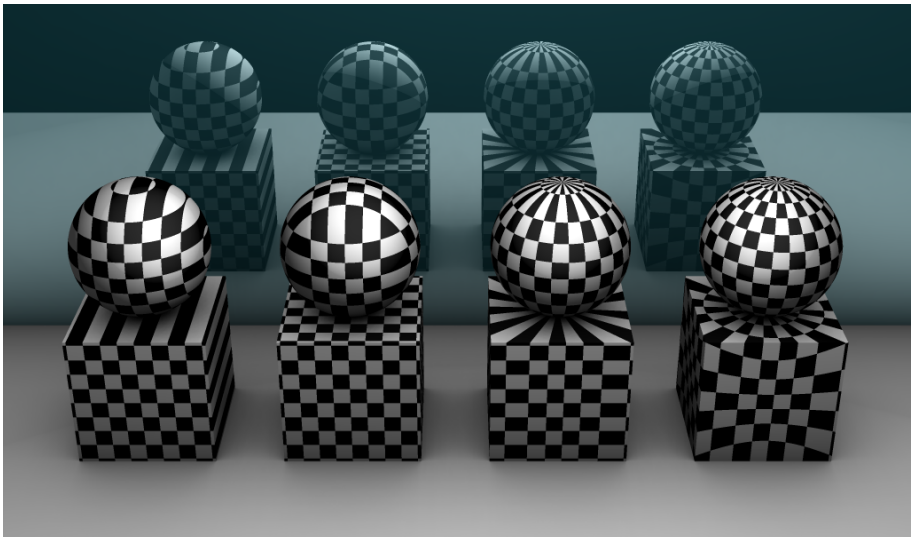
$$v_{sphere} = \arccos\left(\frac{z}{\sqrt{x^2 + y^2 + z^2}}\right) \qquad 0 \leq v_{sphere} \leq \pi \tag{2.21}$$

Cylindrické mapování je podobné sférickému, pouze jedna ze souřadnic je mapována přímo.

$$u_{cyl} = \arctan\left(\frac{y}{x}\right) \qquad -\pi \leq u_{cyl} \leq \pi \tag{2.22}$$

$$v_{cyl} = z \qquad v_{cyl} \in \mathbb{R} \tag{2.23}$$

Textura na trojúhelníkový model může být mapována jedním z jmenovaných způsobů nebo i libovolně jinak. Texturovací souřadnice se obvykle přímo specifikují pro každý vrchol trojúhelníku a interpolují pro body uvnitř trojúhelníku.



Obrázek 2.5: Různé způsoby mapování 2D textur. Zleva planární, kubické, cylindrické a sférické, zobrazeno na krychli a kouli.

Procedurální textury

Klasické textury definují barvu pro určitou množinu bodů a barva v ostatních bodech je dopočítávána interpolací. Oproti tomu procedurální textura je funkce, která pro dané souřadnice bodu přímo vrátí barvu. Je definována plynule ve všech bodech a není nutná interpolace. Procedurální textury mohou být trojrozměrné i dvourozměrné (s využitím vhodného mapování).

Nevýhodou procedurálních textur je poněkud obtížnější hledání funkce pro danou předlohu. Dobře lze popsat přírodní materiály jako dřevo či mramor, různé vzorky a mozaiky, nebo například mraky na obloze. Základem bývá generátor šumu, jehož výstup je dále upravován různými matematickými výrazy a obarven vhodnou paletou.

Texturovací funkce může kromě souřadnic přijímat také další parametry, které mohou ovlivňovat různé rysy výsledné textury.

Procedurální texturování lze velmi dobře kombinovat se zobrazováním metodou sledování paprsku. Jednoduše v každém průsečíku spočítáme barvu pomocí texturovací funkce. Obvykle nejsou potřeba žádné speciální texturovací souřadnice.

Kapitola 3

Možnosti urychlení výpočtu

Základní algoritmus sledování paprsku je výpočetně velmi náročný. Každý vyslaný paprsek je testován se všemi objekty ve scéně, navíc při odrazu a lomu vznikají další paprsky. Bez různých urychlovacích technik by tak byl ray tracing pro svou výpočetní náročnost téměř nepoužitelný.

V průběhu let bylo vymyšleno mnoho metod jak zobrazování sledováním paprsku urychlit. Důkladně byly zkoumány možnosti rychlého výpočtu průsečíku paprsku s trojúhelníkem. Rovněž byla vyzkoušena řada algoritmů dělících prostor na vhodně uspořádané oblasti, jejichž cílem je efektivní nalezení objektů protínajících dráhu paprsku. Z takových prostorových struktur se nejvíce prosazují kd-stromy, které jsou dostatečně jednoduché a přesto dosahují velmi dobrých výsledků na obecných scénách. Algoritmy dělení prostoru také zajišťují, že neviditelné trojúhelníky nepřekáží a nejsou zbytečně zpracovávány, tudíž obyčejně není třeba žádné ořezávání scény ani LOD (level of detail).

Pojďme se podívat, jakými metodami lze dobu výpočtu snížit. Nabízejí se tři možnosti: vysílat méně paprsků, počítat méně průsečíků a počítat průsečíky rychleji.

Největší vliv na rychlost výpočtu má množství počítaných průsečíků. Objekty můžeme v prostoru uspořádat tak, abychom nemuseli každý paprsek testovat proti všem. Z těchto technik dělení prostoru se nyní využívají především oktalové stromy a kd-stromy, které dokáží scénu jemně rozdělit na různě velké podprostory. K oběma známe efektivní algoritmy průchodu daným stromem. Dělení prostoru probereme v druhé části této kapitoly.

Další možností je vysílat méně primárních paprsků. Tato technika se nazývá podvzorkování a podíváme se na ní ve třetí části kapitoly. Podvzorkování má však negativní dopad na kvalitu obrazu a tudíž se tomu většinou snažíme vyhnout.

V první části se budeme věnovat efektivním algoritmům pro hledání průsečíku paprsku s trojúhelníkem.

3.1 Efektivní výpočet průsečíku paprsku s trojúhelníkem

Ray tracer tráví velkou část svého času počítáním průsečíků paprsku a objektů ve scéně, proto je velmi důležité tyto výpočty maximálně optimalizovat. V případě trojúhelníků je základním problémem určit, zda průsečík paprsku s plochou trojúhelníku leží mezi jeho vrcholy nebo vně. Jednou z nejpoužívanějších technik je výpočet barycentrických souřadnic průsečíku.

Barycentrické souřadnice

Mějme trojúhelník definovaný třemi vrcholy $\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}$ a bod $\mathbf{r}$ uvnitř tohoto trojúhelníku. Polohu bodu lze popsat třemi souřadnicemi odvozenými od plochy mezi stranami trojúhelníku a bodem $\mathbf{r}$. Výpočet těchto souřadnic v trojrozměrném prostoru je popsán v článku [16]. Efektivnější je výpočet ve dvou rozměrech, minoritní rozměr trojúhelníku můžeme potlačit. Tento minoritní rozměr je určen podle největší složky normály trojúhelníku. Zbylé dva rozměry označíme u a v.

Definujme vrcholy trojúhelníku $\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}$ a bod $\mathbf{r}$ ve dvou rozměrech dle rovnic 3.1 až 3.4.

$$\mathbf{v_1} = (u_1, v_1) \tag{3.1}$$
$$\mathbf{v_2} = (u_2, v_2) \tag{3.2}$$
$$\mathbf{v_3} = (u_3, v_3) \tag{3.3}$$
$$\mathbf{r} = (u, v) \tag{3.4}$$

Bod $\mathbf{r}$ lze zapsat váženou sumou těchto tří vrcholů dle vztahu 3.5.

$$\mathbf{r} = \lambda_1 \mathbf{v_1} + \lambda_2 \mathbf{v_2} + \lambda_3 \mathbf{v_3} \tag{3.5}$$

Konstanty $\lambda_1, \lambda_2, \lambda 3$ jsou barycentryckými souřadnicemi bodu $\mathbf{r}$. Pomocí těchto souřadnic snadno určíme polohu bodu $\mathbf{r}$ vzhledem k ploše trojúhelníku: Je-li každá ze souřadnic větší nebo rovna nule, bod leží uvnitř trojúhelníku. Naopak, je-li alespoň jedna ze souřadnic záporná, bod leží mimo. Dále platí rovnice 3.6, takže nám stačí dvě ze souřadnic, třetí je určena jako zbytek do jedné.

$$\lambda_1 + \lambda_2 + \lambda_3 = 1 \tag{3.6}$$

Dosazením rovnice 3.6 do 3.5 a rozepsáním vektorů získáme soustavu rovnic 3.7, 3.8.

$$u = \lambda_1 u_1 + \lambda_2 u_2 + (1 - \lambda_1 - \lambda_2) u_3 \tag{3.7}$$
$$v = \lambda_1 v_1 + \lambda_2 v_2 + (1 - \lambda_1 - \lambda_2) v_3 \tag{3.8}$$

Řešením této soustavy získáme rovnice 3.9, 3.10 pro souřadnice λ_1 a λ_2.

$$\lambda_1 = \frac{(v_3 - v)(u_2 - u_3) - (u3 - u)(v2 - v3)}{(u1 - u3)(v2 - v3) - (u2 - u3)(v1 - v3)} \tag{3.9}$$
$$\lambda_2 = \frac{(u_3 - u)(v_1 - v_3) - (v3 - v)(u1 - u3)}{(u1 - u3)(v2 - v3) - (u2 - u3)(v1 - v3)} \tag{3.10}$$

Je výhodné si výrazy nezávislé na u a v dopředu předpočítat a uložit je do datové struktury trojúhelníku. Zvláště se nabízí čitatel zlomků v rovnicích 3.9, 3.10. Podrobně takový algoritmus s předpočítáváním devíti reálných konstant a jedné celočíselné (minoritní osy) popsal Ingo Wald ve své disertační práci [14].

3.2 Dělení prostoru

Než si popíšeme konkrétní algoritmy, podívejme se jaké jsou možnosti dělení prostoru využitelné pro rychlé nalezení objektů ve dráze paprsku.

Nejjednodušší možností je rozdělit prostor rovnoměrně na menší oblasti využitím **uniformní mřížky**. To je pro obecné scény nevýhodné, je potřeba zbytečně mnoho paměti. Navíc objekty ve scéně nebývají rozloženy rovnoměrně.

Techniku lze rozšířit na hierarchii mřížek, která se ale špatně prochází. Speciálním případem takové hierarchie je **oktalový strom** (octree), který vždy dělí prostor na osm stejně velkých oblastí, tedy v podstatě mřížkou 2×2×2. Jsou známy různé algoritmy pro efektivní práci s oktalovým stromem.

Prostor můžeme také dělit obecnými plochami vždy na dva podprostory. Tato technika se nazývá **BSP** (Binary Space Partitioning) a její fundamentální strukturou je binární strom. Pro akceleraci sledování paprsku ale není technika BSP příliš vhodná. Procházení takového stromu je relativně náročné a tudíž nemůže být příliš hluboký. Velmi časově náročná je i stavba BSP stromu.

Mnohem více se používá speciální varianta BSP stromu, nazývaná **kd-tree** (k-dimenzionální strom). Tento strom využívá k dělení prostoru jen plochy kolmé k základním osám souřadného systému. Stavbu i procházení kd-tree lze oproti BSP výrazně urychlit a je to v současnosti jedna z nejoblíbenějších technik pro obecné scény.

Modelovací programy zpravidla vytváří tzv. graf scény, z kterého můžeme odvodit hierarchii obalových těles (Bounding Volume Hierarchy, BVH), též využitelnou jako strukturu dělící prostoru. Výhodou této techniky je snadná úprava stromu u dynamických scén, nevýhodou nutnost explicitního dělení scény – bez grafu scény tuto strukturu nepostavíme a pro obecný oblak trojúhelníků je tedy nepoužitelná. Také je vhodné BVH zkombinovat s nějakým dalším dělícím algoritmem, protože množství trojúhelníků v obalových tělesech nejnižší úrovně může být stále příliš velké.

Jedním z novějších algoritmů je hierarchie hraničních intervalů (Bounding Interval Hierarchy, BIH [13]), technika kombinující obalová tělesa a kd-strom. Oproti kd-stromu zde každý uzel obsahuje dvě dělící plochy, umisťované na hranice objektů. BIH je výborné pro interaktivní raytracing dynamických scén, protože jeho stavba je velmi rychlá a urychlení výpočtu scény se blíží kd-stromu. Má také malé nároky na paměť. Avšak i stavbu kd-stromu lze na úkor kvality výsledného stromu urychlit a dosáhnout podobných výsledků jako má BIH ([7]).

V následující části se podíváme na dva z jmenovaných algoritmů, oktalový strom a kd-strom.

3.2.1 Oktalový strom

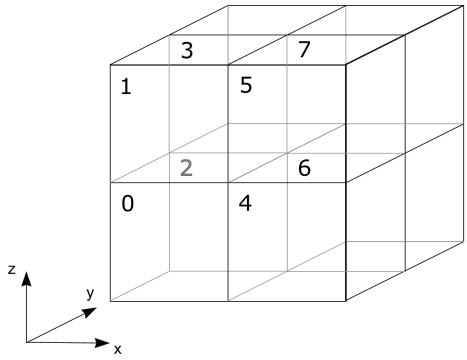
Oktalový strom je stromová struktura, ve které každý uzel obsahuje osm poduzlů, není-li terminální. Uzel představuje krychlovou oblast prostoru, která je rozdělena na osm stejně velkých podkrychlí třemi rovinami kolmými na základní osy.

Hlavní předností této techniky je jednoduchý a rychlý algoritmus stavby stromu. Průchod stromu lze optimalizovat a dosáhnout rychlého nalezení nejbližšího průsečíku. Tento algoritmus je využíván v mnoha modelovacích programech k vykreslování obrázku sledováním paprsku, protože narozdíl od kd-stromu je velmi snadné a rychlé postavit oktalový strom na požádání před vykreslením snímku.

Za povšimnutí může rovněž stát možnost rozšíření algoritmu na libovolné dělící roviny a

požití nějaké heuristiky, například pro kd-stromy oblíbené SAH (viz dále). Tímto by velmi klesla rychlost budování stromu, ale strom by mohl lépe odpovídat stavbě scény. Algoritmus průchodu lze snadno upravit náhradou konstant dělících region v polovině za proměnné uložené ve struktuře stromu. Vraťme se ale nyní k obyčejnému oktalovému stromu.

Uložení uzlů v paměti pro použití v algoritmech bude spočívat na nějakém způsobu uložení odkazů na poduzly a informaci o tom, zda je uzel terminální. Poduzly lze efektivně seřadit do pole a odkazovat se na ně jediným ukazatelem. Jejich počet je vždy osm a vytváříme je všechny v jeden okamžik – udržování osmi ukazatelů by bylo zbytečné. Jediný ukazatel tedy bude odkazovat na první poduzel a k ostatním se dostaneme inkrementováním tohoto ukazatele.



Obrázek 3.1: Rozmístění sektorů oktalového stromu podle indexu

Poduzly je vhodné v poli nějak rozumně seřadit. Intuitivní variantou je indexování podle základních os. Poloha sektoru na každé ze tří os je určena jedním bitem, všechny kombinace tří bitů nám pak dají požadovaných osm indexů (tabulka 3.2). Výsledné uspořádání indexů v třírozměrném prostoru je znázorněno na obrázku 3.1.

Obrázek 3.2: Způsob tvorby indexu uzlu oktalového stromu

bit	2	1	0
osa	X	Y	Z

Stavba stromu

Algoritmus stavby oktalového stromu je implementačně nenáročný. Region dělíme vždy rekurzivně na osm stejných částí a do nich rozdělíme obsažené objekty. Algoritmus 3.1 jsem navrhl tak, aby obsahoval minimum větvění.

Výpis kódu 3.1: Pseudokód algoritmu dělení uzlu oktalového stromu

```
OctreeNode::divide(AABB, depth)
{
    if (depth >= maxdepth || shapes < minshapes)
        return;

    new child[8];
    new childAABB[8];
```

```
    // připravíme všech osm AABB
    childAABB[all] = AABB;
    split{x,y,z} = middle on each axis of AABB;
    childAABB[0,1,2,3].max.x = splitx;
    childAABB[4,5,6,7].min.x = splitx;
    childAABB[0,1,4,5].max.y = splity;
    childAABB[2,3,6,7].min.y = splity;
    childAABB[0,2,4,6].max.z = splitz;
    childAABB[1,3,5,7].min.z = splitz;

    // rozdělíme objekty
    for (each shape)
        for (each child)
            if (shape intersects childAABB)
                child.add(shape);

    // pokračujeme rekurzivně
    for (each child)
        child.divide(childAABB, depth+1);
}
```

Je vhodné navíc určit další podmínku ukončení rekurze, protože při dělení do podregionů se může jeden objekt namnožit až na osm kopií. Nevhodné dělení samozřejmě zpomalí následné průchody stromem. Toto musíme ošetřit až po rozdělení objektů mezi poduzly a tudíž zajistit, aby objekty zůstaly i v mateřském uzlu, vytvořené struktury poduzlů opět uvolnit a daný uzel označit jako list.

Přídavnou ukončovací podmínku jsem nastavil dle rovnice 3.11. Konstanty jsou určeny experimentálně.

$$(N_{node} \leq 8 \wedge N_{sub} > 2 \cdot N_{node}) \vee N_{sub} \geq 6 \cdot N_{node} \tag{3.11}$$

kde N_{node} je počet objektů v uzlu a N_{sub} je celkový počet objektů ve všech poduzlech. Tato podmínka osvědčila, pozitivní vliv na výkon je znatelný.

Algoritmus průchodu stromem

Efektivní algoritmus průchodu oktalovým stromem byl popsán v [12]. Zde si vytvoříme zásobníkovou variantu tohoto algoritmu.

Pro uložení stavu průchodu stromem na zásobník si připravíme strukturu 3.2.

Výpis kódu 3.2: Struktura stavu průchodu stromem

```
struct traversal_state
{
        Float tx0, ty0, tz0, tx1, ty1, tz1, txm, tym, tzm;
        OctreeNode *node;
        int next;
}
```

Velikost zásobníku můžeme pevně nastavit na hloubku stromu plus jednu položku navíc pro uložení aktuálního stavu. Aktuální stav bude vždy na vrchu zásobníku a připravíme si pro něj také přímé pojmenování položek struktury (zde jako alias, v C lze použít #define), aby byl zápis přehlednější. Do položky next je ukládán další poduzel k analýze, případně speciální hodnota UP, pokud paprsek opouští buňku a DOWN pokud je třeba prozkoumat poduzel.

Výpis kódu 3.3: Algoritmus průchodu oktalovým stromem

```
octree_traverse(Ray r)
{
    struct traversal_state st[max_depth+1];
    struct traversal_state *st_cur = st;
    alias node st_cur->node;
    alias tx0 st_cur->tx0;   alias ty0 st_cur->ty0;   alias tz0 st_cur->tz0;
    alias tx1 st_cur->tx1;   alias ty1 st_cur->ty1;   alias tz1 st_cur->tz1;
    alias txm st_cur->txm;   alias tym st_cur->tym;   alias tzm st_cur->tzm;

    a = 0;
    ro = r.origin;
    rdir = Vector(1.0/ray.dir.x, 1.0/ray.dir.y, 1.0/ray.dir.z);
    aabb = (root node AABB)

    // otočit nevhodně směrované paprsky
    if (rdir.x < 0.0)
    {
        ro.x = aabb.min.x + aabb.max.x - ro.x;
        rdir.x = -rdir.x;
        a |= 4;
    }

    // výpočet průsečíku s blízkou a vzdálenou plochou krychle
    tx0 = (aabb.min.x - ro.x) * rdir.x;
    tx1 = (aabb.max.x - ro.x) * rdir.x;

    // vše rovněž pro Y a Z

    if (max3(tx0, ty0, tz0) > min3(tx1, ty1, tz1))
        return NULL;

    node = root;
    st_cur->next = DOWN;

    repeat
    {
        if (st_cur->next == DOWN)
        {
            if (tx1 < 0.0 || ty1 < 0.0 || tz1 < 0.0)
                // paprsek neprotíná tento uzel
                st_cur->next = UP;
            else if (node is leaf)
            {
                compute intersection with objects in this node
                if found
                    return object
                else
                    st_cur->next = UP;
            }
            else
            {
                txm = 0.5 * (tx0 + tx1);
                tym = 0.5 * (ty0 + ty1);
                tzm = 0.5 * (tz0 + tz1);

                st_cur->next = first_node(tx0, ty0, tz0, txm, tym, tzm);
            }
        }

        while (st_cur->next == UP)
        {
            // pop state from stack
            if (st_cur == st)
                return NULL; // nothing to pop, finish
            st_cur--;
        }
```

```
        // push current state
        *(st_cur+1) = *st_cur;
        st_cur++;

        switch (st_cur->next)
        {
            case 0:
                tx1 = txm;
                ty1 = tym;
                tz1 = tzm;
                node = node->getChild(a);
                (st_cur-1)->next = next_node(txm, 4, tym, 2, tzm, 1);
                break;

            // podobně pro 1 až 6

            case 7:
                tx0 = txm;
                ty0 = tym;
                tz0 = tzm;
                node = node->getChild(7^a);
                (st_cur-1)->next = UP;
                break;
        }
        st_cur->next = DOWN;
    }
}
```

3.2.2 kd-strom

Kd-strom je variantou BSP stromu s osově kolmými dělícími plochami. Strom je vybudován s uvážením rozmístění objektů v prostoru. Dělící plochy je třeba volit tak, aby cena procházení stromu byla co nejnižší. K odhadu ceny využijeme vhodnou heuristiku. Kd-strom je vhodným algoritmem pro sledování paprsku díky jednoduchosti a flexibilitě.

Stavba kd-stromu

Stavba kd-stromu spočívá v rekurzivním hledání nejlepší dělící roviny a následné distribuci objektů mezi nové uzly. Daný algoritmus v pseudokódu je vypsán v 3.4.

Výpis kódu 3.4: Algoritmus stavby kd-stromu

```
kdtree_divide(Shape[] shapes, AABB volume)
{
    new node
    if (termination condition)
        set_leaf()
        return node
    find best plane p
    (L, R) = split volume with plane p
    for (each from shapes)
        if (shape intersects L)
            add to shapesL
        if (shape intersects R)
            add to shapesR
    child[0] = kdtree_divide(shapesL, L)
    child[1] = kdtree_divide(shapesR, R)
    return node
}

kdtree_build(Shape[] shapes)
```

```
{
    volume = AABB(shapes)
    root = kdtree_divide(shapes, volume)
}
```

V algoritmu není definována ukončovací podmínka a způsob hledání vhodné dělící roviny. Tyto dvě části určují kvalitu výsledného stromu s ohledem na co nejrychlejší průchod stromu. Hledání dělící roviny je navíc časově nejnáročnější část algoritmu, proto se jím budeme zabývat nejvíce.

Dělící rovinu bychom měli hledat takovou, aby cena průchodu stromem a nalezení průsečíku pro všechny paprsky z daného pohledu byla co nejnižší. Testování všech možných stromů však není reálné, proto musíme využít nějakou heuristiku, která tuto cenu nejlépe odhadne.

Nejpoužívanější heuristikou je SAH (Surface Area Heuristic, [9]). Pro každou dělící rovinu odhadujeme zlepšení ceny hledání průsečíku. Pro odhad je brán v ohled povrch obalového tělesa daného uzlu a počet trojúhelníků v uzlu.

Vypočítáme cenu pro nerozdělený uzel (rovnice 3.12) a pro uzel rozdělený nějakou dělící rovinou (rovnice 3.13).

$$C_{unsplit} = SA(V) \cdot (C_T + N \cdot C_X) \tag{3.12}$$

$$C_{split} = SA(V) \cdot C_T + SA(V_L) \cdot (C_T + N_L \cdot C_X) + SA(V_R) \cdot (C_T + N_R \cdot C_X) \tag{3.13}$$

$SA(V)$ je obalová plocha nerozděleného uzlu, $SA(V_L)$ a $SA(V_R)$ jsou plochy levého a pravého poduzlu po rozdělení dělící rovinou. Podobně N, N_L a N_R jsou počty trojúhelníků v nerozděleném uzlu a v uzlech po rozdělení. C_T je průměrná cena průchodu uzlem a C_X je průměrná cena počítání průsečíku paprsku s trojúhelníkem.

Hledáme takovou dělící rovinu, pro kterou je C_{split} nejmenší a zároveň platí podmínka $C_{split} < C_{unsplit}$. Toto je hledaná ukončovací podmínka, navíc ji můžeme omezit například ještě limitem maximálního zanoření.

Výrazy pro cenu můžeme dále zjednodušit (3.14, 3.15) zmenšením poměrem $1/C_X$, neboť nepotřebujeme absolutní hodnotu ceny, pouze porovnáváme jejich velikosti. Stanovíme konstantu $\mathcal{K} = C_T/C_X$. Nejvhodnější hodnotu této konstanty pro danou scénu a platformu pak můžeme určit experimentálně.

$$C_{unsplit} = SA(V) \cdot (\mathcal{K} + N) \tag{3.14}$$

$$C_{split} = SA(V) \cdot \mathcal{K} + SA(V_L) \cdot (\mathcal{K} + N_L) + SA(V_R) \cdot (\mathcal{K} + N_R) \tag{3.15}$$

Stačí testovat roviny protínající krajní body těles (např. vrcholy trojúhelníků). Je dokázáno, že minimum cenové funkce se vždy nachází v jednom z těchto okrajových bodů. Stále je však množství možných dělících rovin kvadraticky závislé na počtu trojúhelníků ($O(N^2)$). Hledání rovin je pro komplexní scény velmi časově náročné.

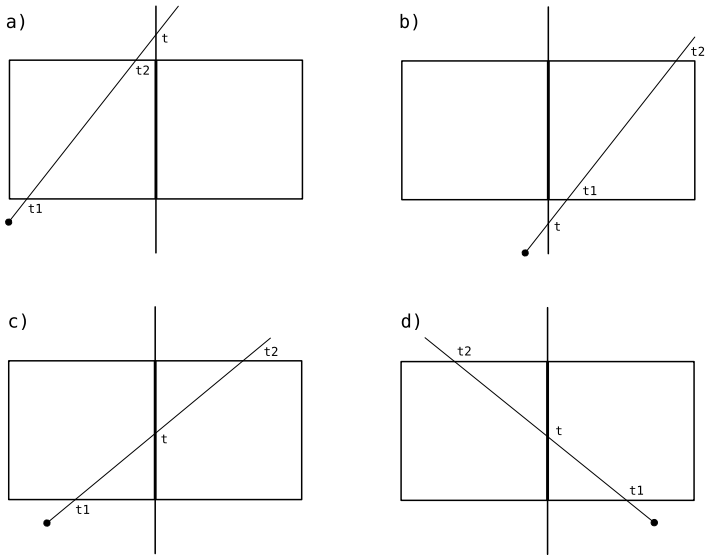
Algoritmus lze optimalizovat pro rychlejší budování s jen malým zhoršením kvality výsledného stromu. Takové efektivnější hledání dělících rovin je popsáno v [15].

Algoritmus průchodu kd-stromem

Vyhledání nejbližšího objektu s pomocí kd-stromu využívá opět rekurzívní dělení stromu od kořene. Zkoumáme, které poduzly daný paprsek protíná a tyto uzly dále procházíme.

Dojdeme-li k listovému uzlu, počítáme průsečík se všemi objekty v tomto uzlu a hledáme nejbližší. Navíc je nutné kontrolovat, jestli je nalezený průsečík skutečně uvnitř daného uzlu. Průsečíky za hranicemi uzlu musíme ignorovat, ty budou zkoumány později znovu (vznikají u objektů, které zasahují do více uzlů).

Algoritmus začíná testem, jestli paprsek protíná obalový kvádr celého stromu. Pokud ano, vypočítáme průsečíky s tímto kvádrem ve formě parametrů rovnice paprsku, t_1 a t_2, kde t_1 je blíže k počátku paprsku. Dále spočítáme průsečík s dělící rovinou, t. Podle vztahu t k t_1 a t_2 pak vybereme další uzly k procházení.



Obrázek 3.3: Přehled případů, které mohou nastat při hledání uzlů proťatých paprskem

$$\begin{aligned}
&\text{a) } t_1 \leq t \wedge t_2 \leq t \Rightarrow \text{pouze levý} \\
&\text{b) } t \leq t_1 \wedge t \leq t_2 \Rightarrow \text{pouze pravý} \\
&\text{c) } t_1 < t \wedge t < t_2 \Rightarrow \text{levý, potom pravý} \\
&\text{d) } t < t_1 \wedge t_2 < t \Rightarrow \text{pravý, potom levý}
\end{aligned} \tag{3.16}$$

Mohou vzniknout čtyři případy, podle kterých rozhodneme, zda budeme dále procházet jeden nebo oba poduzly a v jakém pořadí. Tyto případy jsou znázorněny v diagramu 3.3 a rovnice 3.16 ukazuje příslušné vztahy.

Tyto uzly dále rekurzívně procházíme, dokud nenarazíme na listový uzel. Z čísel t, t_1 a t_2 vždy vybereme ty, které se týkají hranic právě zpracovávané buňky a přiřadíme je do nových parametrů t_1 a t_2. V listových uzlech testujeme průsečík paprsku s objekty. Existuje-li jeden nebo více průsečíků, které jsou zároveň menší než t_2, vybereme nejbližší a algoritmus končí.

Rekurzi lze vhodným způsobem nahradit zásobníkem o velikosti rovné maximální hloubce stromu. Rovněž výpočetně náročný výpočet průsečíku paprsku s dělící rovinou lze odložit a k porovnávání použít přímo polohu dělící roviny a hodnoty krajních průsečíků z předchozích uzlů. Takový rychlý algoritmus průchodu kd-stromem se zásobníkem popsal V. Havran v [6] (včetně ukázkové implementace).

3.3 Omezení počtu vysílaných paprsků

Nyní již víme, jak efektivně vypočítat průsečík s trojúhelníkem, jak rychle najít první protnutý objekt v dráze paprsku a dostáváme se k poslední z urychlovacích technik, omezování počtu vysílaných paprsků.

Nabízejí se dvě možnosti, vysílání méně primarních paprsků, tj. vzorkování méně bodů v ploše obrazu, nebo efektivnější výpočet sekundárních paprsků. Budou nás zajímat pouze stínové paprsky, neboť ostatní sekundární paprsky – odrazové a lomené – vysíláme vždy po jednom a je tedy nutné je vždy testovat. Omezovat sekundární paprsky budeme pouze podle hloubky zanoření případně velikosti příspěvku k hodnotě daného bodu.

Adaptivní vzorkování primárních paprsků

Základní technikou zobrazování sledováním paprsku je vysílání jednoho paprsku pro každý bod obrazu, tedy jednoho vzorku na jeden pixel. V tomto případě jsou v oblastech s menší hustotou objektů některé vysílané paprsky přebytečné a můžeme je vynechat. Je-li v okolí bodu minimální rozdíl hodnoty a odstínu barev, lze předpokládat, že tento bod bude mít podobnou barvu a tedy ji dopočítat jako průměr okolních barev.

Paprsky můžeme vysílat v řídké mřížce napříč obrazem a v oblastech s velkými rozdíly barev mřížku dále dělit. Pro malé rozdíly barev a nepříliš velkou rozteč mřížky lze pak barvu jednoduše interpolovat. Podobně lze pokračovat až pod velikost pixelu a využít tuto techniku i k potlačení aliasingu.

Nevýhodou je, že malé objekty mohou mřížce uniknout. V případě pohyblivé sekvence obrazů pak dochází k střídavému mizení a objevování se těchto malých objektů, podle toho jak se střetávají s mřížkou vzorkovaných bodů. Podobně může být stále viditelný aliasing. Vzorkování pravidelnou mřížkou vytváří zuby na hranách s malými úhly. Množství vzorků k úplnému potlačení aliasingu by bylo příliš veliké.

Místo pravidelné mřížky můžeme vzorky rozmístit víceméně náhodně – obrazové artefakty jsou pak nahrazeny šumem, který je pro lidské oko méně rušivý. Tato technika se nazývá stochastický ray tracing.

Stínové paprsky

K dosažení realistického zobrazení scény může být vyžadováno velké množství světelných zdrojů. Stínovací funkce obvykle vysílá jeden stínový paprsek pro každý světelný zdroj a testuje jeho zastínění. Obyčejně tak klesá rychlost zobrazení s počtem světel lineárně. Testování stínových paprsků je proto vhodné nějakým způsobem urychlit. Podobně jako u primárních paprsků lze použít heuristiky k odhadu toho, které zdroje jsou zastíněny, či která tělesa mohou v daném bodě stínit.

Jednou z pokročilejších technik je technika lokálních světelných prostředí, která byla popsána v [5]. Prostor je rozdělen na regiony (například oktalovým stromem) a v jednotlivých regionech je zjišťováno, která světla jsou plně zakryta, vždy viditelná nebo částečně zakryta. Plně zakrytá světla lze pak ignorovat, příspěvek viditelných světel vždy započítat a pro částečně zakrytá světla si vytvořit seznam možných stínících objektů, proti kterým je pak testován stínový paprsek. Takto lze dle autorů algoritmu urychlit výpočet při velkém množství světel 10× až 30×. Množiny světel a stínících těles lze také dopočítávat adaptivně během zobrazování, čímž se uspoří zbytečná analýza v oblastech, které z daného pohledu nejsou pro paprsky dosažitelné.

Kapitola 4

Paralelizace ray traceru

Současné procesory nabízejí různé možnosti paralelního běhu programu. Algoritmus sledování paprsku lze pro tyto procesory uzpůsobit a dosáhnout tak vyrazného zrychlení výpočtu [3]. Paralelně lze v extrému počítat každý primární paprsek, neboť mezi nimi není žádná závislost, sdílí se pouze geometrie.

K maximálnímu využití výkonu procesoru je nutné rozumnět jeho architektuře a programy ručně optimalizovat. Především právě schopnost paralelního zpracování programu vyžaduje speciální úpravy algoritmů.

Dbát musíme také na způsob sekvenčního zpracování instrukcí. Nejvýhodnější jsou jednoduché, ploché algoritmy s minimem větvění. Obzvlášť vnitřní smyčky algoritmů se vyplatí pečlivě ladit tak, aby zde bylo co nejméně náročných operací jako je dělení. Často lze některá data předpočítat mimo vnitřní smyčku a tím doosáhnout urychlení celého algoritmu.

Na rychlost programu má velký vliv také způsob uspořádání dat. Nevhodně strukturovaná data zpomalují program tím, že se méně využije cache procesoru a často dochází k přístupu do pomalé hlavní paměti. Využít můžeme i speciální instrukci *prefetch*, která dopředu načte data do řádku cache.

Procesor může poskytovat dva hlavní druhy paralelizace, *multi-threading* a SIMD (Single Instruction, Multiple Data) instrukce. Tzv. *out-of-order* vykonávání instrukcí ponechme stranou, protože v jazycích vyšší úrovně toto nemůžeme příliš ovlivnit a záleží spíše na překladači a samotném procesoru.

Paralelní běh ve více vláknech

Multi-threading poskytuje na úrovni procesoru podporu běhu více vláken paralelně. Tuto schopnost mají především vícejádrové procesory. Využití spočívá v úpravě programu tak, aby náročné části výpočtu mohli běžet paralelně ve více vláknech. V případě ray tracingu nějakým způsobem rozdělíme jednotlivé primární paprsky mezi tyto vlákna. Tímto způsobem lze s dvojnásobným počtem procesorů snadno dosáhnout prakticky dvojnásobného urychlení ray traceru.

V případě mého ray traceru je na začátku vytvořen požadovaný počet vláken, které nazývám *workery*. Ve smyčce jsou potom generovány vzorky (třídou Sampler, viz 5.3) do fronty vzorků, ze které si každý worker odebírá balíky vzorků pro zpracování. Přístup k frontě je synchronizován pomocí mutexů. Aby tato synchronizace příliš nebrzdila výpočet, je potřeba nastavit dostatečnou velikost balíku vzorků pro odběr workerem.

Balík vzorků je následně sekvenčně zpracováván. Každý vzorek se podle kamery přetransformuje na paprsek a ten je zpracován funkcí raytrace. Vypočítané barvy jsou nakonec

poslány zpět Sampleru, opět naráz celý balík. Mutexy jsou zamykány jen při získávání vzorků z fronty a při posílání vysledků Sampleru. Časově nejnáročnější část, tedy vlastní výpočet sledování paprsků z celého balíku, pak může běžet zcela paralelně ve všech vláknech.

4.1 Sledování paprsků po svazcích

Většina moderních procesorů poskytuje v nějaké formě instrukce SIMD. Jde o možnost paralelního zpracování více datových položek jedinou instrukcí. K práci s takovými vícepoložkovými daty slouží speciální vektorové registry.

Zabývat se budeme pouze instrukcemi SSE (Streaming SIMD Extensions) od Intelu. Jiné procesory nabízejí velmi podobné sady SIMD instrukcí. SSE umožňuje především počítání aritmetických operací na čtyřech 32bitových reálných číslech najednou.

Některé překladače sice nabízejí tzv. auto-vektorizaci, tedy automatickou transformaci částí programu do SIMD formy. To však selhává v případě smyček s potenciálními vnitřními závislostmi, tedy v případech kdy překladač nedokáže nebo nemůže posoudit závislost vstupu jednotlivých iterací na výstupech předchozích. Problémem jsou především ukazatele, které můžou odkazovat libovolné místo v paměti.

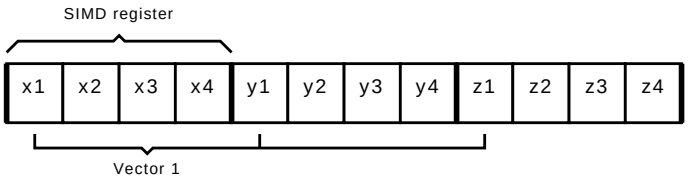
V ray traceru lze SSE instrukce nejefektivněji využít k paralelnímu sledování čtyř paprsků najednou. Takto upravený algoritmus sledování paprsku nazvěme sledováním svazků paprsků (ray packet tracing).

Jinou variantou SSE ray traceru je paralelní výpočet průsečíku každého paprsku se čtyřmi trojúhelníky. Tento algoritmus však neumožňuje urychlit procházení vyhledávací struktury, které také spotřebovává značnou část výpočetních prostředků. Mohl by však být výhodný při velkém množství rozptýlených sekundárních paprsků, což je případ, ve kterém ray packet tracing prakticky selhává.

Důležitým pojmem při sledování paprsků po svazcích je *koherence*. Tu lze popsat jako vzájemnou závislost jednotlivých paprsků, tedy především blízkost jejich počátku a podobnost směru. Koherentní paprsky mají velkou pravděpodobnost, že budou procházet stejnými buňkami prostorového indexu a střetávat se se stejnými objekty.

K sledování čtyř paprsků najednou musíme upravit celý ray tracer. Všechny funkce v hlavním algoritmu by měli podporovat svazky paprsků.

Vzhledem k povaze SIMD instrukcí budeme potřebovat novou strukturu pro vektor tří reálných čísel, která uspořádá čtyři takové vektory podle SIMD registrů. Tedy každou položku původního vektoru nahradíme polem čtyř čísel (viz obrázek 4.1). Svazek paprsků pak bude obsahovat dva tyto packety vektorů.



Obrázek 4.1: Struktura VectorPacket, obsahující čtyři vektory uspořádáné do třech SIMD registrů

Primární paprsky lze snadno generovat do svazků po čtyřech tak, aby byly koherentní. Všechny primární paprsky mají stejný počátek (oko) a od něj se rozptylují v přesně defi-

nované matici, takže můžeme snadno vzít vždy čtyři sousední paprsky v uspořádání 2x2 a vytvořit z nich svazek.

Stínové paprsky budou rovněž dostatečně koherentní, směrem k světelnému zdroji se vždy sbíhají.

Problém představují sekundární paprsky, tedy lomené a odražené. V tomto případě je rozptyl velký, svazky se již „trhají“. Sekundární paprsky je ale možné akumulovat do fronty a zde přeuspořádat do nových svazků.

Pro svazky paprsků je potřeba upravit všechny tři hlavní části algoritmu sledování paprsku. Tedy průchod akcelerační strukturou, výpočet průsečíku s objekty a stínovácí funkci.

SSE intrinsiky

Rozšířit program o SSE instrukce můžeme s pomocí vestavěného assembleru. To je ale z mnoha hledisek nevýhodné. Takový kód navíc nelze snadno přenáset mezi platformami, i na úrovni jednotlivých překladačů nalezneme několik různých způsobů zápisu kódu v assembleru.

Intel pro své překladače nabízí možnost používání strojových instrukcí ve formě funkcí jazyka C [8]. Tyto funkce, nazývané *intrinsiky,* jsou překládány přímo na instrukce procesoru, ale usnadňují nám práci s registry. Místo SIMD registrů se použijí jen speciální proměnné a překladač se sám postará o optimální využití skutečných registrů.

SSE intrinsiky se staly v podstatě standardem a podporují je všechny důležité překladače.

Výpis kódu 4.1: Ukázka implementace skalárního součinu čtyř vektorů pomocí SSE

```
struct VectorPacket
{
  __m128 mx, my, mz;
};

__m128 dot(const VectorPacket &a, const VectorPacket &b)
{
  return
    _mm_add_ps(
      _mm_add_ps(_mm_mul_ps(a.mx, b.mx), _mm_mul_ps(a.my, b.my)),
      _mm_mul_ps(a.mz, b.mz)
    );
};
```

K dosažení maximální efektivity SIMD kódu je nutné minimalizovat množství pomalých instrukcí pro přesun dat mezi SSE registry a pamětí. Jsou to hlavně instrukce pro načítání paměti do registru a ukládání zpět do paměti (load a store). Ukazatele pro použití s těmito instrukcemi by navíc měly odkazovat vždy bloky 16 bytů (16B alignment). Instrukce pro načítání a ukládání na libovolné místo v paměti jsou sice také k dispozici, avšak ty jsou pomalejší, musí nejdříve data zarovnat na 16B.

Problémem při SIMD zpracování dat je větvení, které znamená nucené ukončení bloku SIMD instrukcí, případný skok a následné pokračování. To vše je samozřejmě pomalé. Výsledek SSE instrukcí porovnání lze pro použití v podmínce jazyka C připravit instrukcí `_mm_movemask_ps`, která posune znaménkové bity všech čtyř položek do prvních čtyřech bitů hotnoty typu int.

Některé případy větvení můžeme nahradit sérií SSE instrukcí. Například podmínku 4.2 lze zapsat kódem z výpisu 4.3. Zde jsme si také vytvořili velmi užitečnou kompozitní operaci

select.

Výpis kódu 4.2: Podmíněné přiřazení ideální pro přepis do SSE

```
if (a > 0)
        c = a
else
        c = b;
```

Výpis kódu 4.3: SSE kód pro výběr proměnné podle výsledku podmínky

```
__m128 select(__m128 mask, __m128 a, __m128 b)
{
  return _mm_or_ps(_mm_and_ps(mask, a), _mm_andnot_ps(mask, b));
};

c = select(_mm_cmp_gt(a, _mm_setzero_ps()), a, b);
```

Dále se věnujme jednotlivým algoritmům ray tracingu a jejich úpravou pro SSE.

4.1.1 Procházení kd-stromu svazkem paprsků

Prostorový index je fundamentální součástí rychlých algortimů pro ray tracing. Kd-tree je jedna z takových struktur, která je dostatečně jednoduchá a zároveň velmi efektivní. Algoritmus průchodu kd-stromem sestává z jedné smyčky s pomocným zásobníkem. Rozšíření tohoto algoritmu pro svazky paprsků je přímočaré.

Základní myšlenku lze vyjádřit tak, že pokud jeden paprsek z balíku navštíví daný uzel stromu, bude se tento uzel procházet s celým svazkem. Důležitá je koherence. Budou-li paprsky ve svazku dostatečně koherentní, je velká pravděpodobnost, že navštíví stejné uzly stromu.

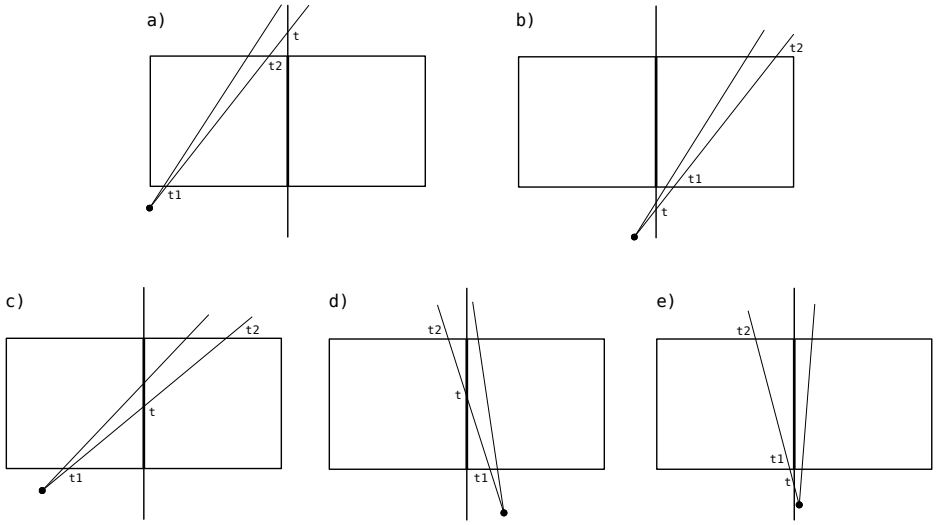
Algoritmus začíná, stejně jako jeho verze pro jeden paprsek (viz 3.2.2), zjištěním průsečíků všech paprsků ve svazku s obalovým kvádrem scény. Pokud jsou nalezeny průsečíky alespoň pro jeden z paprsků, algoritmus pokračuje, jinak končí s tím, že svazek neprotíná žádný objekt.

Průsečíky s obalovým kvádrem uložíme podle velikosti parametru paprsků jako blízký (vektor t_1) a vzdálený (vektor t_2). Následně v cyklu procházíme strom od kořenového uzlu. Průsečíky s dělící rovinou opět spočteme jako parametr paprsků a uložíme do vektoru t.

Porovnáním t_1 a t_2 s t dojdeme ke čtyřem možným případům (obrázek 4.2). Pokud pro všechny paprsky platí $t_1 \leq t$ a zároveň $t_2 \leq t$, potom můžeme pokračovat levým uzlem (případ a). Jsou-li naopak oba parametry t_1, t_2 všech paprsků větší jak t, pokračujeme pravým uzlem (případ b). Ve zbylých případech musíme procházet oba poduzly, je nutné pouze rozhodnout správné pořadí. Jeden z uzlů pak uložíme do zásobníku a druhým pokračujeme.

Jsou-li některým z paprsků protnuty oba poduzly, musíme rozhodnout, který uzel je pro všechny paprsky bližší a který vzdálenější. Intuitivně lze podmínku stanovit tak, že je-li pro všechny paprsky $t_1 \leq t$ nebo $t_2 \geq t$, potom procházíme nejprve levý, následně pravý uzel (případ c). Podobně je-li pro všechny paprsky $t_1 \geq t$ nebo $t_2 \leq t$ začneme pravým a do zásobníku uložíme levý (případ d).

Zde ovšem mohou nastat případy, kdy ani jedna podmínka nebude platit, například případ e na obrázku 4.2. I zde lze ale stanovit takové pořadí procházení poduzlů, že u žádného paprsku nedojde k problému. Vycházíme-li z faktu, že všechny paprsky mají společný po-



Obrázek 4.2: Přehled možných případů průseku uzlu kd-stromu svazkem paprsků

čátek (platí pro primární paprsky), můžeme jako bližší určit ten uzel, na jehož straně dělící roviny se počátek paprsků nachází.

Všimněme si, že paprsky na obrázku 4.2*e* mají odlišný směr – jeden míří vlevo, druhý vpravo. Pokud tomuto zabráníme, můžeme tento případ zcela eliminovat. Dosáhnout toho lze přeuspořádáním paprsků mezi svazky podle znamének směrových vektorů.

Vzhledem k tomu, že tímto algoritmem navštěvujeme uzly i s paprsky, které je neprotínají, musíme tyto případy ošetřit. Takové paprsky rozpoznáme podle nesplněné podmínky $t_1 < t_2$. Uložíme-li výsledek této podmínky jako *masku*, lze pak snadno odmaskovat tyto neplatné paprsky operací *and*.

Dosáhne-li algortimus listového uzlu, kontroluje se průsečík s objekty v tomto uzlu. Test provádíme vždy se všemi čtyřmi paprsky naráz upraveným algoritmem pro získávání průsečíku s objekty.

4.1.2 Hledání průsečíků trojúhelníku se svazkem paprsků

Na algoritmus procházení kd-stromu můžeme přímo navázat hledání průsečíků s paprsky, které ve svazku dorazily až do listového uzlu stromu.

Využijeme dříve prezentovaný algoritmus využívající barycentrických souřadnic (viz 3.1). Algoritmus lze rozšířit pro svazek paprsků použitím SSE instrukcí dle výpisu 4.4. Vstupem je svazek paprsků, obsahující dvě struktury VectorPacket s počátky a směrovými vektory paprsku (složky pojmenované *o* a *dir*). Ve struktuře trojúhelníku jsou připraveny konstantní hodnoty potřebné pro výpočet baricentrických souřadnic. Výstupem jsou vzdálenosti průsečíků v parametru *dists* a jako výstupní hodnota funkce je vrácena maska obsahující jedničky na místech paprsků, pro které byl průsečík nalezen.

Výpis kódu 4.4: SSE kód hledání průsečíků trojúhelníku se svazkem paprsků

```
__m128 Triangle::intersect_packet(const RayPacket &rays, __m128 dists) const
{
        register const int u = modulo3[k+1];
```

```
        register const int v = modulo3[k+2];
        __m128 mask;

        __m128 t = _mm_div_ps(_mm_sub_ps(_mm_sub_ps(
                _mm_sub_ps(_mm_set_ps1(nd), rays.o[k]),
                _mm_mul_ps(_mm_set_ps1(nu), rays.o[u])
                ), _mm_mul_ps(_mm_set_ps1(nv), rays.o[v])),
                _mm_add_ps(rays.dir[k],
                _mm_add_ps(_mm_mul_ps(mset1(nu), rays.dir[u]),
                _mm_mul_ps(_mm_set_ps1(nv), rays.dir[v])))
                );

        mask = _mm_and_ps(_mm_cmplt_ps(t, dists), _mm_cmpge_ps(t, _mm_setzero_ps()));
        if (!_mm_movemask_ps(mask))
                return mask;

        __m128 hu = _mm_sub_ps(_mm_add_ps(rays.o[u],
                _mm_mul_ps(t, rays.dir[u])), _mm_set_ps1(A->P[u]));
        __m128 hv = _mm_sub_ps(madd(rays.o[v],
                _mm_mul_ps(t, rays.dir[v])), _mm_set_ps1(A->P[v]));
        __m128 beta = _mm_add_ps(_mm_mul_ps(hv, _mm_set_ps1(bnu)),
                _mm_mul_ps(hu, _mm_set_ps1(bnv)));

        mask = _mm_and_ps(mask, _mm_cmpge_ps(beta, _mm_setzero_ps()));
        if (!_mm_movemask_ps(mask))
                return mask;

        const mfloat4 gamma = _mm_add_ps(_mm_mul_ps(hu, _mm_set_ps1(cnv)),
                _mm_mul_ps(hv, _mm_set_ps1(cnu)));

        mask = _mm_and_ps(mask, _mm_and_ps(mcmpge(gamma, _mm_setzero_ps()),
                _mm_cmple_ps(_mm_add_ps(beta, gamma), mOne)));
        if (!_mm_movemask_ps(mask))
                return mask;

        dists = select(mask, t, dists);
        return mask;
}
```

Důležité jsou tři ukončovací podmínky. První odfiltruje trojúhelníky za počátkem paprsků, nebo dále než jsou současné nejblíže nalezené trojúhelníky. Další dvě podmínky pak kontrolují, zda některé z paprsků protínají plochu trojúhelníků.

Ve všech případech pokračujeme, pokud alespoň pro jeden trojúhelník podmínka vyhovuje. Tím vzniká redundance, stejně jako u SSE algoritmu průchodu kd-stromem, takže získaný výkon se částečně koriguje. Přesto je zpracování čtyř paprsků naráz výhodné.

4.1.3 Stínovací funkce pro svazek

Ze svazků paprsků těží i stínovací funkce, která většinou vyžaduje poměrně náročné výpočty. S pomocí SSE opět počítáme vše naráz pro celý svazek paprsků.

Rovněž pro stínové paprsky můžeme s výhodou využít hledání průsečíku pro celý svazek, který je díky bodovým zdrojům světla koherentní. Vzhledem k tomu, že nám v tomto případě stačí jakýkoliv průsečík, ne nutně nejbližší, bylo by možné upravit i způsob procházení kd-stromu a hledání vlastních průsečíků. Taková optimalizace ovšem neslibuje příliš velký zisk výkonu, takže jsem ji ponechal stranou.

Problémem jsou sekundární paprsky, které zpravidla zvyšují rozptyl svazku a narušují tak koherenci. Kdybychom chtěli i tyto paprsky dále sledovat po svazcích, museli bychom je přeuspořádat do lepších svazků podle podobnosti směrových vektorů. To je ale vzhledem k rekurzívní povaze algoritmu obtížné.

Stínovací funkce obyčejně předpokládá, že podle vyslaného sekundárního paprsku získá ihned hodnotu barvy a může pokračovat. V případě přeskupování paprsků by bylo nutné tuto funkci nějakým způsobem pozastavit nebo jí upravit tak, aby prováděla výpočet ve více fázích – první, přípravná fáze by vygenerovala sekundární paprsky a návratové hodnoty by byly zpracovány ve druhé fázi. Vzhledem k možné hloubce algoritmu by to ale celé muselo probíhat ve vlnách a otázkou je, zda se taková technika dostatečně vyplatí, nebude-li režie front paprsků příliš velká.

Kapitola 5

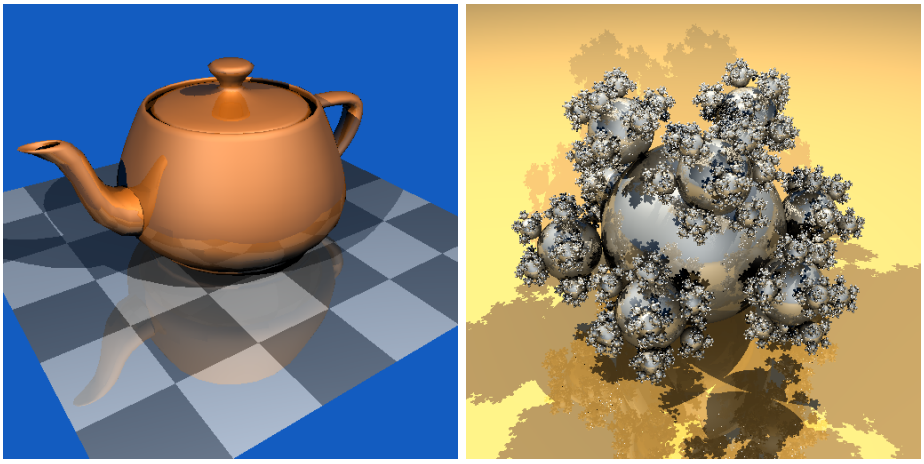
Implementace – ray tracer Pyrit

Tato kapitola se věnuje realizaci softwaru pro sledování paprsku implementujícího techniky z předchozích kapitol.

Program by měl být dostatečně univerzální a přesto využívat optimalizace pro současné procesory, bez kterých lze jen těžko dosáhnout vynikajícího výkonu. Volba programovacího jazyka pro renderovací jádro je tedy prostá, z nynějších možností vyhovuje pouze jazyk C++. Získáme tak možnost objektově orientovaného návrhu programu a výhody s tím spojené, ale i možnost nízkoúrovňových optimalizací. Osvědčilo se mi též využívání přetěžování operátorů jazyka C++, které je v případě 3D grafického softwaru velmi výhodné. Lze tak intuitivně pracovat s vektory, maticemi, kvaterniony i dalšími matematickými strukturami.

Ray tracer implementovaný v rámci této práce jsem nazval *Pyrit*. Projektu jsem vytvořil webovou prezentaci, kde je umístěn repozitář s aktuálními zdrojovými kódy, spolu s generovanou dokumentací kódu, přeloženými ukázkovými programi a dalšími informacemi. Stránka je zveřejněna na adrese `http://wiki.fiction.cz/Pyrit`.

Součástí projektu je knihovna pro C++, modul pro Python a ukázkové programy a scény v obou jazycích. Některá dema jsou interaktivní – využívají ray tracer ke kontinuálnímu zobrazování scény.



Obrázek 5.1: Standardní scény *teapot* a *sphereflake* zobrazené ray tracerem Pyrit.

Dále v této kapitole nahlédneme do vnitřní struktury programu a ukážeme si, co všechno Pyrit již dokáže. Nejdříve se ale podívejme na používané nástroje.

5.1 Nástroje použité při implementaci

Veškerý použitý software je dostupný pod některou z open source licencí. Nejdůležitějším nástrojem pro vývoj ray traceru Pyrit byl balík GNU Compiler Collection, který poskytl kvalitní překladač jazyka C++, standardní knihovnu a další podpůrné nástroje, především debugger a profiler. Při ladění kódu jsem využil také program Valgrind, který pomohl při hledání chyb v alokaci paměti.

Program jsem otestoval s více překladači a v různých operačních systémech. Překlad by měl bez problému proběhnout s nástroji Intel C++ Compiler, Microsoft Visual C++ Compiler a Minimal GNU for Windows (mingw).

Skriptovací jazyk Python

Ray tracer Pyrit lze používat jako C++ knihovnu, s jejíž pomocí lze z instancí nabízených tříd sestavit scénu a následně spustit proces renderování. Samozřejmě je možné napsat i proceduru načítající nějaký souborový formát s popisem modelů či scén – mezi ukázkovými programy je i taková funkce, která načítá soubory ve formátu PLY.

Nevýhodou tohoto přístupu je nutnost mít k dispozici kompletní vývojové prostředí, kterým programy/scény překládáme. Přijatelnější variantou je nabídnout univerzální souborový formát, z kterého se bude scéna načítat. Takový formát může být velmi složitý, má-li nabídnout všechny schopnosti ray traceru. Kvůli větší flexibilitě by měl podporovat i skriptování.

Místo psaní interpretu nového či existujícího jazyka pro popis scény jsem zvolil interpret existující. Konkrétně interpret jazyka Python, který umožní napojit objekty z C++ přímo na objekty tohoto jazyka a zachovat tak schéma programu.

V Pythonu bude možné scénu zapsat podobným způsobem jako v C++, ale ušetříme si nutnost opakovaného překládání po úpravě scény. Síla tohoto jazyka navíc dovolí snadno naprogramovat parsery různých datových formátů a případně načítat celé scény vygenerované v modelovacích programech. Pro jazyk Python je dostupné množství různých knihoven, například pro postprocessing obrazu může být užitečná knihovna PIL (Python Imaging Library).

Chceme-li propojit program v C++ s jazykem Python, máme dvě možnosti. Buď přilinkovat interpret Pythonu do programu nebo program napsat jako knihovnu a vytvořit modul pro Python. První možnost by sice uživateli ušetřila nutnost mít nainstalován interpret Pythonu, ale druhá možnost je flexibilnější a pro tento případ vhodnější.

Propojení s Pythonem je v Pyritu zajištěno jedním dodatečným zdrojovým souborem s popisem propojení na objekty jazyka C++. Překladem tohoto souboru a slinkováním s knihovnou ray traceru pak vznikne modul pro Python, který lze již běžným způsobem importovat do Pythonovských programů.

Skript v Pythonu potom připraví či načte scénu z datového souboru a předá řízení ray traceru v nativním kódu, který na základě této scény efektivně spočítá obrázek a data předá zpět řídícímu skriptu nebo uloží do souboru. Skript může s výslednými daty dále libovolně naložit, například je zobrazit v okně. Skript může ray tracer volat i opakovaně a případně reagovat na vstup uživatele.

SCons build system

Jedním z důležitých problémů je také otázka překladu a sestavení programů a knihoven v projektu. Vhodný konstrukční systém by měl být multiplatformní a podporovat více různých

překladačů. Vzhledem k již využívanému jazyku Python se ukázal dobrou volbou systém SCons, který je sám napsán v tomto jazyce. V Pythonu se zde píší i soubory nahrazující tradiční *makefile*.

Součástí systému SCons je také schopnost konfigurace ve stylu systému *autoconf*, tedy uzpůsobení překladu podle parametrů a detekovaných knihoven v daném prostředí. Navíc jsem přidal i detekci typu procesoru a volbu vhodných voleb kompilátoru pro daný procesor. Tyto záležitosti obstarává zvláštní utilita v jazyce C, která je během konfigurační fáze přeložena a spuštěna, získané volby pak obratem použity při překladu.

Další použité nástroje

Důležitým pomocníkem při práci na složitějším projektu je verzovací systém. Použil jsem systém Bazaar VCS, který je multiplatformní, snadno použitelný a ke své funkci potřebuje pouze běžný adresář. Je napsán v Pythonu.

K implementaci vláken využívám knihovnu Pthreads. Dále jsem použil *libpng* pro ukládání obrázku ve formátu PNG a knihovnu SDL pro interaktivní dema.

5.2 Přehled funkce ray traceru

Vzhledem k charakteristice algoritmu sledování paprsku je výhodné ray tracer psát v objektově orientovaných jazycích a využívat jejich předností. Užitečný je především koncept polymorfismu a zapouzdření. Postačí tak několik generických tříd s dobře definovaným rozhraním, z nichž odvozené třídy již implementují konkrétní algoritmy či objekty trojrozměrného světa.

Použití ray traceru Pyrit je přímočaré, přesto nabízí široké možnosti. Nezávisí na tom, použijeme-li přímo C++ API nebo modul v Pythonu. Vždy je k dispozici sada tříd, která poskytuje danou funkcionalitu. Z těchto tříd poskládáme scénu požadovaných parametrů a spustíme proces renderování (nebo-li stínování).

Příprava scény

K dispozici máme třídy reprezentující různé objekty ve scéně a také algoritmy (viz 5.3). Stavba scény spočívá v tvorbě instancí těchto tříd a jejich vzájemného provázání.

Základní třídou je Raytracer, která uchovává seznam všech objektů ve scéně a také nastavení různých globálních parametrů. Přidání objektu do scény spočívá ve vytvoření instance třídy tohoto objektu a její vložení do instance Raytraceru pomocí příslušné metody. Objekty samy mohou obsahovat odkazy na instance dalších tříd, například materiál a texturu.

Hlavní algoritmus má však povědomí pouze o několika generických třídách, například Sampler, Container, Shape, Material. Třídy specifických vlastností jsou pak od těchto základních odvozené a implementují jejich virtuální metody, které jsou z hlavního algoritmu volány.

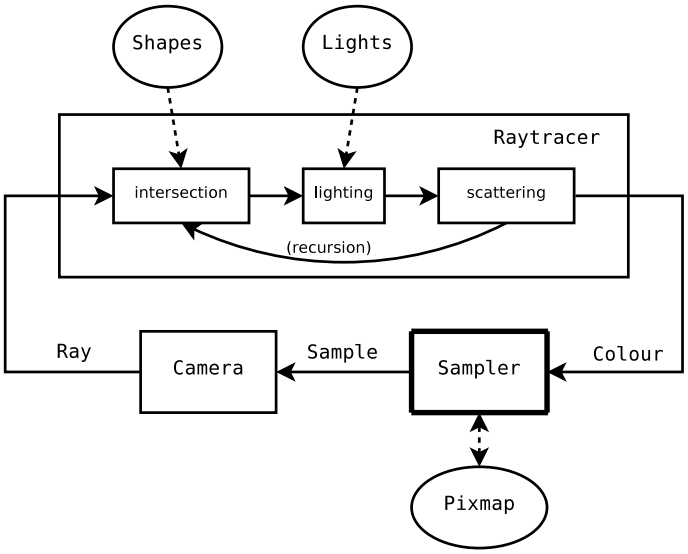
Tímto způsobem nabízí ray tracer Pyrit vysokou míru flexibility, avšak k rozšiřování je třeba dobře rozumnět rozhraní daných objektů.

Máme-li připravenu scénu, můžeme spustit hlavní cyklus zobrazovacího algoritmu, zavoláním metody render() objektu Raytracer.

Proces renderování

Hlavní cesty toku dat během procesu zobrazování jsou zobrazeny v diagramu 5.2. Na začátku i konci koloběhu je objekt třídy Sampler. Ten generuje vzorky (*samply*), tedy body v rovině obrazovky a později přijímá výsledné barvy v těchto bodech. Rozmístění bodů může být libovolné, takže ve třídách odvozených od třídy Sampler lze implementovat různé metody nadvzorkování a podvzorkování. Ke každému vygenerovanému vzorku dostane později zpět barvu, kterou může dale zpracovat či přímo uložit jako pixel výsledného obrázku (pixmapy).

Proces převodu souřadnic bodu v rovině obrazovky (vzorku) na barvu je vlastním algoritmem sledování paprsku.



Obrázek 5.2: Schéma procesu renderování

Prvním požadavkem algoritmu je získat paprsek procházející daným bodem obrazovky. Převod bodu na paprsek sprostředkovává kamera. Na základě nastavených parametrů, jako jsou například souřadnice oka (pozorovatele), směr pohledu a vektor „nahoru“, je vypočítán vektor počátku a směrový vektor paprsku. Tuto funkci zajišťuje třída Camera, případně její potomci.

Nyní máme k dispozici parametry paprsku a scénu složenou ze základních objektů a světel. Barvu z těchto informací získáme nalezením průsečíku s nejbližším objektem, výpočtem osvětlení a případných odrazů a lomů. Hledání průsečíku s objekty zajišťuje třída Container, na kterou se blíže podíváme později.

O výpočet osvětlení se stará osvětlovací funkce, jejíž vstupem je bod na povrchu tělesa, normála v tomto bodě, seznam světelných zdrojů a parametry materiálu. Tuto funkci jsem implementoval přímočaře a používá Phongův osvětlovací model, lze ale použít obecnější přístup a osvětlování implementovat pomocí rozšiřitelných abstraktních tříd. Materiál také může obsahovat texturu, tímto se budeme podrobněji zabývat dále.

V této chvíli máme již k dispozici barvu bodu a zobrazují se stíny od bodových zdrojů světla. Důležitou vlastností metody sledování paprsku je také odraz a lom světla. Ty jsou opět implementovány s pomocí parametrů materiálu, v tomto případě odrazivost, propustnost a index lomu. Odraz a lom lze obecně nazvat rozptylem světla a implementován je v rozptylovací funkci (scatter).

Zde jsou vytvářeny sekundární paprsky (lomené či odražené), které znovu prochází tímto renderovacím procesem od fáze hledání průsečíku – nastává rekurze. Barva původní je pak sloučena s barvami získanými renderováním sekundárních paprsků podle parametrů materiálu.

Výsledná barva se dostává zpět k Sampleru. Stejný proces pak nastává pro další vygenerované vzorky, až dokud žádný další není potřeba. V pixmapě v objektu Sampler je pak sestaven celý renderovaný obrázek.

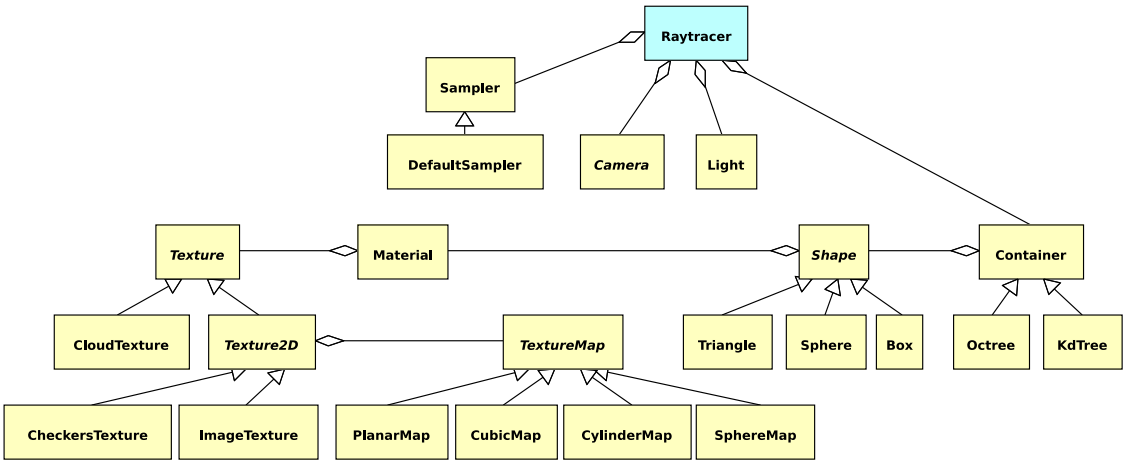
5.3 Schéma a přehled tříd

Základními kameny ray traceru je několik pomocných struktur. Především je to třída Vector, která zapouzdřuje vektor tří reálných čísel a související operace. Tato třída slouží pro ukládání bodových a směrových vektorů, ale také k reprezentaci barvy v klasickém uspořádání RGB.

V C++ lze s výhodou využít přetížení operátorů pro implementaci operací vektoru. Skalární a vektorový součin však raději implementujeme jako obyčejné metody, zneužívání nesouvisejících operátorů by mohlo snížit čitelnost kódu a zapříčinit špatně zjistitelné chyby. Operátor hvězdička (*) tak bude sloužit pouze k násobení vektoru po složkách, které je užitečné při práci s barvami.

Všudypřítomným elementem sledování paprsku je samotný paprsek. S využitím třídy vektoru paprsek vytvoříme složením dvou vektorů, které určí počátek a směr paprsku.

Na obrázku 5.3 je schéma tříd a jejich vzájemných vztahů. Podívejme se nyní na některé z nich podrobněji.



Obrázek 5.3: Schéma tříd

Vzorkování, kamera

Základní třídou a vstupním bodem celého algoritmu je **Raytracer**. Tento objekt nese informace o celé scéně i parametrech renderování. Agregovány zde jsou objekty scény, světla a kamera. Má-li nějaký objekt mít vliv na výsledek zobrazování, musí k němu být nějaká cesta z objektu Raytracer.

Raytracer obsahuje také dva důležité atributy, barvu pozadí a maximální hloubku rekurze. Barva pozadí se použije, pokud paprsek nezasáhne žádný objekt.

Třídy Sampler a Camera ovlivňují způsob generování paprsků.

Sampler řídí a abstrahuje vzorkování obrazu. Počítačové obrazovky mají formu pravidelné mřížky bodů, renderovaný obraz je tedy v těchto bodech nějakým způsobem vzorkován. Přímočaré řešení je jeden vzorek pro každý bod, avšak z důvodu potlačení aliasingu je vhodné implementovat i složitější metody.

Třídy odvozené od třídy Sampler mohou generovat souřadnice vzorků různým způsobem, rozhraní tohoto objektu vyžaduje pouze implementaci metody pro získání dalšího vzorku a tedy uchovávání stavu vzorkování. Druhá metoda pak slouží pro předání výsledné barvy vzorku, spolu s jeho dříve vygenerovanými souřadnicemi. Sampler tuto barvu dále zpracuje a postupně tak vytváří obraz. Ten je nakonec přečten volajícím programem z pixmapy objektu Sampler. Tato pixmapa může být zpřístupněna uživateli i během procesu vykreslování a tedy například průběžně zobrazována na obrazovce.

Třída **Camera** transformuje souřadnice vzorku na paprsek. Zde je implementována tradiční kamera pro ray tracing, vycházející z principu štěrbinového fotoaparátu. Odvozené třídy mohou implementovat i složitější algoritmy a například simulovat čočky objektivu, hloubku ostrosti a další specifika snímacích zařízení.

Tvary, kontejnery

Scéna je složena z elementárních objektů, tzv. tvarů. Ray tracer Pyrit podporuje tři základní tvary: trojúhelník, koule a kvádr. Pokud bychom chtěli dosáhnout maximální optimalizace i za cenu poněkud snížené rozšiřitelnosti programu, bylo by možné podporovat jediný tvar – trojúhelník. Vše ostatní pak lze na trojúhelníkové modely převést teselací. Domnívám se však, že režie nutná k vytvoření obecného rozhraní pro různé tvary nemá velký vliv na celkovou rychlost výpočtu. Zaměřit se na optimalizace s trojúhelníkovými modely lze samozřejmě i pokud povolíme jiné tvary.

Abstraktní třída pro tvar se jmenuje **Shape**. Nejdůležitější z této třídy je metoda *intersect*, která počítá průsečík daného tvaru s paprskem. Lze zároveň říct, že implementace této metody určuje tvar objektu v prostoru. Normálu v libovolném bodu na povrchu objektu počítá metoda *normal*.

Důležitá je také metoda *intersectBBox*, používaná pro techniky dělení prostoru (kapitola 3.2). Počítá průsečík tvaru a osově zarovnaného kvádru. Takový kvádr tvoří podprostory většiny algoritmů dělení prostoru. Tato metoda je využita pro zjišťování, zda daný tvar do podprostoru spadá.

Materiál objektu určuje atribut *material*. Ten obsahuje ukazatel na třídu Material, na kterou se podíváme dále.

Každý podporovaný tvar musí implementovat všechny abstraktní metody třídy Shape.

K implementaci technik dělení prostoru (viz 3.2) slouží třída **Container**. V ní jsou sdruženy objekty a definován obalový kvádr, který je všechny objímá. Velikost obalového tělesa může být počítána jednorázově nebo při vkládání tvarů do kontejneru. Třída poskytuje metodu *optimize*, která provádí výpočet akceleračních dat a metodu *intersect* hledající nejbližší průsečík předaného paprsku s tělesem z kontejneru. Na základě těchto dvou virtuálních metod lze implementovat libovolnou techniku dělení prostoru. Třída Container sama o sobě hledá průsečík bez využití jakékoliv akcelerační struktury, tedy testováním všech obsažených tvarů.

Různé akcelerační techniky jsou implementovány jako potomci třídy Container. V současnosti jsou to třídy **Octree** a **KdTree**, implementující oktalový strom a kd-strom.

Osvětlování a texturování

Důležitou součástí ray traceru je stínovací funkce, která určuje vzhled povrchu objektu. Zde může být implementováno mnoho různých technik pro dosažení realistického zobrazení povrchu. Stínovací funkce má k dispozici bod a normálu povrchu a seznam světelných zdrojů.

Světelné zdroje v klasickém ray traceru jsou pouze bodové a tudíž vytvářejí ostré stíny. Takové světlo implementuje třída **Light**. Zdroj je specifikován třemi atributy: polohou, barvou a energií. Z důvodu větší obecnosti jsem nahradil tři Phongovy atributy (viz 2.3) jediným parametrem udávajícím energii či sílu světelného zdroje. To obyčejně stačí a je tak umožněno případně využít i jiné osvětlovací modely. Parametry Phongova osvětlovacího modelu mohou být přesně doladěny pomocí parametrů materiálu jednotlivých objektů.

Od základní třídy Light mohou být odvozeny další typy světelných zdrojů, například směrové světlo nebo nebodové světelné zdroje pro zobrazování měkkých stínů. Každý druh světelného zdroje ale musí být podporován stínovací funkcí.

Existuje i pokročilejší technika, umožňující vyšší flexibilitu. Jedná se o definování světel na základě speciálního shaderu, zvaného *light shader*. Ten na základě průsečíku a normály povrchu počítá směrový vektor a intenzitu světla. Polohu světelného zdroje pak vůbec není třeba znát.

Osvětlování je v ray traceru Pyrit počítáno podle Phongova osvětlovacího modelu (viz 2.3). V osvětlovací funkci se zkontroluje viditelnost jednotlivých světelných zdrojů a započítá jejich příspěvek. Celková barva je zkomponována z těchto příspěvků a základní barvy objektu v daném bodě povrchu. Základní barva může být buď konstantní nebo definována texturou.

Materiál objektu definuje třída **Material**. Obsahuje barvu či odkaz na texturu, Phongovy konstanty a dále odrazivost *reflectivity* a průhlednost *transmisivity*. Jsou-li tyto parametry větší než nula, nastává generování sekundárních paprsků v rozptylovací funkci. Atribut *refract_index* čili index lomu je používán při výpočtu směru lomeného paprsku.

Barvu povrchu objektu může upravovat textura, popsaná třídou **Texture**. Tato abstraktní třída obsahuje jedinou metodu, *evaluate*, vyhodnocující barvu v bodě povrchu na základě jeho souřadnic v prostoru. Při vyhodnocování se mohou použít rastrová data nebo je barva počítána přímo, pomocí matematických funkcí a šumů (procedurální textury).

Vztah mezi dvourozměrnými rastrovými daty a souřadnicemi bodu v prostoru vyjadřují mapovací funkce **TextureMap**. Pyrit nabízí čtyři třídy implementující různé způsobý mapovaní dle 2.4. Podtřída textury pro použití s těmito mapováními se jmenuje **Texture2D**.

Zatím neimplementovaná třída **UVMap** bude poskytovat mapování textury na trojúhelníkové modely pomocí u, v souřadnic. Tato třída bude ke své činnosti potřebovat odkaz na zpracovávaný trojúhelník, jehož data jsou k výpočtu texturovacích souřadnic potřeba. Pro tento účel bude také potřeba rozšíření vrcholů trojúhelníků (Vertex) o souřadnice u, v (budou zadány uživatelem), z nich se pak interpolací získá souřadnice v rovnině textury.

Rastrové textury jsou representovány třídou **ImageTexture**, která pomocí třídy Pixmap získává rastrová data a vrací barvu na základě souřadnic u, v. Vzhledem k tomu, že souřadnice jsou reálné a data diskrétní, bude vhodné použít interpolaci barev. Rozměry pixmapy mohou být libovolné, není třeba se omezovat na násobky dvou (nepoužíváme-li mipmapy). Souřadnice uvnitř pixmapy se pohybují od nuly do jedné, je-li u nebo v mimo tento rozsah, dojde k opakování textury.

K dispozici jsou dvě ukázkové procedurální textury. **CheckersTexture** implementuje

klasickou „šachovnici“, **CloudTexture** vytváří dojem mraku či kamene pomocí skládaného Perlinova šumu.

5.4 Ukázka použití

Základním uživatelským prostředkem pro používání ray traceru Pyrit je aplikační rozhraní pro Python (dokumentace je v dodatku B). Toto rozhraní je tvořeno vazbou většiny původních C++ objektů do jazyka Python. Dokumentace k rozhraní C++ vygenerovaná systémem Doxygen se nachází na přiloženém CD v adresáři `pyrit/docs/html`.

Následuje komentovaná ukázka tvorby scény, jejího zobrazení a uložení výsledku do souboru (výpis 5.1). To celé ve skriptu v jazyce Python. Třídy raytraceru jsou importovány z modulu `pyrit`.

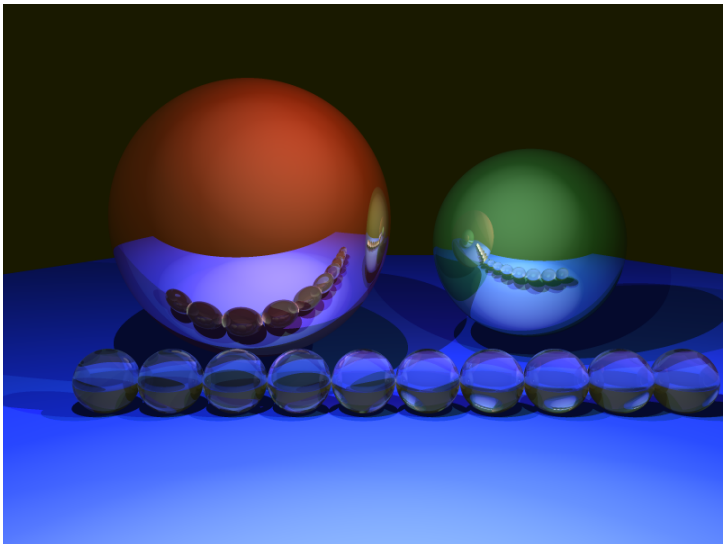
Po spuštění tohoto skriptu získáme obrázek 5.4.

Výpis kódu 5.1: Ukázka skriptu, který vytvoří a zobrazí 3D scénu

```
#!/usr/bin/env python
from pyrit import *

# vytvoříme objekt Raytracer
rt = Raytracer()

# pozadí nastavíme okrové (použije se pro paprsky, které nezasáhnou žádný tvar)
rt.setBgColour((0.1, 0.1, 0.0))

# nastavíme look-at kameru; vektor "up" je implicitně (0,1,0)
rt.setCamera(Camera(eye=(10,4,-6),lookat=(10,3.5,0)))

# vytvoříme hlavní kontejner pro objekty: akcelerační strukturu kd-tree
top = KdTree()

# a nastavíme jej jako aktivní
rt.setTop(top)

# vytvoříme tři světla různých barev
light1 = Light(position=(10.0, 7.0, 3.0), colour=(0.9, 0.3, 0.6))
light2 = Light(position=(8.0, 5.0, 1.0), colour=(0.7, 1.0, 0.3))
light3 = Light(position=(12.0, 8.0, -1.0), colour=(0.8, 0.9, 1.0))

# a přidáme je do scény
rt.addLight(light1)
rt.addLight(light2)
rt.addLight(light3)

# připravíme si materiál pro zem
mat_ground = Material(colour=(0.1, 0.2, 0.9))

# zakážeme odrazy
mat_ground.setReflectivity(0.0)

# vytvoříme zem pomocí kvádru a přidáme ji do scény
ground = Box(L=(-10.0, 0.0, 50.0), H=(30.0, 1.0, -1.0), material=mat_ground)
rt.addShape(ground)

# podobným způsobem připravíme dvě lesklé koule
mat1 = Material(colour=(1.0, 0.2, 0.1))
mat1.setReflectivity(0.7)
bigsphere = Sphere(center=(12.0, 4.0, 6.0), radius=2.5, material=mat1)
rt.addShape(bigsphere)

mat2 = Material(colour=(0.1, 0.4, 0.2))
mat2.setReflectivity(0.6)
```

```
smallersphere = Sphere(center=(6.5, 3.5, 8.0), radius=2.0, material=mat2)
rt.addShape(smallersphere)

# nyní si připravíme materiál imitující sklo
mat3 = Material(colour=(0.9, 0.9, 1.0))
mat3.setPhong(0.2, 1.0, 0.2)
mat3.setReflectivity(0.1)
mat3.setTransmissivity(0.88, 1.5)  # druhý parametr je index lomu

# a vytvoříme řádku skleněných kuliček
for i in range(10):
        sphere = Sphere(center=(5.0 + i, 1.5, 4.0), radius=0.5, material=mat3)
        rt.addShape(sphere)

# scénu máme kompletní, můžeme spustit budování kd-stromu
top.optimize()

# nyní si připravíme sampler s pixmapou o velikosti 800x600
sampler = DefaultSampler(800, 600)
rt.setSampler(sampler)

# povolíme oversampling 9x
sampler.setOversample(2)

# a spustíme proces renderování
rt.render()

# obrázek se zapíše do pixmapy Sampleru, uložíme ho tedy do souboru
sampler.getPixmap().writePNG('demo.png')
```

Možnosti použití jsou široké, k dispozici máme kompletní programovací jazyk včetně knihoven. Získaná obrazová data například nemusíme ukládat do souboru, lze je dále zpracovat. Vytvořit tak lze například program s grafickým uživatelským rozhraním, kde uživatel bude nějakým způsobem upravovat scénu a výsledek se bude přímo v okně programu zobrazovat.



Obrázek 5.4: 3D scéna zobrazená skriptem 5.1.

K dispozici je ovšem také rozhraní pro C++, které je vhodné pro načítání složitějších

modelů a scén. Ukázkové programy využívající toto rozhraní jsou součástí přiloženého softwaru. Tyto programy demonstrují použití ray traceru Pyrit k interaktivnímu zobrazování scén.

5.5 Experimenty

V této části srovnám některé z implementovaných algoritmů a technik z hlediska jejich reálné rychlosti (doby renderování). Testoval jsem na procesorech s jádrem řady Intel Core2.

Skript se scénou použitou při testech (obrázek 5.5) je součástí softwarového balíčku na přiloženém CD – nachází se v souboru `demos/bench.py`. Scéna obsahuje dva trojúhelníkové modely: auto (5 972 trojúhelníků) a model *bunny* (69 451 trojúhelníků, materiál částečně průsvitný i odrazivý). Renderoval se vždy obrázek o velikosti 1024×768 se zapnutým nadvzorkováním 4× (tj. celkem přes 3 mil. primárních paprsků). Hloubka rekurze byla tři úrovně.

Obrázek 5.5: 3D scéna používaná pro test rychlosti výpočtu s různými algoritmy.

Vliv použití více procesorů na rychlost

Jedním z velmi efektivních způsobů urychlení ray tracingu je použití více procesorů paralelně. Testoval jsem na počítači se čtyřmi CPU. V tabulce 5.1 můžeme vidět jak se přidáváním vláken zkracuje čas nutný k výpočtu obrazu.

Tabulka 5.1: Vztah počtu vláken (podporovaných v HW) a rychlosti renderování

počet vláken	*čas renderování*
1	38.75 s
2	19.53 s
4	10.06 s

Doba se zkracuje téměř lineárně s přibývajícím počtem procesorů (čili nezávislých vláken). Takto lze algoritmus sledování paprsku velmi snado urychlovat „hrubou silou“, stačí přidávat procesory či počítače propojené sítí.

Algoritmy dělení prostoru

Výrazný vliv na rychlost má také zvolený algoritmus dělení prostoru. Já jsem implementoval dva z nich, octree a kd-tree. V tabulce 5.2 jsou naměřené časy. Pro srovnání jsem přidal také čas bez použití urychlovací struktury (tj. vždy se kontrolují všechny objekty). Ten jsem sice nezměřil přesně, ale dle rychlosti zpracování části obrazu odhaduji, že by to takto trvalo více než 24 hodin.

Tabulka 5.2: Rychlost budování stromu a renderování s různými algoritmy dělení prostoru

algoritmus	*čas přípravy stromu*	*čas renderování*
žádný	0	více než den
octree	2.43 s	13.88 s
kd-tree	3.74 s	10.06 s

Dobře zde můžeme vidět, že octree se sice rychleji staví, ale ve výsledku je méně efektivní. Algoritmus kd-tree byl pro tuto testovací scénu značně rychlejší.

Algoritmy hledání průsečíku s trojúhelníkem

Implementoval jsem dva algoritmy pro získání průsečíku paprsku a trojúhelníku.

První je založen na barycentrických souřadnicích a k dispozici jsou jeho dvě varianty. Buď je celý výpočet proveden až při hledání průsečíku s daným trojúhelníkem, nebo jsou pevné složky předpočítány, uloženy ve struktuře trojúhelníku a při hledání průsečíku využity. Druhá varianta je sice náročnější na paměť, ale ulehčí procesoru a je tedy rychlejší.

Druhý implementovaný algoritmus využívá Plückerových souřadnic, které vyžadují ještě více předpočítaných dat, ale zcela se vyhýbají operaci dělení při hledání průsečíku. Tento algortimus by měl být vhodný pro SIMD implementaci [3].

Z testovaných algoritmů byl pouze jeden již upraven pro použití se svazky paprsků, takže bylo při tomto měření vypnuto použití SSE.

Tabulka 5.3: Rychlost renderování s různými algoritmy hledání průsečíku s trojúhelníkem

algoritmus	*čas renderování*
barycentrické souř.	28.11 s
baryc. souř. s předpoč.	27.63 s
Plückerovy souřadnice	31.10 s

V tabulce 5.3 vidíme, že nejrychlejší je podle očekávání algoritmus s předpočítanými daty pro barycentrické souřadnice. Tento algoritmus se v praxi používá nejvíce. Je-li nutné šetřit pamětí, například v případě extrémně velkého počtu trojúhelníků ve scéně, můžeme použít i variantu bez předpočítávání, která není příliš pomalejší.

Svazky paprsků s využitím SSE

Poslední test se týká urychlování s využítím sledování svazků paprsků a instrukcí SSE. V tabulce 5.4 jsou výsledky pro testovací scénu.

Tabulka 5.4: Vliv použití SSE instrukcí na rychlost renderování

SSE	*čas renderování*
povoleno	11.73 s
zakázáno	10.06 s

Tato čísla nijak neuchvátí a je třeba přemýšlet, co je špatně. Daná scéna sice způsobuje tvorbu množství sekundárních paprsků, avšak i přesto bychom čekali větší zrychlení, počítáme-li čtyři primární paprsky naráz. Podle různých studií by touto metodou měla rychlost vzrůst až o 200 procent.

Můj způsob použití SSE instrukcí zdá se není ideální a bude nutné dané algoritmy ještě dále vylepšit. Zde tedy zůstává místo pro další vývoj programu.

Kapitola 6

Závěr

V rámci diplomové práce jsem nastudoval techniky sledování paprsku a možnosti urychlení této zobrazovací metody. Navrhl jsem a vytvořil program implementující tyto techniky.

Návrh tříd ray traceru byl podrobně popsán v semestrálním projektu, v této práci jsem podle návrhu rozšířil implementaci a je zde uveden již jen celkový přehled systému a schéma tříd se stručným popisem. Podrobně jsou třídy ray traceru zdokumentovány v dodatku B a na přiloženém CD.

V dodatku A jsou informace k použití softwaru vytvořeného v rámci této práce.

Přínosem této práce je otevřená implementace multiplatformního optimalizovaného ray traceru s možností dalšího rozšiřování. Odlišností od jiných podobných programů je způsob vazby jádra ray traceru na scriptovací jazyk.

Z technik urychlení sledování paprsku jsem v této práci rozebral a v programu implementoval zejména algoritmy dělené prostoru kd-tree a octree, rychlý průsečík paprsku s trojúhelníkem pomocí barycentrických souřadnic, paralelní zpracování celého algoritmu pomocí vláken a paralelní sledování svazků čtyř paprsků pomocí instrukcí SSE.

Z hlediska dalšího vývoje programu vidím možnosti v implementaci dalších algoritmů dělení prostoru, například BIH, který poskytuje extrémně rychlý algoritmus stavby stromu, a ve vylepšení těch stávajících. Hledání průseku celého svazku paprsků s trojúhelník lze například efektivněji implementovat s využitím Plückerových souřadnic [3].

Zajímavým problémem bude způsob řešení možnosti rozšiřování rozhraní pro Python uživatelem. Je třeba to zařídit tak, aby nedocházelo k výraznému zpomalení programu. Nové objekty vytvořené v jazyce Python by bylo vhodné před použitím zkompilovat do nativního kódu. Jednou z možností je také jejich překlad zpět do jazyka C++ a objekty opět importovat do Pythonu přes vazbu. V tomto případě by překladač, pokud neporozumí nějakému příkazu, použil přímo kód v Pythonu a uživatele upozornil na problém s nízkou efektivitou.

Z nastudovaných materiálů vyplynulo, že v oblasti ray tracingu je stále mnoho příležitostí k dalšímu vývoji a zejména optimalizaci. V současnosti jsou intenzivně zkoumány možnosti použití ray tracingu pro interaktivní scény, kde je hlavním problémem dynamika objektů ve scéně, s čímž příliž nepočítají urychlovací struktury jako je kd-tree.

Výzkum se věnuje také možnostem hardwarové implementace techniky sledování paprsku.

Literatura

[1] Hans-Jochen Bartsch. *Matematické vzorce*. Mladá fronta, Praha, 2002.

[2] Dana Batali, Byron Bashforth, Chris Bernardi, Per H. Christensena, David Laur, Christophe Herya, Guido Quaroni, Erin Tomson, Thomas Jordan, and Wayne Wooten. *RenderMan, Theory and Practice*, July 2003. Siggraph 2003 course notes.

[3] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006. http://graphics.cs.uni-sb.de/~benthin/.

[4] Sean Borman. *Raytracing and the camera matrix – a connection*. A tutorial on the relationships between raytracing formulations of projective geometry and the standard camera matrix representation, June 2003.

[5] Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Local illumination environments for direct lighting acceleration. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 7–14, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[6] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[7] Warren Hunt, William R. Mark, and Gordon Stoll. Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, Sept. 2006.

[8] Intel. *Intel C++ Compiler for Linux Systems User's Guide : Intel C++ Intrinsics Reference*, 2004.

[9] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, 1990.

[10] Jeffrey Mahovsky and Brian Wyvill. Fast ray-axis aligned bounding box overlap tests with plücker coordinates. *Journal of graphics tools*, 9(1):35–46, 2004.

[11] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[12] J. Revelles, C. Ureña, and M. Lastra. *An Efficient Parametric Algorithm for Octree Traversal*. In The 8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Media.

[13] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006: 17th Eurographics Workshop on Rendering*, pages 139–150, June 2006.

[14] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[15] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, September 2006.

[16] Wikipedia. Barycentric coordinates. Dostupné na WWW: http://en.wikipedia.org/wiki/Barycentric_coordinates_(mathematics).

[17] Wikipedia. Beer-lambert law. Dostupné na WWW: http://en.wikipedia.org/wiki/Beer's_Law.

[18] Wikipedia. Snell's law – vector form. Dostupné na WWW: http://en.wikipedia.org/wiki/Snell's_Law#Vector_form.

Další odkazy

Software používaný při implementaci

GNU Compiler Collection <http://gcc.gnu.org/>
Python Programming Language <http://www.python.org/>
Python Imaging Library <http://www.pythonware.com/products/pil/>
SCons: A software construction tool <http://scons.org/>
Bazaar Version Control <http://bazaar-vcs.org/>
Valgrind <http://valgrind.org/>
MinGW – Minimalist GNU for Windows <http://www.mingw.org/>
Portable Network Graphics Reference Library <http://www.libpng.org/>
Simple DirectMedia Layer <http://www.libsdl.org/>
POSIX Threads for Win32 <http://sourceware.org/pthreads-win32/>

Modely používané v ukázkových scénách

The Stanford 3D Scanning Repository <http://www-graphics.stanford.edu/data/3Dscanrep/>
Standard Procedural Databases <http://tog.acm.org/resources/SPD/>
DMI Car 3D Models <http://dmi.chez-alice.fr/models1.html>

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Dodatek A

Informace k softwaru Pyrit

Organizace souborů

`/build`	výstupní adresář pro přeložené binární soubory a jiné pomocné soubory
`/ccdemos`	ukázkové programy v jazyce C++
`/demos`	ukázkové scény v jazyce Python
`/docs`	programová dokumentace generovaná systémem Doxygen
`/include`	hlavičkové soubory pro použití v programech
`/models`	různé modely používané ukázkovými programy
`/src`	zdrojové kódy jádra ray traceru
`/tests`	testy tříd ray traceru
`/tools`	pomocné programy

Kompilace

Příkaz `scons pyrit` spustí překlad. Seznam dalších cílů je umístěn v nápovědě, kterou lze zobrazit příkazem `scons -h`.

Softwarové požadavky:

- SCons
- pthreads (viz níže)
- libpng, zlib
- Python 2.4 nebo novější
- SDL (pro ukázkové programy v C++)

Podporovány jsou tyto překladače: GCC, IntelC, MSVC.
GCC je implicitně použit v Linuxu, MSVC ve Windows.

Stahování souborů s modely

V balíčku nejsou zahrnuty všechny modely používané ukázkovými programy. Některé příliš velké modely je potřeba stáhnout z internetu. Tyto soubory jsou vesměs stahovány ze Stanford 3D Scanning Repository. Příkaz `scons download-models` spustí stahování souborů a extrahování modelů do správných adresářů.

Stahovací script používá utility **tar** a **wget**.
Pro Windows lze tyto utility získat zde:

- http://gnuwin32.sourceforge.net/packages/wget.htm
- http://gnuwin32.sourceforge.net/packages/libarchive.htm

Pthreads

Ray tracer Pyrit ke své funkci vyžaduje knihovnu **Pthreads**. Tato knihovna je standardně k dispozici v unixových operačních systémech.
Pro Windows lze knihovnu Pthreads získat zde:

- http://sources.redhat.com/pthreads-win32/.

Ukázky v Pythonu

Tyto ukázkové programy po spuštění vyrenderují scénu a uloží výsledný obrázek do souboru pojmenovaného stejně jako skript.

boxes.py

- 512 krychlí s odrazy a lomy světla
- 4x oversampling, 2 zdroje světla, octree

bunny.py

- model bunny (70k trojúhelníků) s materiálem imitujícím sklo
- 4x oversampling, 2 zdroje světla, kd-tree

car.py

- zobrazuje model auta ve formátu LWOB
- 9x oversampling, 2 zdroje světla, kd-tree

spheres_shadow.py

- tři koule s ostrými stíny
- 4x oversampling, 2 zdroje světla, kd-tree

spheres_ao.py

- ukázka efektu ambient occlussion

spheres_glass.py

- řada skleněných kuliček s názornou ukázkou refrakce
- 4x oversampling, 3 zdroje světla, kd-tree

render_nff.py [input.nff] [output.png]

- zobrazuje scény fe formátu *nff* <http://tog.acm.org/resources/SPD/>
- čte ze standardního vstupu nebo zadaného souboru

Ukázkové programy v C++

Zobrazují scénu interaktivně, klávesami lze ovládat průlet kamery.
Ovládání je následující:

- šipky – pohled do stran a nohoru a dolů
- `w/s` – pohyb dopředu a dozadu

Demo spheres_shadow navíc podporuje:

- `r/t, f/g, v/b` – pohyb světlem (-x/+x, -y/+y, -z/+z)
- `z/x` – změna úhlu záběru (-/+)

Licence

Tento software je publikován se smluvními podmínkami definovanými licencí MIT. Plné znění licence je v souboru COPYING.

Website

Nejnovější verzi tohoto softwaru lze nalézt na stránce http://wiki.fiction.cz/Pyrit.

Dodatek B

Aplikační rozhraní pro Python

Toto je dokumentace ray traceru Pyrit k rozhraní pro Python. Jsou zde rozepsány všechny obsažené třídy spolu s konstruktory a metodami. Abstraktní třídy nelze přímo použít, slouží jen k zajištění polymorfismu. Dědické vztahy mezi třídami jsou zde rovněž vyznačeny.

V prototypech metod používám speciální typy *tuple3f* a *tuple4f*. Tyto typy znamenají, že je jako parametr očekáván *tuple* s příslušným počtem čísel typu *float*.

Raytracer

Main Raytracer class. It aggregates scene objects (via a Container) and lights. Other objects needed for rendering are Sampler and Camera, which must be also set.

Constructors:

Raytracer ()

Methods:

render ()
Render the scene. The image is created in Sampler's pixmap.

setSampler (Sampler *sampler*)
Set the sampler.

setCamera (Camera *camera*)
Set camera for the scene.

setTop (Container *top*)
Set top container for shapes.

addShape (Shape *shape*)
Add new shape to scene (via container).

addLight (Light *light*)
Add new light source to scene.
setBgColour (tuple3f *colour*)
Set background colour.

ambientOcclusion (int *samples*, float *distance*, float *angle*)
Set ambient occlusion parameters.

Sampler

Sampler abstract class. This object is used for generating samples of the screen. Colours of samples are saved to pixmap, which can be obtained from Sampler.

This cannot be created directly as it has virtual methods. See DefaultSampler for usable Sampler implementation.

Methods:

Pixmap **getPixmap** ()
Get sampler's pixmap.

Subclasses: **DefaultSampler**

DefaultSampler

DefaultSampler class, inherited from Sampler. This implements basic sampler with subsampling and oversampling.

Constructors:

DefaultSampler (float *width*, float *height*)
Creates sampler with pixmap of size width × height.

Methods:

setSubsample (int *size*)
Set subsampling mode. 0 and 1 means no subsampling, 1+ means size of sampling grid.

setOversample (int *osa*)
Set oversampling mode. 0 = off, 1 = 4x, 2 = 9x, 3 = 16x. All are regular grids.

Camera

Camera class. Implements basic ray tracing camera.

Constructors.

Camera (tuple3f p=(0,0,-1), tuple3f u=(-1,0,0), tuple3f v=(0,1,0))
The p,u,v camera constructor.

Camera (tuple3f *eye*, tuple3f *lookat*, tuple3f up=(0,1,0))
Look-at camera constructor.
This constructor is distinguished from p,u,v one by keyword arguments.

Methods:

setEye (tuple3f *eye*)
Set eye of the camera.

setAngle (float *angle*)
Set vertical angle of view (in radians).

rotate (tuple4f q)
Rotate camera with a quaternion.

Light

Point light.

Constructor:

Light (tuple3f *position*, tuple3f *colour*=(0.9, 0.9, 0.9))

Class methods:

castShadows (int *enable*)
Enable or disable shadows from this light.

Container

Container class. Basic container for shapes, no acceleration.

Constructors:

Container ()

Methods:

optimize ()
Build acceleration structure.
Does nothing for pure Container, but it is overriden by subclasses.

Subclasses: **Octree**, **KdTree**

Octree

Octree class, inherited from Container. Implements octree acceleration structure.

Constructors:

Octree ()

KdTree

KdTree class, inherited from Container. Implements kd-tree acceleration structure.

Constructors:

KdTree ()

Shape

Shape abstract class. It provides base attributes for all shapes. Cannot be used directly.

Subclasses: **Sphere**, **Box**, **Triangle**

Sphere

Sphere shape class.

Constructors:

Sphere (tuple3f *center*, float *radius*, Material *material*)

Box

Box shape class.

Constructors:

Box (tuple3f L, tuple3f H, Material *material*)
L and H are corners of the box.
L should contain lower bounds on all three axes and H the higher bounds.

Triangle

Triangle shape class.

Constructors:

Triangle (Vertex A, Vertex B, Vertex C, Material *material*)

Methods:

tuple3f **getNormal** ()
Get normal of whole triangle.

Vertex

Vertex class. Defines a vertex of the Triangle.

Constructors:

Vertex (tuple3f *vector*)

Subclasses: **NormalVertex**

NormalVertex

NormalVertex class. Defines a vertex of Triangle and normal in this vertex.

Constructors:

NormalVertex (tuple3f *vector*, tuple3f *normal* = (0,0,0))

Methods:

setNormal (tuple3f *normal*)
Set normal of this vertex.

Material

Material class. Used for defining shape surfaces.

Constructors:

Material (tuple3f *colour*=(1,1,1))

Methods:

setPhong (float *ambient*, float *diffuse*, float *specular*, float *shininess* = 0.5)
Set ambient, diffuse, specular and shininess Phong model constants.

setReflectivity (float *reflect*)
Set reflectivity.

setTransmissivity (float *transmiss*, float *rindex* = 1.3)
Set transmissivity and refraction index.

setSmooth (int *enable* = 1)
Set triangle smoothing.

setTexture (int *enable* = 1)
Set the texture.

Texture

Abstract Texture class. Cannot be created directly. Used via Material to define texture of a Shape.

Subclasses: **ImageTexture**, **CheckersTexture**, **CloudTexture**

ImageTexture

ImageTexture class. Use a raster image as the texture.

Constructors:

ImageTexture (TextureMap *tmap*, Pixmap *image*)

CheckersTexture

CheckersTexture class. Classic checkers texture. Colours must be defined via ColourMap, three values of the band are used – 0, 0.5, 1. Can be mapped on any colours.

Constructors:

CheckersTexture (TextureMap *tmap*, ColourMap *cmap*)

CloudTexture

CloudTexture class. Use Perlin cloud as a 3D texture. Can be used for clouds or stone textures.

Constructors:

CloudTexture (float *detail*, ColourMap *cmap*)
The detail specifies how many iterations of the noise should be added to texture.

TextureMap

Abstract TextureMap class. Cannot be created directly. Defines mapping of 2D texture to 3D space.

Subclasses: **PlanarMap**, **CubicMap**, **CylinderMap**, **SphereMap**

PlanarMap

PlanarMap class. Planar texture mapping.

Constructors:

PlanarMap (tuple3f *center*, float *size*)
Center is point in world space where texture's point (0.5, 0,5) is mapped.
Size means how large should the unit texture tile be in world units.

CubicMap

CubicMap class. Cubic texture mapping.

Constructors:

CubicMap (tuple3f *center*, float *size*)
See PlanarTexture constructor for description.

CylinderMap

CylinderMap class. Cylindrical texture mapping.

Constructors:

CylinderMap (tuple3f *center*, float *size*)
See PlanarTexture constructor for description.

SphereMap

SphereMap class. Spherical texture mapping.

Constructors:

SphereMap (tuple3f *center*, float *size*)
See PlanarTexture constructor for description.

ColourMap

Abstract Colourmap class. Cannot be created directly. Defines mapping of values <0..1> to colours.

Subclasses: **LinearColourMap**, **BoundColourMap**

LinearColourMap

LinearColourMap class.

Constructors:

LinearColourMap (tuple3f *clow*, tuple3f *chigh*)
This colour map interpolates colours between clow and chigh for whole range 0 to 1.

BoundColourMap

BoundColourMap class.

Constructors:

BoundColourMap (tuple *bounds*, tuple *colours*)
First argument is tuple of bounds (float), second argument tuple of colours (tuple3f).
Both must have same size, otherwise smaller size is chosen.
Bounds must contain numbers between 0 and 1 in rising order.
Last number should be larger than one.
Colours are mapped to slices between bounds, with zero being implicit bottom bound.

Pixmap

Pixmap class. Contains raster data.

Constructors:

Pixmap (float *width*, float *height*)
Reserves memory for pixmap of size width × height.

Methods:

float **getWidth** ()
Get width of pixmap.

float **getHeight** ()
Get height of pixmap.

string **getCharData** ()
Get raw byte data.

int **writePNG** (string *fname*)
Write pixmap to PNG file.