VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Interaktivní sledování paprsku

Bakalářská práce

2007 Rostislav Jadavan

Interaktivní sledování paprsku

Vedoucí :

Herout Adam, Ing., Ph.D., UPGM FIT VUT

Řešitel :

Rostislav Jadavan

Kategorie :

Počítačová grafika

Zadání :

1. Prostudujte a popište algoritmus sledování paprsku pro zobrazování scén.
2. Prostudujte a popište metody implementace sledování paprsku pro zobrazování v reálném čase.
3. Vyberte a implementujte vhodné metody pro urychlování výpočtu sledování paprsku.
4. Vytvořte demonstrační a testovací ray-tracer (program zobrazující metodou sledování paprsku); demonstrujte jeho funkčnost na několika jednoduchých scénách.
5. Zhodnoťte dosažené výsledky a navrhněte možnost pokračování projektu; vytvořte plakátek pro prezentování projektu.

Část požadovaná pro obhajobu semestrálního projektu :

Body 1.-2., experimenty vedoucí k řešení bodu 3.

Interaktivní sledování paprsku

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Adama Herouta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Rostislav Jadavan
22.1.2007

Abstrakt

Tato práce se zabývá problematikou zobrazování technikou sledování paprsku v reálném čase. Po úvodu, který obsahuje stručný základ této problematiky, následuje rozebrání technik pro urychlování výpočtu. Většina rozebíraných technik je aplikovatelná na dnes běžně používané počítače. Poslední část se zaměřuje na vlastním implementaci vybraných technik a zhodnocení výsledků měření.

Klíčová slova

Sledování paprsku, optimalizace, dělení scény, adaptivní dělení mřížky, SIMD instrukce, grafický akcelerátor

Poděkování

Chtěl bych poděkovat Ing. Adamu Heroutovi za kvalitní vedení bakalářské práce a rady při studiu zadané problematiky.

Abstract

This work deals with image rendering using raytracing in realtime. After short introduction to raytracing, different technics to compute raytracing in realtime is described. Most of them can be aplied to current computers. Last part shows implementation and test results.

Keywords

Raytracing, optimalization, space partitioning, adaptive subsampling, SIMD instructions, graphic procesor

Obsah

Obsah 6
1 Úvod 8
1.1 Organizace dokumentu 8
2 Sledování paprsku 9
2.1 Princip zobrazování 9
2.2 Řešení průsečíku paprsku s objekty 9
2.2.1 Řešení průsečíku paprsku s koulí 9
2.3 Osvětlení 11
2.3.1 Osvětlovací model 11
2.3.2 Stíny 12
2.3.3 Lom světla 12
2.4 Odraz 13
2.5 Textury 13
2.5.1 Procedurální textury 13
2.6 CSG 13
2.7 Porovnání s rasterizací 14
3 Optimalizace 15
3.1 Optimalizace neovlivňující kvalitu 15
3.1.1 Preprocesing (předpočítávání) 15
3.1.2 Optimální řešení průsečíku paprsku 15
3.1.3 Dělení scény 17
3.1.4 Obalující objem 20
3.1.5 Optimalizace pro stínové paprsky 20
3.1.6 Využití SIMD instrukcí 21
3.1.7 Využití více procesorů nebo vícejádrový procesor 23
3.2 Optimalizace ovlivňující kvalitu 23
3.2.1 Adaptivní subsampling 23
3.2.2 Využití grafického akcelerátoru 25
3.3 Optimalizační techniky pro programování 29
3.3.1 Paměťová náročnost 29
4 Vlastní implementace 30
4.1 Architektura programu 30
4.2 Vykreslovací jádro 31
4.3 Seznam vlastností programu 32

4.4 Měření 33
4.4.1 Testovací scény 33
4.4.2 Výsledky měření 35
4.4.3 Shrnutí 38
5 Závěr 40
Literatura 41
Seznam obrázků 42
Seznam grafů 42
Seznam tabulek 42

1 Úvod

Cílem počítačové grafiky je dosáhnout kvality vykreslování nerozeznatelné od fotografie či filmu. Sledování paprsku je zobrazovací metoda, pomocí které můžeme dosáhnout vysoké zobrazovací kvality a fyzikálně přesného obrazu. Oproti dnes používanému přístupu, tj. rasterizaci, je však výpočetně mnohem náročnější. Rasterizace však stále častěji naráží na limity, které si vynucují složité metody řešení. Sledování paprsku je přitom mnohem přirozenější způsob zobrazování.

S nárůstem výkonu osobních počítačů se stává zobrazování v reálném čase možným. Účelem této práce je prozkoumat vybrané techniky pro urychlení výpočtu a zhodnotit jejich vliv na rychlost a kvalitu výstupu.

1.1 Organizace dokumentu

Kapitola 2 popisuje základní princip algoritmu sledování paprsku. Je zde obsažen teoretický popis všech základních vlastností spolu se vzorci nutnými pro výpočet. Další kapitola rozebírá techniky pro urychlování výpočtu, které jsou rozděleny na metody ovlivňující a neovlivňující výslednou kvalitu výstupu. Každá metoda je podrobně popsána a na konci je shrnutí, které obsahuje hodnocení z hlediska využitelnosti pro urychlení výpočtu. Poslední kapitola obsahuje vlastní implementaci vybraných technik z kapitoly 3 a výsledky měření. Je zde také popis architektury programu se seznamem implementovaných vlastností.

2 Sledování paprsku

2.1 Princip zobrazování

Princip algoritmu sledování paprsku spočívá ve vrhání paprsků z místa pozorovatele a řešení viditelnosti pro tyto paprsky. Vlastně hledáme pro každý paprsek (polopřímku) první objekt, který protíná.

Postupně vrháme paprsky pro každý bod obrazu a řešíme viditelnost pro daný paprsek. Řešení viditelnosti znamená, že počítáme průsečíky pro všechny objekty a hledáme ten nejbližší. Po nalezení nejbližšího průsečíku vyhodnotíme barvu daného bodu a bod vykreslíme. Tím nám vznikne obraz celé scény z místa pozorovatele.

2.2 Řešení průsečíku paprsku s objekty

Hledáme bod v prostoru, ve kterém vržený paprsek zasáhl daný objekt, pokud vůbec daný bod existuje. Bod průsečíku potřebujeme znát pro další výpočty, tj. osvětlení, odraz a lom světla.

Pokud lze popsat objekt jednou rovnicí (koule, elipsoid, plocha), řešení spočívá v dosazení rovnice paprsku do rovnice objektu. Například u trojúhelníku musíme řešit nejprve průsečík s plochou a pokud existuje, pak na dané ploše zjistit jestli bod leží uvnitř trojúhelníku.

2.2.1 Řešení průsečíku paprsku s koulí

Budeme počítat průsečík paprsku s koulí, takže vlastně budeme hledat bod na kouli, který protíná polopřímka. Body můžou být dva, jeden nebo žádný, pokud jde paprsek mimo kouli.

Paprsek je definován jak polopřímka :

$$P(t) = P_0 + t \cdot P_s \ , \ t > 0$$

kde :

$P_0 = [x_0, y_0, z_0]$ je bod počátku, tj.místo pozorovatele

$P_s = [x_s, y_s, z_s]$ je směrový vektor paprsku

Koule je definována pozicí středu $[x_c, y_c, z_c]$ a poloměrem r :

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$$

Dosadíme rovnici paprsku do rovnice koule :

$$(x_0 + x_s \cdot t - x_c)^2 + (y_0 + y_s \cdot t - y_c)^2 + (z_0 + z_s \cdot t - z_c)^2 = r^2$$

Po úpravě dostaneme :

$$A \cdot t^2 + B \cdot t + C = 0$$

kde :

$$A = x_s^2 + y_s^2 + z_s^2$$

$$B = 2 \cdot (x_s \cdot (x_0 - x_c) + y_s \cdot (y_0 - y_c) + z_s \cdot (z_0 - z_c))$$

$$C = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2$$

Pokud je směrový vektor paprsku normalizovaný, tj. $|P_s| = 1$, potom $A = 1$. Řešení kvadratické rovnice je tedy :

$$t_{0,1} = \frac{(-B \pm \sqrt{B^2 - 4 \cdot C})}{2}$$

Výsledný bod potom získáme :

$$V = [x_0 + x_s \cdot t_i, y_0 + y_s \cdot t_i, z_0 + z_s \cdot t_i]$$

Podle diskriminantu poznáme, zda paprsek kouli zasáhl nebo minul. Jestliže je diskriminant rovnice menší než 0, znamená to, že neexistuje reálný kořen rovnice, tj. neexistuje průsečík, paprsek neprotíná kouli. Pokud je diskriminant roven 0, paprsek se koule dotýká v jednom bodě, kde řešení rovnice je právě tento jeden bod. Pokud má rovnice dvě řešení, tj. diskriminant je větší než 0, paprsek kouli protíná ve dvou bodech. Bližší z nich vybereme podle velikosti t_0 a t_1, kde menší z nich bude znamenat bližší bod.

2.3 Osvětlení

2.3.1 Osvětlovací model

Fyzikálně přesný model osvětlený je příliš složitý a navíc nevhodný pro sledovaní paprsku, proto využijeme aproximaci tohoto modelu. Pro výpočet osvětlení je běžně používaný Phongův model osvětlení, který má 3 složky :

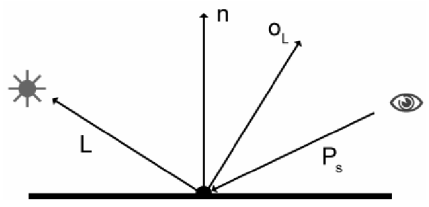
- difúzní světlo - barva materiálu objektu
- spekulární odraz - pro simulaci odrazivých objektů, např. kov nebo plast
- ambientní světlo - okolní osvětlení

Phongův model osvětlení tedy získáme jako součet všech tří složek pro každé světlo :

$$I = \sum_{L} I_L \cdot b_d \cdot (\vec{n} \cdot \vec{L}) + I_L \cdot b_s \cdot (\overrightarrow{o_L} \cdot \overrightarrow{P_S}) + b_a$$

kde :

I_L ... barva světla
b_d, b_s ... barva difúzní a spekulární složky
b_a... ambientí světlo
n ... normála objektu
o_L ... vektor odraženého paprsku světla
L, P_s ... vektor světla a směrový vektor paprsku



Obrázek 2.1: Phongův osvětlovací model

2.3.2 Stíny

Stín v bodě vznikne, jestliže mezi místem zásahu paprsku a světlem leží nějaký objekt. Budeme tedy zjišťovat jestli existuje průsečík mezi světelným paprskem a některým objektem. Světelný paprsek vzniká v bodě světla a končí v místě zásahu objektu. Pokud průsečík neexistuje, na daný bod aplikujeme Phongův osvětlovací model, jinak bude mít bod barvu ambietního světla, tj. světla okolí.

2.3.3 Lom světla

Využijeme fyzikálního zákona o lomu světla, pomocí kterého vypočítáme směr nově vyslaného paprsku.

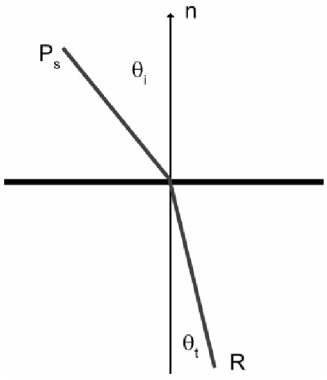
$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_t}{\eta_i} = \eta$$

Tento vztah je však nepraktický. Potřebujeme znát výsledný vektor paprsku, který vzniká v bodě lomu. Po úpravě získáme vektor pro nově vzniklý paprsek :

$$R = \eta \cdot I + \left(\eta \cdot c - \sqrt{1+\eta^2 \cdot (c^2 - 1)}\right) \cdot n$$

kde :

$$c = -n \cdot I \qquad \eta = \frac{\eta_t}{\eta_i}$$



Obrázek 2.2: Lom světla

2.4 Odraz

Odraz slouží k simulaci povrchů jako je například zrcadlo. Z místa zásahu paprsku vyšleme odražený paprsek podle rovnice :

$$O = P_S + 2 \cdot n \cdot (-P_S \cdot n)$$

kde :

P_s ... směrový vektor paprsku
n ... normálový vektor objektu

2.5 Textury

Texturou se rozumí materiál, který je nanesen na povrch objektu. Obvykle jde o dvourozměrnou bitmapu. Nevýhodou bitmapy je, že kvalita je závislá na jejím rozlišení. Při velkém zvětšení dojde k degradaci kvality a jsou viditelné jednotlivé pixely. Obvyklé řešení spočívá v použití filtrování, kde při zvětšení textury dojde k vyhlazení. Jednotlivé pixely již nejsou tolik viditelné, ale tvary s ostrými hranami mají rozmazané okraje.

2.5.1 Procedurální textury

Procedurální textura je zadána rovnicí (nebo více rovnicemi), která reprezentuje popis textury. Při zvětšování nedochází ke zmenšení kvalitu výstupu, protože dojde k přesnému výpočtu barvy bodu pro konkrétní místo na povrchu objektu.

2.6 CSG

CSG je zkratka pro Constructive Solid Geometry, což znamená vytváření objektů ze základních primitiv, např. koule, plocha, válec apod. Jednotlivá primitiva můžeme kombinovat pomocí logických operací, například sloučení, rozdíl nebo průnik. Výsledný bod vzniká na základě výsledku logické funkce, která je vyhodnocena podle testu průsečíku paprsku s objekty.

2.7 Porovnání s rasterizací

Při rasterizaci dochází k vykreslování trojúhelníků, kde viditelnost je řešena pomocí paměti hloubky (*z-buffer*). U sledování paprsku řešíme, který objekt je viditelný pro daný paprsek. Zásadní rozdíl je tedy v tom, že při rasterizaci kreslíme navzájem nezávislé trojúhelníky, kdežto při sledování paprsku operujeme s celou scénou. To nám umožňuje lépe vykreslovat globální efekty, např. stíny a odrazy. U rasterizace jsme nuceni použít pro vykreslení odrazu bitmapu, na které bude vykreslen odraz. Při vícenásobném odrazu již toto řešení nelze použít.

Obecně lze tedy říct, že u rasterizace jsme nuceni používat pro reálné zobrazení materiálu určité speciální řešení, které navíc vždy není univerzální. Naopak při sledování paprsku nám stačí nadefinovat chovaní paprsku pro různé druhy materiálů a ve výstupu dojde ke korektnímu zobrazení i ve vícenásobné kombinaci různých vlastností. Navíc můžeme aplikovat fyzikální zákony a dosáhneme tak realistického zobrazení materiálu.

3 Optimalizace

Sledování paprsku je oproti rasterizace výrazně výpočetně náročné. Jednoduchá implementace algoritmu nedokáže dosáhnout takový počet snímků za sekundu, který by byl interaktivní.

Výsledky měření ukázaly, jak jsou různé části algoritmu výpočetně náročné.

Testovací scéna (9 objektů a 3 světla)	
primární paprsky	9,395·10-6s
sekundarní paprsky - osvětlení	25,4 ·10-6s
sekundární paprsky - odraz	0,475 ·10-6s
textura	4,512 ·10-6s

Tabulka 3.1: Výsledky měření ze semestrální práce

Nejprve se tedy budeme snažit zrychlit výpočet průsečíků, tzn. zmenšit počet výpočtů nutných pro nalezení průsečíku. To povede k urychlení výpočtu primárních a sekundárních paprsků. Další možností je snížit počet vržených paprsků nutných k zobrazení.

3.1 Optimalizace neovlivňující kvalitu

3.1.1 Preprocesing (předpočítávání)

Pokud některé data zůstávají v průběhu vykreslování stejná, je vhodné je uložit a přepočítávat jen v případě změny. To se týká například statických objektů, tj. objekty které v čase nemění svoji pozici. Vezměme pro příklad kouli. Při výpočtu průsečíku primárního paprsku lze část rovnice předpočítat, protože poloměr a pozice zůstávají stejné, mění se pouze směr paprsku. Podrobněji je řešení průsečíků probráno v další části.

3.1.2 Optimální řešení průsečíku paprsku

Množství počítaných průsečíků s objektem tvoří největší podíl při výpočtu sledovaní paprsku. Proto je nutné jejich výpočet co nejvíce urychlit. Pro každý typ objektu ve scéně najdeme optimální metodu řešení průsečíku. Například průsečík s koulí můžeme řešit použitím obecné rovnice pro kuželosečky,

ale výpočet bude pomalý. Algoritmus, který bude počítat pouze průsečík s koulí, bude velmi jednoduchý a rychlý.

Paprsky lze rozdělit do dvou skupin, na primární paprsky, tj. paprsky vycházející z počátku (z místa pozorovatele) a sekundární paprsky. Sekundární paprsky mohou mít svůj počátek různý, např. stínové paprsky nebo paprsky odrazivých objektů (zrcadlo). Pro každý druh paprsku můžeme ještě nalézt další optimalizace.

3.1.2.1 Primární paprsky

Primární paprsky mají svůj počátek v místě pozorovatele. Pokud si souřadný systém, pro vykreslování, nastavíme tak, aby místo pozorovatele bylo v počátku, tj. [0, 0, 0], zjednoduší se nám rovnice pro průsečík.

3.1.2.2 Sekundární paprsky - stínové

Stínový paprsek je vrhán z místa zásahu primárního paprsku směrem ke světlu. Zde můžeme přepočítat pozici všech objektů vzhledem k pozici světla tak, že budeme vlastně počítat primární paprsek od světla směrem k objektu. Opět dojde ke zjednodušení rovnic a tím pádem k urychlení. Vyplatí se to ale pouze pro více světel. Při užití jednoho světla zatěžuje algoritmus přepočítávání objektů natolik, že optimalizace je nevhodná.

Pro stínové paprsky taky nepotřebujeme znát místo zásahu, stačí nám indikace, zda byl objekt zasažen, či ne, tzn. zjišťujeme zda je objekt mezi světlem a místem zásahu nebo ne.

3.1.2.3 Příklad optimalizace primárních paprsků pro kouli

Rovnice pro řešení průsečíku paprsku a koule vypadá následovně :

$$A \cdot t^2 + B \cdot t + C = 0$$

kde :

$$A = x_s^2 + y_s^2 + z_s^2$$

$$B = 2 \cdot (x_s \cdot (x_0 - x_c) + y_s \cdot (y_0 - y_c) + z_s \cdot (z_0 - z_c))$$

$$C = (x_0 - x_c)^2 + (y_0 - y_c)^2 + (z_0 - z_c)^2 - r^2$$

Vidíme, že složku C můžeme kompletně předpočítat, protože není závislá na směru paprsku, pouze na pozici pozorovatele, která je však konstantní. Pokud zvolíme pozici pozorovatele [0, 0, 0], vznikne výsledný tvar rovnice pro složku *C* :

$$C = x_0^2 + y_0^2 + z_0^2 - r^2$$

Podobně upravíme *B*, *A* vyřadíme úplně, protože pokud je směrový vektor paprsku normalizovaný, tj. $|P_s| = 1$, pak $A = 1$. Pseudokód pro výpočet průsečíku s koulí po optimalizaci pak bude vypadat :

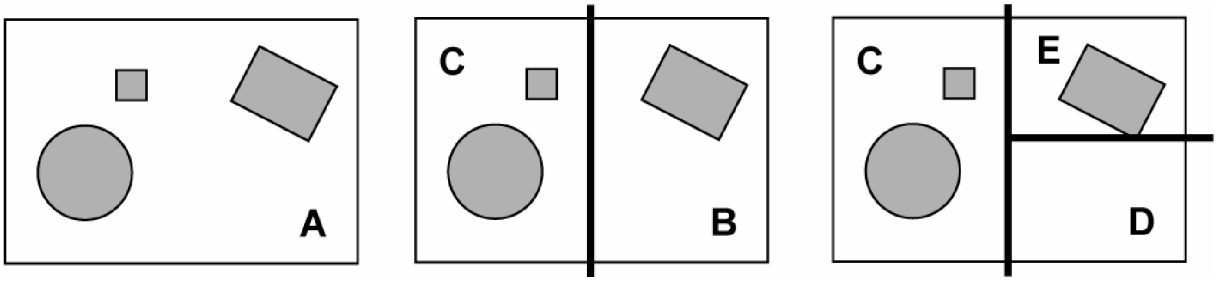
```
function preCalc {
      C = dot_product(pos, pos) - r*r
}

function intersectPrimary {
      B = dot_product(ray.d, pos)
      D = B*B - C

      if (D < 0) NO_HIT

      return t = -B - square_root(D)
}
```

3.1.3 Dělení scény

Při velkém počtu objektů ve scéně je potřeba snížit množství průsečíků s objekty. Dosáhneme toho, rozdělením scény na části a testovat potom budeme pouze objekty z části kterou prochází paprsek.

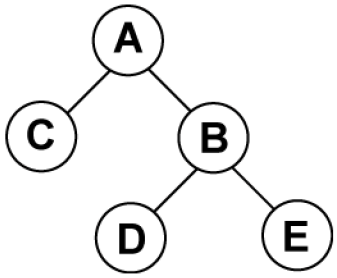
3.1.3.1 BSP

Algoritmus využívá k dělení prostoru plochu, která rozdělí scénu na dvě poloviny. Plocha může být libovolně orientována. Rekurzivně tedy dělíme každou novou polovinu dále, dokud jsme nedosáhli požadované hloubky. Do každého uzlu uložíme dělící plochu.



Obrázek 3.1: Dělení scény pomocí BSP stromu

Při dělení se vytváří strom, kde každý koncový uzel obsahuje seznam objektů které se v daném prostoru nacházejí.



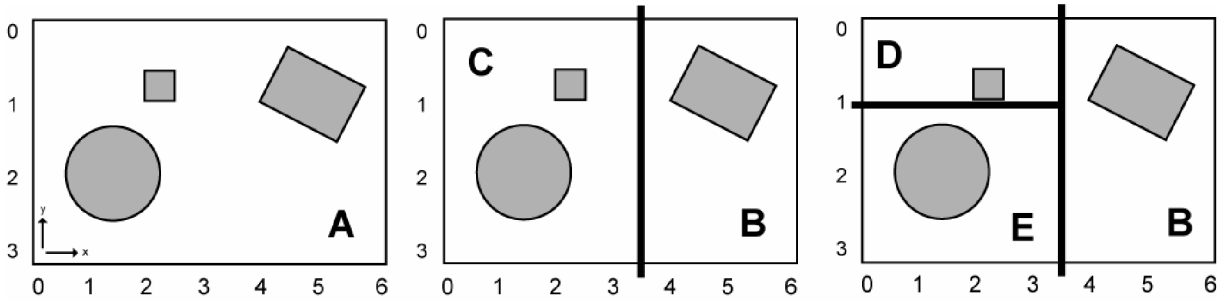
Obrázek 3.2: BSP strom

Pro dělení je potřeba navrhnout optimální algoritmus, který každou novou část opět vhodně rozdělí, tj. najde optimální pozici a orientaci dělící plochy. Jako podmínku pro ukončení dělení můžeme použít například počet objektů v nově vzniklých dvou uzlech. Pokud se bude počet velmi lišit a prostor bude rovnoměrně rozdělen, víme, že je třeba dále dělit pouze část s větším počtem objektů. Úplně ukončíme dělení například pokud se dalším dělením nezmenší počet objektů, tj. v prostoru je pouze jeden objekt.

Dělení touto technikou je velmi efektivní a hojně využívané při rasterizaci. Hlavní nevýhodou je složitější test průsečíku s dělící plochou, při procházení stromu, protože plocha není osově orientována. Není také vhodný pro animované scény, protože vytvoření stromu je časově příliš náročné.

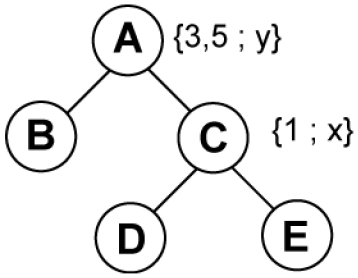
3.1.3.2 KD-Tree

KD strom využívá k dělení také plochu, ale plocha musí být rovnoběžná s některým vektorem souřadného systému.



Obrázek 3.3: Dělení scény pomocí KD stromu

Při dělení se vytváří strom, kde každý uzel obsahuje informace o dělící ploše, tj. dvojici souřadnice plochy a osa souřadného systému.



Obrázek 3.4: KD strom

Pro vhodné dělení je opět nutné najít optimální algoritmu, což v tomto případě není jednoduché. Základní metodou je rozdělit obě poloviny tak, aby na každé půlce byl stejný počet objektů. Vlastně je potřeba najít funkci, která ohodnotí nově vzniklý prostor. Takto dosáhneme vyváženého stromu a dojde k optimálnímu urychlení.

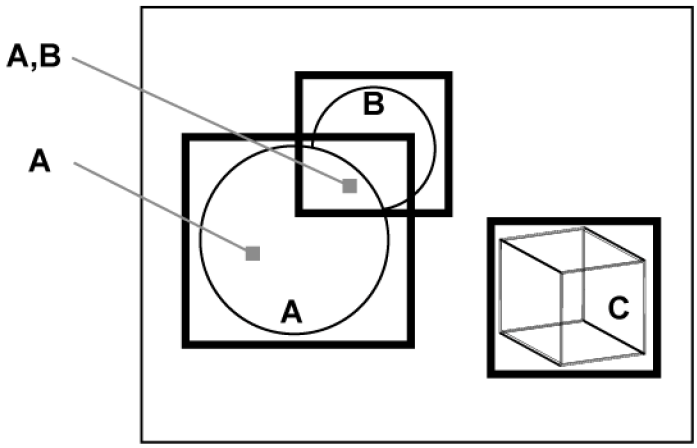
Výhodou je, že touto technikou dokážeme dobře rozdělit prostor s nerovnoměrným rozdělením objektů. Prostor, který vznikne dělením, je osově orientován, tudíž procházení je velmi snadné. Nevýhodou je menší flexibilita při dělení osově orientovanou plochou.

3.1.3.3 Shrnutí

Pro rozsáhlé scény s velkým množstvím objektů je použití techniky dělení scény nutnost, protože množství počítaných průsečíků by byl jinak neúnosný. V práci [1] se ukazuje, že pro sledování paprsku je nejvýhodnější použití KD stromu. Velkou výhodou je rychlost procházení, protože dělící rovina je osově orientována. To nám umožňuje také rychlou konstrukci stromu, protože podle znaménka poznáme, zda se objekt nachází před nebo za dělící rovinou.

3.1.4 Obalující objem

Jedná se vlastně o obálku (*bounding volume*), která je přetransformována do obrazového prostoru a umožní nám zjistit, který objekt se vyskytuje pro daný pixel obrazu. To nám umožní nalézt konkrétní objekt a není nutné procházet všechny objekty ve scéně.



Obrázek 3.5: Obalující objem

3.1.5 Optimalizace pro stínové paprsky

Stínové paprsky tvoří velké množství ze všech vrhaných paprsků. Pro každý průsečík, vzniklý od primárního paprsku, je nutné vrhnout paprsek do každého světla a pro každý z nich zjistit, zda-li mezi nimi neleží nějaký objekt. Proto se snažíme množství počítaných průsečíků zmenšit.

3.1.5.1 Paměť stínu

Paměť stínu (*shadow-cache*) se skládá z těchto částí :

- každé světlo si bude pamatovat který objekt byl naposledy zasažen světelným paprskem
- při testování objektů, zda-li je mezi průsečíkem a světlem, ukončíme testování po nalezení prvního objektu

Poslední zasažený objekt ukládáme proto, že pokud vrháme paprsky postupně jak jdou po sobě, je velká pravděpodobnost, že pokud bude daný bod ve stínu, vedlejší bude určitě taky. Algoritmus se ovšem často dopustí chyby (třeba u složitější scény, s velkým množstvím objektů blízko sebe), ale zatížení je natolik nepatrné, že ve výsledku je výhodné tuto techniku použít.

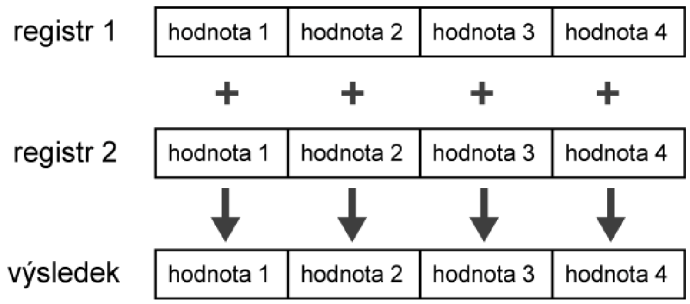
Ukončit testování po nalezení prvního objektu můžeme proto, že nalezení dalšího objektu by stejně nemělo smysl. Zajímá nás pouze jestli je daný bod ve stínu či ne. Pokud je první testovaný objekt právě ten, který byl uložen jako poslední počítaný, dojde k velmi rychlému vyhodnocení stínu.

3.1.5.2 Objem stínu

Abychom ještě více omezili množství počítaných průsečíků se stínovými paprsky, vytvoříme takový obalující objem, který bude mít počátek v místě světla a konec bude pokrývat daný objekt. Potom otestujeme, jestli ostatní objekty neleží uvnitř tohoto objemu. Pokud ano, přidáme je do seznamu pro daný objekt. Potom při počítání stínového paprsku omezíme testované objekty pouze na tento seznam. Tím dosáhneme toho, že budeme počítat průsečíky s objekty, které opravdu leží mezi světlem a daným objektem.

3.1.6 Využití SIMD instrukcí

Moderní procesory obsahují jednotky pro paralelní výpočet dat. Pomocí instrukcí typu SIMD (např. Intel SSE, AMD 3DNow! nebo IBM/Motorola AltiVec) lze jednou instrukcí zpracovávat více hodnot v plovoucí řádové čárce. To nám nabízí k optimalizaci všechny výpočty, které lze vykonávat paralelně. A protože sledování paprsku jich obsahuje dostatečné množství, jsou SIMD instrukce výhodným nástrojem pro optimalizaci.



Obrázek 3.6: Součet 4 hodnot pomocí SIMD instrukce

3.1.6.1 Vektory

Vektorové operace jsou ideální pro výpočet za pomocí SSE. Můžeme sečíst dva vektory pomocí jedné instrukce. Ve výsledku dojde asi k 30% zrychlení, po implementaci do programu je zrychlení už jen v řádech jednotek procent.

3.1.6.2 Urychlení výpočtu průsečíku

Protože rychlost vykreslování závisí na počtu počítaných paprsků, je vhodné zvýšit rychlost jejich výpočtu. Nabízí se dvě možnosti :

- vrhat jeden paprsek a počítat průsečík se čtyřmi objekty
- pro jeden objekt počítat průsečíky se čtyřmi paprsky

1 paprsek, 4 objekty

Budeme tedy testovat průniky čtyř objektů s jedním paprskem. Nevýhodou je, že testované objekty musí být stejného typu. Pokud vyžadujeme různé objekty, tudíž různé algoritmy pro řešení průsečíku, bylo by nutné shlukovat objekty podle stejného typu. Algoritmus shlukování zvýší složitost a toto řešení bude neefektivní.

Toto řešení lze použít pro jeden obecný typ, kterým lze popsat celou scénu, například trojúhelníky. Nevýhodou ale zůstává větší zátěž na propustnost paměti, protože informace o objektu jsou vždy objemově větší než popis paprsku, což znamená větší objem přenesených dat.

4 paprsky, 1 objekt

Možnost testovat 4 paprsky najednou je daleko vhodnější. Je zde pouze jeden algoritmus pro výpočet průsečíku, který lze lehce převést na paralelní. Pro plné využití SIMD jednotek je nutné data vhodně uspořádat. Například pro paprsek vytvoříme pole, které bude obsahovat 4 hodnoty. Každá bude reprezentovat jednu složku vektoru každého paprsku.

```
CtyriPaprsky
{
      float origin_x[4], origin_y[4], origin_z[4]
      float dir_x[4], dir_y[4], dir_z[4]
}
```

Podobně vytvoříme datovou strukturu pro objekt. Tímto uspořádáním bude moci použít kteroukoliv SSE instrukci na paralelní výpočet průsečíku.

3.1.7 Využití více procesorů nebo vícejádrový procesor

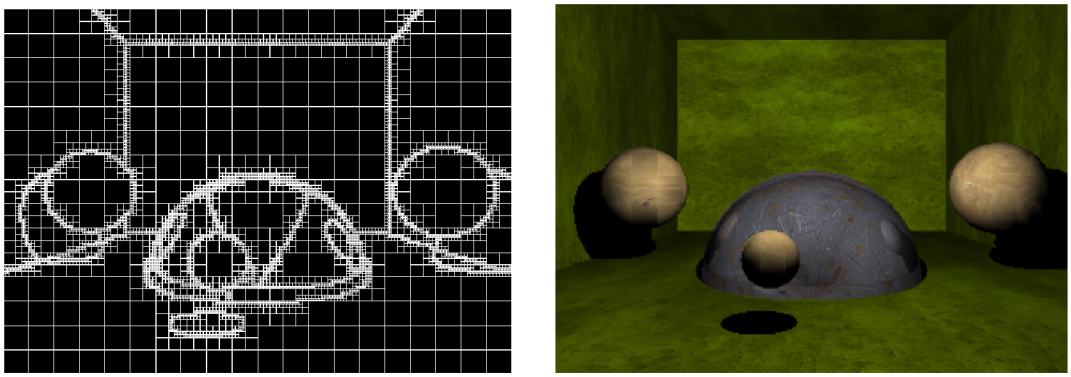
Na příkladu využití SIMD instrukcí je jasné, že výpočet sledování paprsku lze vhodně převést na paralelní. Složitost algoritmu sledování paprsku je téměř lineární, takže rychlost na dvou procesorech vzroste téměř dvakrát oproti procesoru jednomu (zdroj [2] a [4]).

3.2 Optimalizace ovlivňující kvalitu

Doteď jsme se snažily hlavně zrychlit výpočet paprsků. Nyní se budeme snažit zmenšit počet paprsků nutných k zobrazení. Dojde tím ke zmenšení kvality výstupu, ale zisk rychlosti by měl být značný.

3.2.1 Adaptivní subsampling

Tato technika spočívá v rozdělení obrazu na pravidelnou mřížku. Nyní budeme zjišťovat jaké objekty leží uvnitř každého čtverce mřížky. Uděláme to tak, že z každého rohu čtverce vyšleme paprsek. Pro každý si uložíme informace o průniku, tj. identifikátor objektu (ID), který paprsek zasáhl, identifikátor světelného zdroje, výslednou barvu pixelu a koordináty pro textury. Potom procházíme mřížku po čtvercích a podle shody identifikátorů (ID) dělíme mřížku dál nebo daný čtverec vykreslíme.



Obrázek 3.7: Adaptivní subsampling

ID objektů jsou stejná

Nyní víme, že uvnitř čtverce leží jeden objekt. Můžeme ho vykreslit postupným vrháním paprsků pro každý bod čtverce mřížky, kde budeme hledat průsečík pouze s tímto objektem a dosáhneme tak zrychlení výpočtu. Objekt bude přesně vykreslen a nezmenšíme tak kvalitu výstupu.

Rychlejší možností je na základě barvy interpolovat čtverec mřížky např. pomocí Gouraudova stínování, které vytvoří plynulý přechod mezi rohy čtverce. Tady už dojde ke zmenšení kvality výstupu, ale zisk rychlosti bude velký. Hrany objektů budou nepřesné ("kostky") a kvalitu bude ovlivňovat hloubka dělení mřížky, tj. velikost nejmenšího čtverce při posledním dělení.

ID objektů jsou různá

V tomto případě dělíme mřížku dál, ale pouze pokud jsme nedosáhli požadované hloubky dělení[1]. Mřížku dělíme tak, že vrhneme paprsky ze středu každé hrany čtverce a ze středu čtverce. Získáme tak čtyři nové čtverce, které buď dále dělíme nebo vykreslujeme.

Problém nastane pokud jsou ID různá a my jsme dosáhli požadované hloubky dělení mřížky. Jedna z možností je opět vrhat paprsky pro každý bod čtverce. Podobně můžeme omezit testování průsečíku pouze na objekty, které se v daném čtverci vyskytují. Celkově tato možnost vychází nevýhodněji oproti možnosti další.

Pokud je čtverec dostatečně malý, můžeme použít uložené informace z jednoho krajního bodu, ostatní tři podle něj dopočítat a vykreslit celý čtverec. Kvalita výstupu je horší. Například klasickým projevem je u koule hranatost okrajů, ale zisk rychlosti při výpočtu je značný, protože nemusíme provádět výpočet průsečíku s objektem.

3.2.1.1 Světla a stíny

Při použití více světel ve scéně je nutné při dělení mřížky porovnávat, kromě ID objektů, také ID světelného zdroje. Jinak bychom nebyli při dělení schopni rozlišit, který stín patří ke kterému světlu a výsledek by byl (například pro dvě navzájem různě barevná světla) nepřesný. Při prolínání stínů je třeba uložit všechny identifikátory světel nebo je vhodně zkombinovat, aby vytvořili jiný unikátní identifikátor, který navzájem odliší všechny možnosti vzájemné kombinace, např. u dvou světelných zdrojů můžou nastat tři stavy, tj. první světlo, druhé světlo nebo prolnutí obou světel. Výhodou tohoto řešení je, že při dělení mřížky se porovnávají pouze dvě hodnoty a ne obsahy dvou polí navzájem.

3.2.1.2 Příklad kombinace identifikátorů

Jde o jednoduché pole bitů, nejlépe o velikosti registru (32 nebo 64 bitů) plněné pomocí bitového posunu.

[1] Například dosažením velikosti čtverce 2x2 pixely, kde by další dělení nemělo výsledku smysl.

Například pro 4 světla a 256 objektů použijeme 32 bitovou hodnotu, protože pro 256 světel potřebujeme 8 bitů ($2^8 = 256$) a pro 4 světla čtyři položky pole, tedy 4 x 8 = 32 bitů. Při zobrazování potom :

```
první světlo : pole = id_světla
další světla : pole = (pole << 4) OR id_světla
```

Pro 3 světla bude situace vypadat :

```
pole : {0000, id_světla_1, id_světla_2, id_světla_3}
```

3.2.1.3 Odrazy

Podobně jako u světel je nutné přidat ID odraženého objektu, aby byl odražený objekt podrobně vykreslen, tj. aby se dělila mřížka. Avšak je to nutné jenom pokud bude hloubka rekurze výpočtu odražených paprsků větší než 1. U hloubky 1 nám postačí indikace, zda-li je daný objekt odrazem či ne. Větší hloubka než 1 je pro zobrazení v reálném čase stále příliš výpočetně náročná.

3.2.1.4 Shrnutí

Adaptivní subsampling je výbornou technikou k urychlení sledování paprsku a vhodným nastavením, mřížky a hloubky dělení, můžeme dosáhnout vysoké kvality, srovnatelné s výstupem bez této techniky.

3.2.2 Využití grafického akcelerátoru

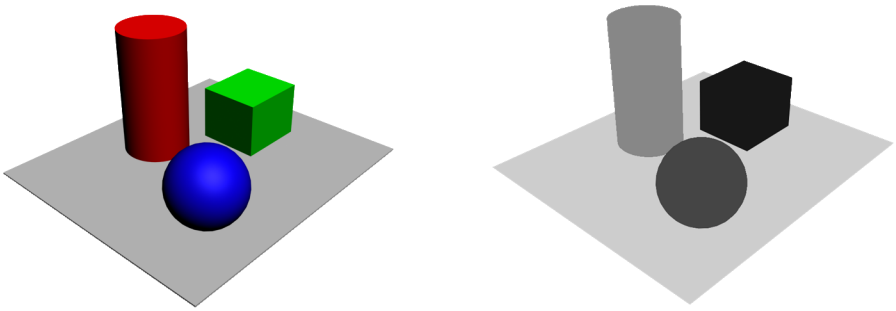
3.2.2.1 Interpolace mřížky

K interpolaci mřížky při subsamplingu lze také použít grafický 3D akcelerátor, který má hardwarově implementované kreslení trojúhelníků s Gouraudovým stínováním. Výsledný čtverec získáme složením ze dvou trojúhelníků. Můžeme navíc využít texturování, protože procedurální texturování je pro zobrazovaní v reálném čase stále příliš velká zátěž, proto je vhodné použít dvourozměrnou bitmapu. Získáme navíc filtrování textur, který přidá na výsledné kvalitě výstupu. Další výhodou, při odrazech, je možnost kombinovat více textur v jednom průchodu, tj. multitexturing.

Jelikož výpočet algoritmu sledování paprsku probíhá na procesoru, dojde při vykreslování k odlehčení zátěže, protože samotné vykreslení čtverce bude probíhat na grafické kartě.

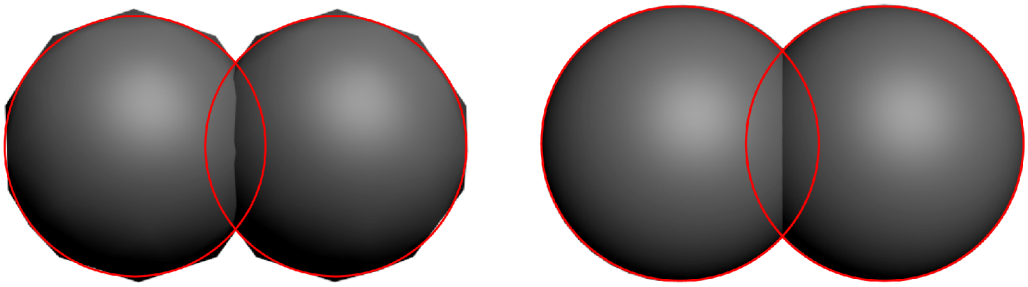
3.2.2.2 Průsečík paprsku s objektem

Urychlit lze také výpočet průsečíku s primárním paprskem u objektů, které lze zobrazit pomocí grafického akcelerátoru. Každý objekt vykreslíme do bitmapy, barva bude identifikátor objektu, a při vrhání paprsků zjistíme, podle barvy pixelu, který objekt byl paprskem zasažen.



Obrázek 3.8: Urychlení výpočtu primárních paprsků

Přesnost objektů na bitmapě bude záviset na počtu trojúhelníků vykresleného objektu. Z obrázku 3.9 je patrné, že při nižším počtu použitých trojúhelníků nebude objekt správně vykreslen. Při požití vysokého počtu trojúhelníků bude objekt vykreslen správně, ale při velkém počtu objektů dojde k velké zátěži grafického akcelerátoru, který potom bude výpočet brzdit, a tudíž nezískáme žádnou rychlost navíc.



Obrázek 3.9: Nepřesnost při vykreslení koule pomocí trojúhelíků (80 a 1260 trojúhelníků)

3.2.2.3 Využití GPU

Moderní grafické karty obsahují programovatelný procesor (*GPU*), který je vysoce specializovaný na vektorové operace. Jde vlastně o paralelní SIMD jednotky, které se používají pro rasterizaci obrazu a vektorové výpočty. Můžeme si jej představit jako obecnou paralelní jednotku pro výpočty s čísly v plovoucí řádové čárce.

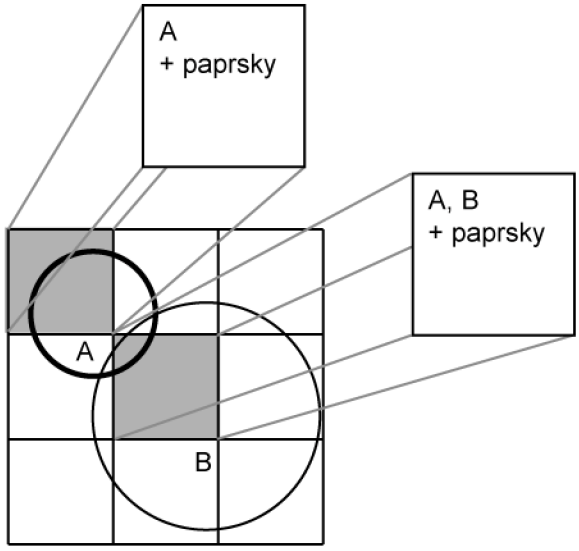
Velkou výhodou GPU je velký výpočetní výkon v plovoucí řádové čárce, který překoná rychlost výpočtu na procesoru. Další výhodou je nezávislost GPU na procesoru počítače, protože oba mají vlastní paměť a výpočet může běžet nezávisle na sobě. Takže zatímco GPU bude počítat průsečíky s objekty, procesor může připravovat další data procházením stromu.

3.2.2.4 Počítání průsečíků pomocí GPU

Procesor grafické karty má stále omezení v počtu vykonaných instrukcí a pomalé předvídání skoků, ovšem počítaní průsečíku není nemožné ().

Protože máme omezené možnosti, budeme tedy posílat data po menších částech, které budou snáze vykonatelné. Posílané data mohou být částí prostoru vzniklé při použití některé techniky pro dělení scény.

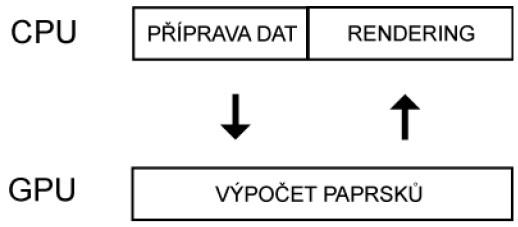
Data se musí přenášet na grafickou kartu přes textury, kde jeden pixel reprezentuje hodnotu v plovoucí řádové čárce. Tedy pro každou část pošleme všechny paprsky a seznam objektů, které se v dané části nacházejí.



Obrázek 3.10: Počítání průsečíků pomocí GPU

Výstupem bude textura, kde budou uložené informace o průniku pro každý paprsek. Ty pak zpracujeme na procesoru, který zároveň provede výsledné vykreslení (*rendering*).

Pro plné využití obou výpočetních jednotek (CPU a GPU) je vhodné najít optimální poměr mezi časem stráveným na přípravě dat, vykreslováním a výpočtem průsečíků, a data mezi jednotkami vhodně distribuovat.



Obrázek 3.11: Přesun dat mezi CPU a GPU

3.2.2.5 Shrnutí

Procesor grafické karty je výborným prostředkem pro urychlení vykreslování (zdroj [3]). Stále ale zůstává nevýhodou složitá implementace, např. spolu s dělením scény některou z technik. Nevýhodou je také neexistence vhodného vývojového nástroje a nástrojů pro ladění (debugger).

Těmito nevýhodami trpí generace dnešních grafických karet. V budoucnu, díky všestrannějšímu využití (např. fyzika, detekce kolizí), by mohla být vyřešena například efektivnější komunikace nebo lepší podpora pro větvení programu, tzn. lepší programovatelnost, což by vedlo k lepšímu použití pro urychlení sledování paprsku.

3.3 Optimalizační techniky pro programování

3.3.1 Paměťová náročnost

Přestože moderní procesory mají různé hardwarové optimalizační techniky (předvídání větvení, spekulativní vykonávání instrukcí apod.), jejich možnosti pro rozsáhlý kód jsou omezené. Proto je vhodné zorganizovat kód do menších výkonných částí, které lze potom lépe zoptimalizovat a jsou také vhodnější pro překladač.

Při výpočtu čte procesor data z paměti přes vyrovnávací paměť (L1 a L2 cache), která má větší přenosovou rychlost než operační paměť. Proto je vhodné minimalizovat čtení z operační paměti vhodným zarovnáním dat. Například ideální je, pokud jsou kód a data algoritmu průsečíku celá ve vyrovnávací paměti a procesor nemusí čekat na čtení z operační paměti.

4 Vlastní implementace

Nejprve byl implementován jednoduchý algoritmus sledování paprsku, který vytvářel korektní výstup, ale neobsahoval žádné optimalizace. Čas potřebný pro vykreslení jednoho snímku se pohyboval v řádech jednotek sekund. Poté následovala první verze, která již běžela několik málo snímků za sekundu a obsahovala základní optimalizace a vylepšenou strukturu programu. Ta byla postupně doplněna o některé další vybrané optimalizační techniky.

Pro testovací účely byly vytvořeny dvě oddělené implementace. První implementace používá pro výpočet pouze procesor počítače. Druhá implementace využívá navíc pro zobrazování grafický akcelerátor. Obě implementace byly vytvořeny ve Visual Studiu .NET v jazyce C++, který nabízí podporu pro objektově orientované programovaní.

4.1 Architektura programu

Obě implementace používají pro výpočet podobnou architekturu a liší se pouze ve fázi vykreslování, která je přizpůsobena používanému prostředku. Procesor provádí vykreslování do bitmapy (*frame buffer*), kde vznikne celý výsledný obraz, který je grafickou kartou zobrazen na výstup. Grafický akcelerátor je plněn daty postupně během výpočtu a výsledný obraz vznikne voláním funkce pro vykreslení po kompletním výpočtu scény. Pro komunikace s grafickým akcelerátorem byla použita knihovna OpenGL. Pro podporu softwarového zobrazování byla použita knihovna OpenPTC, která poskytuje jednoduchý přístup k frame-bufferu, tedy umožňuje přímý přístup k jednotlivým pixelům.

Celý systém se skládá z objektů kde základem jsou entity, se kterými pracuje algoritmus sledování paprsku. Tyto objekty jsou doplněny o funkce pro správu textur, objektů a celé scény. Hlavním prvkem je grafické jádro, které provádí výpočet a následné vykreslení celé scény.

Zobrazitelné entity (CEnt, CSphere, CPlane)

CEnt obsahuje abstraktní definici entity a základní funkce, např. výpočet průsečíku pro primární a sekundární paprsky, funkce pro výpočet normály a koordinátů pro textury. Každá entita má také funkci pro přepočítání konstant pro každý snímek a definování matice transformace.

Správa scény (CScene)

Obsahuje informace o všech objektech ve scéně. Umožňuje vkládání základních primitiv (koule, plocha) a světel. Pro každý zobrazitelný objekt lze nastavit materiál. Vlastnosti materiálu zahrnují

texturu (dvourozměrná bitmapa), barvu (v případě, že není textura), odrazivost a světelnou odrazivost (spekulární složku světla).

Pomocné objekty (CVec3, CMatrix, CRay, CLight, CMat, CTimer, CLog)

Mezi pomocné objekty patří funkce pro výpočty s vektory a maticemi, popis paprsku, popis světla a definice vlastností materiálu. Nezbytnou součástí je také časovač, který je nutný pro animace a také pro měření počtu snímků za sekundu. Poslední pomocný objekt obsahuje funkce pro záznam informací o průběhu, tzv. log.

Vykreslovací jádro (CRayScene)

Je komponentou pro vykreslení scény pomocí algoritmu sledování paprsku. Zde probíhá iniciliazace pro adaptivní dělení mřížky, funkce pro výpočet nejbližšího průsečíku paprsku s objektem a následné dělení mřížky.

Objekt také obsahuje nastavení pro vykreslování, např. hloubka rekurze sekundárních paprsků nebo hloubka dělení mřížky.

4.2 Vykreslovací jádro

Vykreslování probíhá v několika krocích, které navzájem navazují. Nejprve je třeba přepočítat konstanty objektů ve scéně. Poté je vržen pro každý pixel paprsek. Následně je nalezen nejbližší průsečík a podle vlastností materiálu a osvětlení je vyhodnocena barva příslušného pixelu. U stínu je nejprve vržen paprsek z místa průsečíku směrem ke světlu a poté dojde k vyhodnocení stínu nebo výpočtu osvětlení. Při odrazu je rekurzivním voláním vržen nový paprsek z místa průsečíku ve směru odraženého vektoru. Psedukód vypadá následovně :

```
vykreslení_scény ()
 pro každý pixel (x,y)
      obraz(x,y) = vyšli_paprsek(x,y)

rgb vyšli_paprsek (ray r)
 pro každý objekt s
      t = průsečík(r, s)
      nejbližší_t = MIN(nejbližší_t, t)
 jestliže je objekt zasažen
      vrať osvětlení_a_odraz(s, r, nejbližší_t)
 jinak
      vrať barvu_pozadí
```

Pro adaptivní subsampling je potřeba nejprve inicializovat mřížku. Poté se provede vyhodnocení bodů mřížky a každý čtverec je testován na dělení. Testování čtverce na dělení je voláno rekurzivně pro každý nově vzniklý čtverec. Testování na dělení čtverce probíhá, díky kombinaci identifikátorů, porovnáním čtyř hodnot. Pro každý nový snímek je nutné znovu přepočítat mřížku.

4.3 Seznam vlastností programu

Kromě základních vlastností algoritmu sledování paprsku (kap 2), implementuje program některé vybrané techniky z kap 3.

- Předpočítávání (*preprocesing*)
- Optimalizace pro primární a sekundární paprsky
- Paměť stínu (*shadow-cache*)
- Adaptivní dělení mřížky s podporou grafického akcelerátoru nebo softwarového vykreslování

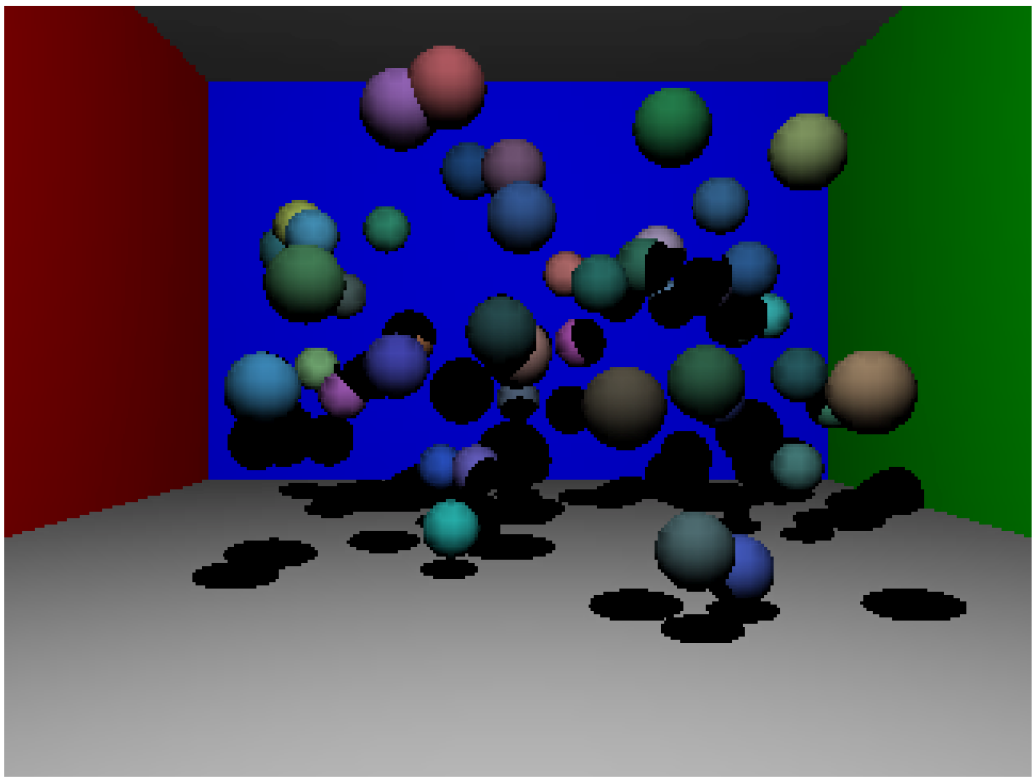
4.4 Měření

Pro každé měření byl použit program přesně na míru danému testu, aby nepotřebné funkce nezatěžovaly výpočet.

4.4.1 Testovací scény

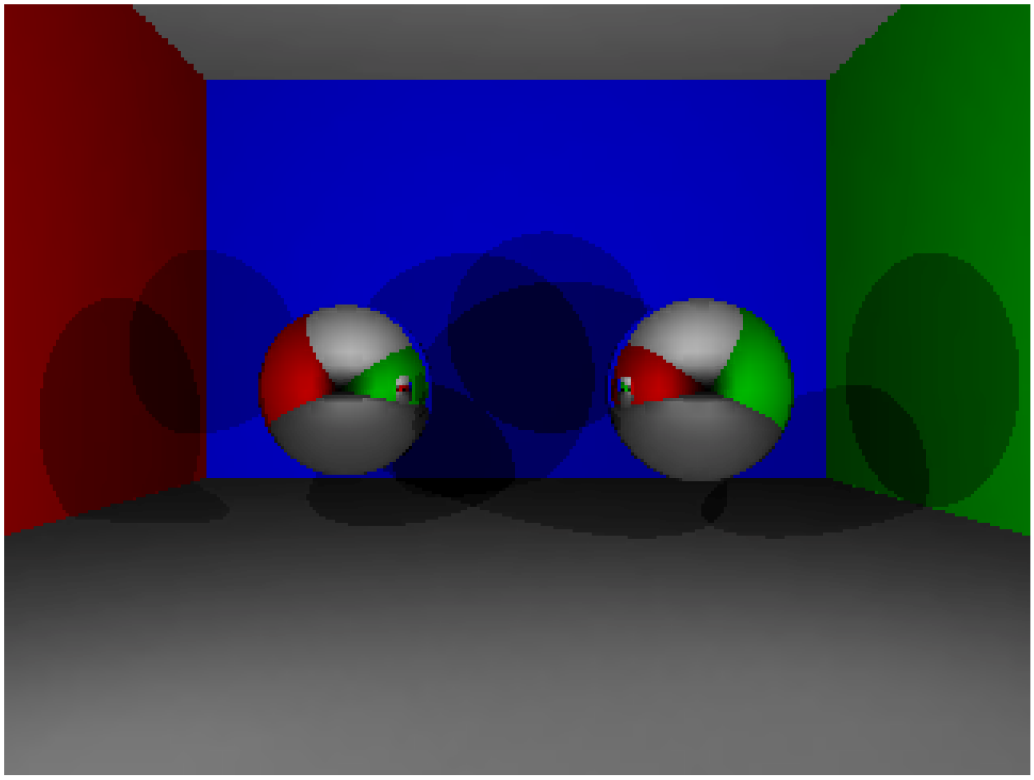
Pro testování byly vytvořeny dvě scény, kde každá klade různé zatížení na implementovaný algoritmus sledování paprsku.

První obsahuje velké množství objektů a pouze jeden světelný zdroj. Tato scéna se při výpočtu vyznačuje velkým množstvím vržených primárních paprsků a malým množstvím sekundárních paprsků, které slouží pouze pro výpočet stínu.



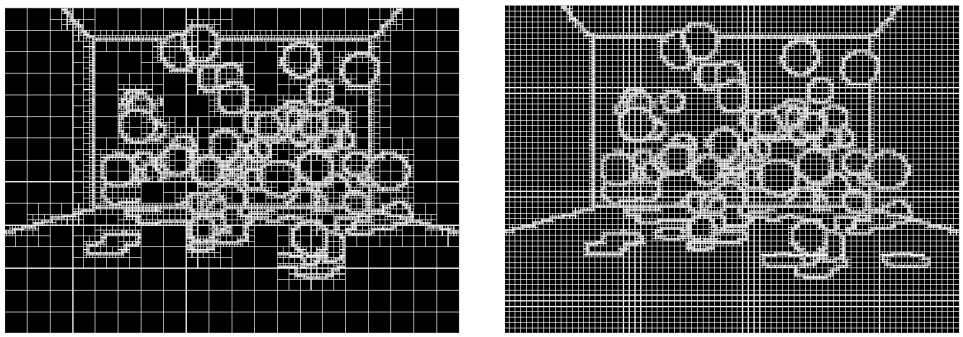
Obrázek 4.1: Testovací scéna 1

Druhá scéna naopak demonstruje velké množství vržených sekundárních paprsků, tj. stínových paprsků a paprsků vzniklých odrazem. Scéna obsahuje 5 světelných zdrojů, 2 koule a 5 rovin.

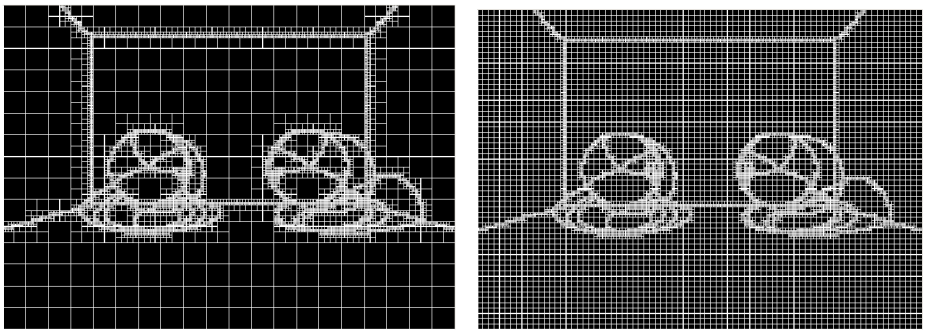


Obrázek 4.2: Testovací scéna 2

Při měření byla každá scéna vykreslována se dvěma nastaveními mřížky. Jednou je použita mřížka 20x15 čtverců, která by měla přinést velké zrychlení, avšak projevuje se chyba, která způsobuje nezobrazování malých objektů. Druhá mřížka má velikost 80x60 čtverců. Zde je již mřížka natolik hustá, že se zobrazují i velmi malé objekty.



Obrázek 4.3: Testovací scéna 1 – mřížka 20x15 a 80x60

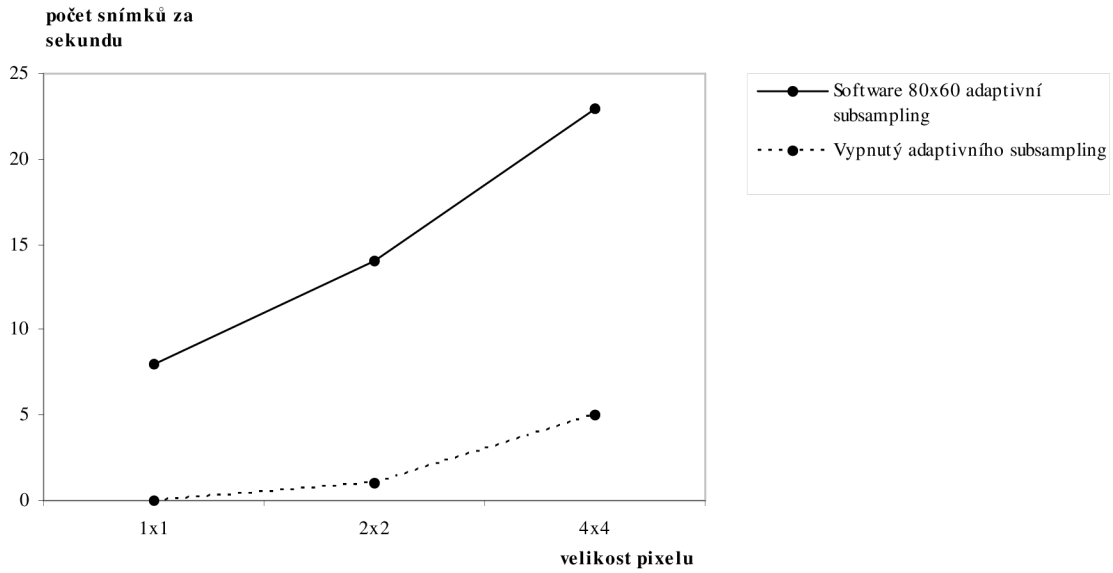


Obrázek 4.4: Testovací scéna 2 – mřížka 20x15 a 80x60

4.4.2 Výsledky měření

4.4.2.1 Adaptivní subsampling

Obě scény byly testovány[2] bez adaptivního subsamplingu, tedy pro každý bod je vyslán paprsek, a s adaptivním subsamplingem, který je vykreslován softwarově nebo pomocí OpenGL. U všech tří systémů bylo užito předpočítávání (*preprocesing*), optimalizace pro primární a sekundární paprsky a paměti stínu (*shadow-cache*).



Graf 4.1: Subsampling

[2] Pro testování byla použita konfigurace Intel Pentium M 1.7GHz, 1GB RAM, GeForce 6200

Z grafu 4.1 je viditelné že zrychlení, při použití adaptivního subsamplingu, je značné. V nejlepším případě byl počet snímků skoro 10krát vyšší, ale je to případ nejmenší mřížky, která značně degraduje kvalitu výstupu. Srovnatelná kvalita byla dosažena nastavením mřížky na velikost 80x60 čtverců. Zde narostl počet snímků skoro 7krát.

Podrobný test odhalil změnu při použití grafického akcelerátoru. Z tabulky 4.1 vidíme, že softwarové řešení je nepatrně rychlejší. Rozdíl je však zanedbatelný. Může to být způsobeno nedostatečně efektivním předáváním dat grafické kartě.

	OpenGL			**Software**		
Hloubka dělení mřížky	1x1	2x2	4x4	1x1	2x2	4x4
Scéna 1						
velikost mřížky	počet snímků za sekundu			počet snímků za sekundu		
20x15	6	13	26	9	15	30
80x60	5	12	26	8	14	23
Scéna 2						
velikost mřížky	počet snímků za sekundu			počet snímků za sekundu		
20x15	13	30	59	16	26	39
80x60	11	22	39	14	22	27

Tabulka 4.1: Adaptivní subsampling (OpenGL vs Software)

Velikost pixelu	1x1	2x2	4x4
Scéna 1	počet snímků za sekundu		
	0	1	5
Scéna 2	počet snímků za sekundu		
	0	1	6

Tabulka 4.2: Vypnutý adaptivní subsampling (Software)

4.4.2.2 Rozměry mřížky

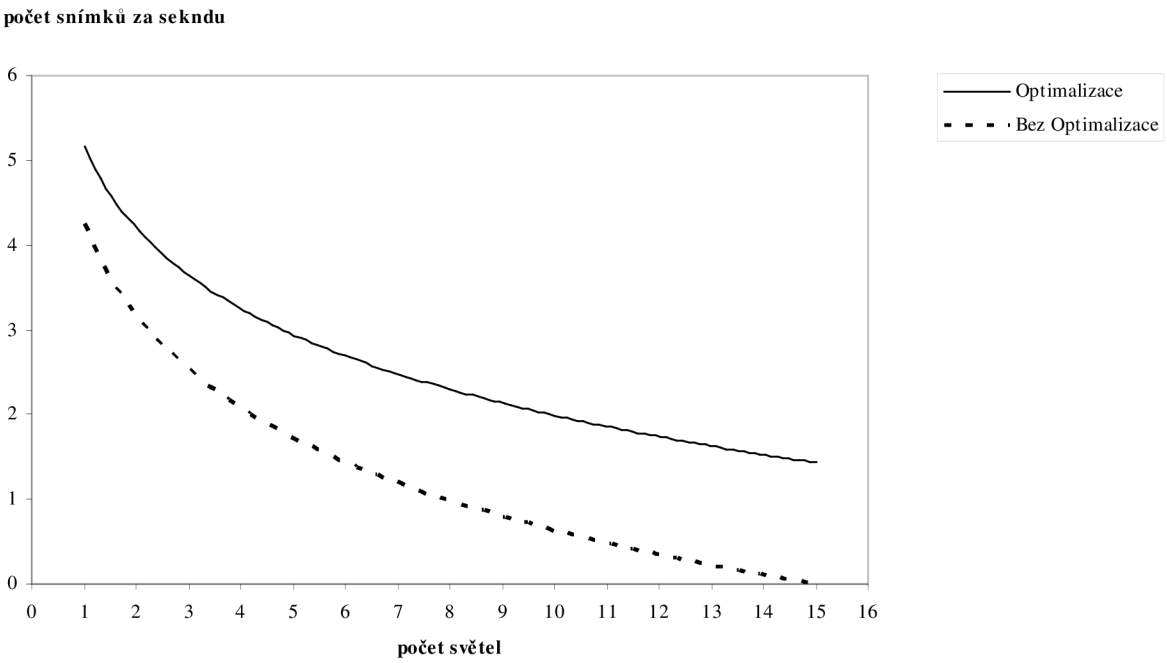
Z tabulky 4.3 vyplývá, že se zvětšující se mřížkou stoupá počet průsečíků a tím pádem klesá rychlost vykreslování. Velikost mřížky však musíme volit tak, aby zachytila nejmenší používaný objekt ve scéně. Při použití velmi malých detailů, které vedou k velké hustotě mřížky, ztrácí tato optimalizace smysl.

rozměry mřížky	snímky za sekundu	počet průsečíků
Scéna 1		
20x15	11	814 582
40x30	11	874 892
80x60	10	1 048 316
160x120	6	2 027 669
320x240	1	8 034 421
Scéna 2		
20x15	23	222 357
40x30	23	236 362
80x60	19	321 650
160x120	8	751 605
320x240	2	2 985 049

Tabulka 4.3: Rozměry mřížky

4.4.2.3 Ostatní optimalizace

Protože každá z ostatních optimalizací nepřináší tak značný rozdíl jako adaptivní subsampling, jsou ostatní optimalizace shrnuty do jednoho testu. Budeme tedy porovnávat implementaci základního algoritmu pro sledování paprsku s optimalizacemi v podobě předpočítávání (*preprocesing*), optimalizací pro primární a sekundární paprsky, a pamětí stínu (*shadow-cache*). Pro testování byla použita scéna s velkým množstvím objektů a měření probíhalo postupně pro různý počet světelných zdrojů.



Graf 4.2: Ostatní optimalizace

Z grafu 4.2 je patrné, že při malém počtu světel je vliv optimalizací malý. Také jde ale vidět, že při použití optimalizace klesá křivka grafu pomaleji. Je to způsobeno tím, že dojde k rychlejšímu vyhodnocení stínu díky optimalizaci, která nepočítá takové množství průsečíků.

4.4.3 Shrnutí

Adaptivní subsampling vedl při nastavení řídké mřížky ke značnému urychlení. Bohužel kvalita výstupu při tomto nastavení nesplňuje nároky pro praktické využití. Další studium této techniky by mohlo některé chyby odstranit.

Ostatní optimalizace nepřináší příliš velké urychlení výpočtu, avšak jejich implementace není náročná a je tedy vhodné je používat.

4.5 Demonstrační program

Pro lepší ukázku byl vytvořen demonstrační program, který používá adaptivní subsampling. Jsou také použity všechny ostatní optimalizace, které se vykytovaly v měření. Obsahuje obě testovací scény a scénu využívající texturování, vše se zobrazuje pomocí grafického akcelerátoru. Podrobnější popis ovládání a funkcí programu je v dodatku A.

5 Závěr

Tato práce ukazuje možné metody pro výpočet sledování paprsku v reálném čase jako alternativu k dnes běžně používané rasterizaci. Na základním principu algoritmu je ukázána jednoduchost implementace různých vlastností, kterých při rasterizaci dosahujeme velmi složitě. Všech běžných vlastností můžeme naopak dosáhnout velmi jednoduše.

Metod k urychlení výpočtu paprsku v reálném čase je mnoho. Některé ovlivňují výslednou kvalitu výstupu, avšak snaha je dosáhnout co nejvíce realistické zobrazení. Implementovaný adaptivní subsampling dosahuje dobré kvality zobrazení pouze pro jednoduché testovací scény. Pro složitější scény s mnoha detaily ve scéně by byla výsledná kvalita nedostačující.

Výhodné je využít ostatní techniky, které nedegradují kvalitu výstupu. Dělení scény je pro rozsáhlejší vizualizace výbornou technikou, kterou ukazuje mnoho jiných prací (např. [1]). Vhodné je také využití paralelního zpracování, které lze v algoritmu sledovaní paprsku jednoduše implementovat. Dnes běžně dostupné vícejádrové procesory s podporou instrukcí typu SIMD nabízejí dobré místo pro využití. Výborným prostředkem je také procesor grafické karty, který je stále vylepšován. Stává se tak obecnou jednotkou pro výpočet čísel v plovoucí řádové čárce, které můžeme využit pro urychlení výpočtu.

Všechny techniky, implementované v této práci, nepřinesly dostatečnou kvalitu pro praktické použití, ale vedly k podrobnému studiu algoritmu sledování paprsku, které podněcuje zájem o další studium.

Literatura

[1] Havran, V.: *Heuristic Ray Shooting Algorithms*. Disertační práce, ČVUT Praha, 2000.

[2] Bikker, J.: *Interactive raytracing*. http://www.intel.com, 2006.

[3] Elhassan, I.: *An Analysis Of GPU-based Interactive Raytracing*, University of Texas, 2006

[4] Parker, S., Martin, W., Sloan, P., Shirley, P., Smits, B., Hansen, Ch., *Interactive raytracing*. University of Utah, 1999

[5] UPGM, *Skripta pro předmět Počítačová grafika*, VUT Brno, Fakulta informačních technologií, 2005

[6] Wald, I., Slusallek, P., *State of art in interactive raytracing*, Saarland University, 2001

[7] Wikipedie: otevřená encyklopedie, http://www.wikipedia.org

Seznam obrázků

Obrázek 2.1: Phongův osvětlovací model 11
Obrázek 2.2: Lom světla 12
Obrázek 3.1: Dělení scény pomocí BSP stromu 18
Obrázek 3.2: BSP strom 18
Obrázek 3.3: Dělení scény pomocí KD stromu 19
Obrázek 3.4: KD strom 19
Obrázek 3.5: Obalující objem 20
Obrázek 3.6: Součet 4 hodnot pomocí SIMD instrukce 21
Obrázek 3.7: Adaptivní subsampling 23
Obrázek 3.8: Urychlení výpočtu primárních paprsků 26
Obrázek 3.9: Nepřesnost při vykreslení koule pomocí trojúhelíků (80 a 1260 trojúhelníků) 26
Obrázek 3.10: Počítání průsečíků pomocí GPU 27
Obrázek 3.11: Přesun dat mezi CPU a GPU 28
Obrázek 4.1: Testovací scéna 1 33
Obrázek 4.2: Testovací scéna 2 34
Obrázek 4.3: Testovací scéna 1 – mřížka 20x15 a 80x60 34
Obrázek 4.4: Testovací scéna 2 – mřížka 20x15 a 80x60 35

Seznam grafů

Graf 4.1: Subsampling 35
Graf 4.2: Ostatní optimalizace 38

Seznam tabulek

Tabulka 3.1: Výsledky měření ze semestrální práce 15
Tabulka 4.1: Adaptivní subsampling (OpenGL vs Software) 36
Tabulka 4.2: Vypnutý adaptivní subsampling (Software) 36
Tabulka 4.3: Rozměry mřížky 37

Dodatek A : Demonstrační program

Vlastnosti programu

- Přepočítávání (*preprocesing*)
- Optimalizace pro primární a sekundární paprsky
- Paměť stínu (*shadow-cache*)
- Adaptivní dělení mřížky s podporou grafického akcelerátoru nebo softwarového vykreslování
- Zobrazuje textury pomocí grafického akcelerátoru

Demonstrační program

Obrázek A.1: Demonstrační program

Při spuštění se program zeptá, zda spustit v okně nebo v režimu na celou obrazovku.

V horním pravém rohu se zobrazuje průměrný počet snímků za sekundu (*fps*). Vlevo se zobrazuje hloubka dělení mřížky (*subsampling*) a aktuální velikost mřížky (*grid*).

Ovládání

Program lze kdykoliv ukončit klávesou *Escape*. Stisknutím tlačítka *H* dojde k otevření nápovědy, která obsahuje všechny volby programu. Tlačítka *1*, *2*, *3* mění postupně scény, kde první dvě jsou identické ze scénami pro měření. Třetí scéna ukazuje objekty pokryté texturami, které můžeme vypnout klávesou *T*. Tlačítkem *M* lze přepínat velikost mřížky (20x15, 40x30, 80x60, 160x120). Zobrazit mřížku lze klávesou *G*.