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

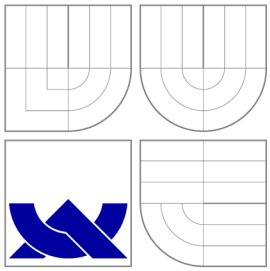
RAY-TRACING S VYUŽITÍM SSE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

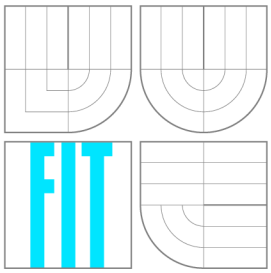
AUTOR PRÁCE
AUTHOR

JIŘÍ KUČERA

BRNO 2010



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

RAY-TRACING S VYUŽITÍM SSE

RAY-TRACING USING SSE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JIŘÍ KUČERA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ HAVEL
SUPERVISOR

BRNO 2010

Abstrakt

Tato práce se zabývá využitím SSE instrukcí k akceleraci výpočtů probíhajících při ray-tracingu. Aby bylo možné SSE instrukce co nejefektivněji použít, bylo zvoleno současné sledování čtyř paprsků uzavřených v jednom svazku. Byla provedena vektorizace algoritmů použitých v ray-tracingu a také bylo navrženo a implementováno řešení rozpadu svazku paprsků. Provedenými testy pak byla sledována doba renderování obrazu pro případ, kdy jsou všechny paprsky pohromadě, ale také pro případ, kdy se ve svazku nachází pouze jeden paprsek.

Abstract

This thesis describes the acceleration technique of ray-tracing method using SSE instruction set. Choosing the parallel tracing of four rays enclosed in one beam turned to be the best way of using SSE effectively. Also the vectorization of algorithms which are used in ray-tracing method was implemented. The solution of beam splitting was designed and implemented too. The time for rendering image was monitored in the tests — one for case when the beam includes all the rays and one for case when there is just one ray in the beam.

Klíčová slova

ray-tracing, Streaming SIMD Extensions, SSE, vektorové instrukce, svazek paprsků, rozpad svazku paprsků, vektorizace, optimalizace pomocí SSE

Keywords

ray-tracing, Streaming SIMD Extensions, SSE, SIMD instructions, beam of rays, beam of rays splitting, vectorization, optimizing using SSE

Citace

Jiří Kučera: Ray-tracing s využitím SSE, bakalářská práce, Brno, FIT VUT v Brně, 2010

Ray-tracing s využitím SSE

Prohlášení

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

.....
Jiří Kučera
19. května 2010

Poděkování

Na tomto místě bych chtěl poděkovat svému vedoucímu, panu Ing. Jiřímu Havlovi, za jeho rady a odbornou pomoc při vypracování této práce. Dále bych chtěl také poděkovat svým rodičům za jejich podporu při studiu.

Obsah

1 Úvod **3**

2 Metoda ray-tracing **5**
2.1 Vznik a vývoj ray-tracingu . . . 5
2.2 Princip metody ray-tracing . . . 5
2.3 Kamera a systém souřadnic . . . 6
2.4 Určení pozice paprsku vůči tělesům . . . 7
2.4.1 Koule . . . 7
2.4.2 Rovina . . . 8
2.4.3 Trojúhelník . . . 9
2.5 Osvětlovací model . . . 9
2.5.1 Rozšířený Phongův osvětlovací model . . . 9
2.5.2 Ostatní osvětlovací modely . . . 12
2.6 Optimalizace metody ray-tracing . . . 12

3 Instrukční sada SSE **14**
3.1 Předchůdci SSE — instrukční sady MMX a 3DNow! . . . 14
3.2 Charakteristika instrukční sady SSE . . . 15
3.2.1 Novější verze instrukční sady SSE . . . 16
3.3 Programovací techniky . . . 17
3.3.1 Podpora SSE instrukcí v jazycích `C` a `C++` . . . 17
3.3.2 Problém vektorizace . . . 17
3.3.3 Oranizace dat v paměti . . . 18
3.3.4 Podmíněné přiřazení . . . 18
3.3.5 Dělení a reciproké instrukce . . . 19

4 Návrh a implementace **21**
4.1 Možnosti použití SSE instrukcí v ray-tracingu . . . 21
4.2 Existující implementace ray-traycingu využívající SSE instrukce . . . 21
4.3 Řešení rozpadu svazku paprsků . . . 22
4.3.1 Algoritmus rozpadu svazku na podsvazky . . . 23
4.4 Vlastní aplikace . . . 26
4.4.1 Volba programovacího jazyka a překladače . . . 26
4.4.2 Struktura aplikace . . . 27
4.4.3 Datové struktury — svazek paprsků . . . 28
4.4.4 Emulace složitějších matematických funkcí SSE instrukcemi . . . 29
4.4.5 Implementace skalárního součinu dvou vektorů . . . 29
4.4.6 Zarovnání dynamicky alokovaných dat . . . 30

4.4.7 Ladění aplikace 30

5 Průběh testů a dosažené výsledky **31**
5.1 Použité testovací scény 31
5.1.1 Scéna 1: „*Whitted sphere*“ 31
5.1.2 Scéna 2: „*Giza*“ 31
5.1.3 Scéna 3: „*Billiard*“ 31
5.2 Parametry a spouštění testů 32
5.3 Výsledky testů 33
5.3.1 Scéna 1 33
5.3.2 Scéna 2 34
5.3.3 Scéna 3 34
5.4 Vliv způsobu výpočtu matematických funkcí pomocí SSE instrukcí na kvalitu obrazu 35

6 Závěr **37**

A Obsah CD a tabulky s výsledky testů **41**
A.1 Obsah CD 41
A.2 Tabulky s výsledky všech testů 41

Kapitola 1

Úvod

Metoda ray-tracing se používá při renderování velmi realistických[1] obrazů prostorových scén. V současnosti nachází ray-tracing uplatnění především ve filmovém průmyslu, kde se používá hlavně při tvorbě vizuálních efektů (například efekty ve filmu Avatar byly vytvořeny programem Autodesk Maya[2], který používá program *mental ray*[3], jenž je založený na metodě ray-tracing). Kromě specializovaných aplikací využívajících ray-tracing (jako je Blender[4] nebo již zmiňovaná Maya) existují i jeho samostatné implementace — raytracery. Asi nejznámějším a nejpropracovanějším raytracerem je POV-Ray[5].

Vysoká kvalita obrazu získaného ray-tracingem je vykoupena jeho obrovskými výpočetními nároky. Ačkoli je metoda ray-tracing známá více než čtyřicet let, větší zájem o ni se začal projevovat teprve s příchodem výkonnější výpočetní techniky. Zlom nastal při zavedení SIMD[6] instrukcí, zejména pak zaváděním různých verzí rozšiřující instrukční sady SSE firmou Intel. Vhodným použitím SSE instrukcí bylo možno dosáhnout i několikanásobného zrychlení často používaných výpočtů, jakými jsou například operace s vektory. Dokonce se začaly objevovat implementace ray-tracingu pracující v reálném čase[7].

Implementace ray-tracingu v hardware zatím zůstává na experimentální úrovni. Za zmínku stojí RPU (Ray Processing Unit) vyvinutý na univerzitě v Sársku[21].

Tato práce se zabývá využitím různých verzí instrukční sady SSE (konkrétně verzí SSE a SSE2 — novější verze SSE3 a SSE4 přináší instrukce, které pracují s daty uloženými v jiném formátu, než jaký byl použit v programové části této práce) k akceleraci výpočtů probíhajících při ray-tracingu. Metodě ray-tracing je věnována následující kapitola, která je pojata jako malý průvodce dějí probíhajícími v ray-tracingu od začátku renderování obrazu až po jeho konečné zobrazení na výstupu. Na konci kapitoly je pak uveden soupis některých optimalizačních technik, které se při urychlování metody ray-tracing nejčastěji používají.

Třetí kapitola se zabývá instrukční sadou SSE, jejími přednostmi, omezeními, rozdíly mezi jednotlivými verzemi této sady a doporučenými programovacími technikami, které vedou k jejímu efektivnímu využití.

Návrh a implementace praktické části této práce je probírán ve čtvrté kapitole, včetně

[1] Samotný ray-tracing se nedokáže vypořádat s některými optickými jevy. K dosažení fotorealističnosti renderovaného obrazu bývá proto ray-tracing kombinován s dalšími metodami (viz následující kapitola).

[2] http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13577897

[3] http://www.mentalimages.com/products/mental-ray.html

[4] http://www.blender.org/

[5] http://www.povray.org/

[6] *Single instruction, multiple data* — jedinou instrukcí lze vykonat konkrétní operaci s několika datovými položkami paralelně.

[7] Jedná se především o knihovnu *OpenRT* vytvořenou na základě disertační práce Ingo Walda[18].

problémů, které se při implementaci objevily a jejich řešení. V páté kapitole jsou srovnány výsledky testů praktické části využívající výhod instrukční sady SSE vůči verzi, která používá k výpočtům pouze instrukční sadu matematického koprocesoru. V testech jsou zahrnuty i vlivy použitých překladačů na výkon a také i vlivy různých operačních systémů.

V závěru práce jsou pak zhodnoceny dosažené výsledky a uvedeny další možnosti pokračování vývoje, kterými se může tato práce v budoucnu dále zabývat.

Obrázek 1.1: *Jednou z implementací ray-tracingu využívající výhod SSE instrukcí je i knihovna OpenRT. Scéna na obrázku je renderována v reálném čase, obrazové snímky zobrazující se v televizoru jsou přijímány pomocí multicastu. Obrázek pochází z* `<http://openrt.de/Applications/mr.php>`.

Kapitola 2

Metoda ray-tracing

2.1 Vznik a vývoj ray-tracingu

Základní algoritmus metody *ray-tracing* (sledování paprsku) vyvinul[1] a poprvé v počítačové grafice použil roku 1968 Arthur Appel, který ji ve své práci[2] nazval *point-by-point shading* (později se pro tuto metodu ujal název *ray-casting* — vrhání paprsku). Appelovu metodu pak rozšířil v roce 1980 Turner Whitted[20] a bylo s ní možné simulovat odraz a lom světelných paprsků, což vedlo ke zvýšení realističnosti obrazu výsledné scény. V současnosti se pod pojmem *ray-tracing* myslí metoda, kterou vyvinul právě Whitted[2].

2.2 Princip metody ray-tracing

Metoda ray-tracing vytváří dvojrozměrný obraz trojrozměrné scény pomocí simulace šíření světelných paprsků okolním prostředím. V reálném světě se světlo šíří od svého zdroje do okolí, kde se odráží, láme nebo je pohlcováno tělesy, které mu přijdou do cesty. Jen část světla pak dorazí k pozorovateli, kde je vnímána jako obraz.

U ray-tracingu se používá opačný postup[22] — paprsky jsou vysílány z místa pozorovatele do scény. Světelný paprsek je reprezentován polopřímkou, jednotlivé světelné paprsky jsou na sobě nezávislé. Tělesa ve scéně jsou složena z geometrických primitiv (koule, rovina, trojúhelník aj.) a každému tělesu ve scéně přísluší popis charakterizující jeho optické vlastnosti. Ty závisí na použitém osvětlovacím modelu.

Na obrázku 2.1 je znázorněn princip, na kterém je metoda založena.

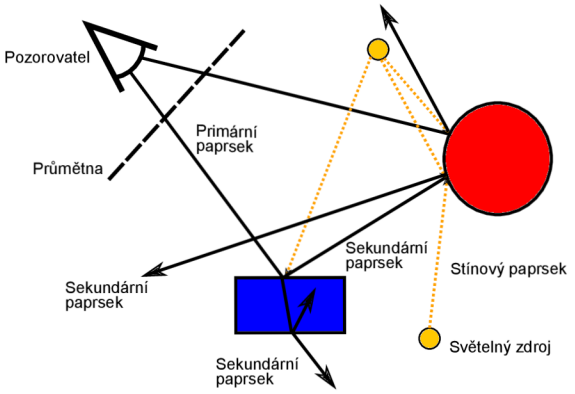
Cílem každého *primárního paprsku* je zjistit barvu bodu na průmětně přes který právě prochází. Nejprve dochází k hledání průsečíku paprsku s nejbližším tělesem ve scéně. Po jeho nalezení mohou vzniknout v místě dopadu až dva *sekundární paprsky* a tolik *stínových paprsků*, kolik je ve scéně světelných zdrojů.

Stínové paprsky[22] mají za úkol zjistit, zda jsou zdroje světla z místa, odkud byly vyslány, viditelné. Pokud ano, je jejich vliv zahrnut do výpočtu celkové barvy primárního nebo sekundárního paprsku v místě dopadu na těleso. Tímto způsobem lze ve scéně dosáhnout stínů vrhaných tělesy. Takto vytvořené stíny však mají ostré přechody, protože většina implementací ray-tracingu používá bodové zdroje světla.

[1] Podobný algoritmus vyvinuli také Goldstein a Nagel, kteří ho použily při simulaci trajektorií balistických střel a jaderných částic[7].

[2] V literatuře je tato metoda obvykle uváděna pod výstižnějšími názvy, např. *recursive ray tracing*[7] nebo *sledování paprsku vyššího řádu*[22].

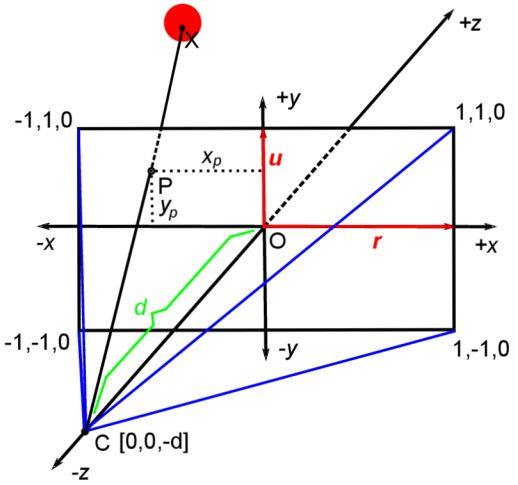
Sekundární paprsky[22] jsou dvojího druhu — odražený a lomený. S těmito paprsky jsou i nadále hledány nejbližší průsečíky s tělesy ve scéně, kde dochází ke vzniku dalších sekundárních paprsků. Celý proces se opakuje tak dlouho, dokud paprsky neopustí prostor scény nebo nebylo dosaženo stanovené hloubky rekurze. Při návratu z rekurze dochází ke kombinaci barevných informací zjištěných paprsky a výsledná barva je pak zobrazena na průmětnu.



Obrázek 2.1: *Znázornění ray-tracingu. Paprsky rekurzivně produkují další paprsky, v závislosti na optických vlastnostech povrchu těles ve scéně.*

2.3 Kamera a systém souřadnic

Aby bylo možno vyslat paprsek od pozorovatele do scény, je třeba určit jeho směr. Na obrázku 2.2 je znázorněný model kamery založený na *perspektivní projekci*, v ray-tracingu často používaný.



Obrázek 2.2: *Princip kamery v ray-tracingu založené na perspektivní projekci.*

Bod C, ve kterém se sbíhají všechny paprsky protínající průmětnu, leží na kolmici vycházející z jejího středu. V tomto bodě je umístěno oko pozorovatele nebo čočka kamery a přímka CO bývá označována jako *optická osa kamery*[22]. Pro x-ovou a y-ovou souřadnici libovolného bodu průmětny platí[7]

$$-k\,|\boldsymbol{r}| \leq x \leq k\,|\boldsymbol{r}|\,, \qquad -|\boldsymbol{u}| \leq y \leq |\boldsymbol{u}|\,,$$

kde k vyjadřuje poměr stran výsledného obrazu ($\frac{\check{s}\acute{i}rka}{v\acute{y}\check{s}ka}$), vektory $\boldsymbol{r}$ a $\boldsymbol{u}$ jsou jednotkové. Poměr k má důležitý význam. Při ray-tracingu je plocha průmětny rozdělena na síť malých plošek, které odpovídají pixelům ve výsledném obrazovém rastru. Pokud jsou šířka a výška obrazu různé a je-li k opomenuto, plošky budou mít obdélníkový tvar a důsledkem toho dojde k deformaci zobrazení těles na průmětně (koule se bude jevit jako elipsoid apod.).

Směr paprsku vyslaný z bodu C přes bod P (viz obrázek 2.2) skrz průmětnu do scény určuje vektor $(P - C) = (x_p, y_p, d)$. Je-li směr pohledu kamery (vektor $(C - O)$) normalizovaný, pak směrový vektor paprsku CP je $(\frac{x_p}{d}, \frac{y_p}{d}, 1)$. Při určení směru paprsku je lepší použít vektorový součet, rovnice paprsku má pak tvar[7]

$$X = C + t \cdot \left(\frac{k \cdot x_p}{d} \cdot \boldsymbol{r} + \frac{y_p}{d} \cdot \boldsymbol{u} + \boldsymbol{d} \right), \tag{2.1}$$

kde $\boldsymbol{d}$ je normalizovaný vektor $(C - O)$.

Vzdálenost d kamery od průmětny určuje úhel pod jakým se paprsky rozbíhají do scény. Pokud je d příliš malé, dochází v blízkosti okrajů průmětny k deformacím zobrazovaných objektů.

2.4 Určení pozice paprsku vůči tělesům

V momentě, kdy byl paprsek vyslán do scény, jsou hledány odpovědi na následující otázky: „Nachází se v trajektorii paprsku nějaká překážka?“ a „Jak daleko je průsečík s první překážkou, která stojí paprsku v cestě?“. Nejjednodušší způsob, jak na tyto otázky odpovědět, je testovat pozici paprsku vůči každému tělesu ve scéně a pamatovat si jen to těleso, jehož průsečík s paprskem je nejblíže počátku paprsku. Vzhledem k tomu, že tělesa ve scéně jsou složena z geometrických primitiv, které lze charakterizovat matematicky, je určení vzájemné pozice paprsku a tělesa zároveň řešením příslušné matematické rovnice.

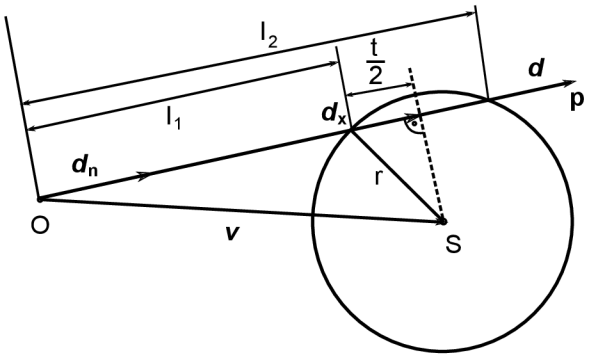
2.4.1 Koule

Koule patří, co do výpočtu, mezi nejjednodušší primitivy. Analyticky ji lze vyjádřit rovnicí

$$(x - S_x)^2 + (y - S_y)^2 + (z - S_z)^2 = r^2. \tag{2.2}$$

K jejímu popisu tedy stačí udat polohu středu S a velikost poloměru r. Vzájemnou polohu koule a paprsku pak lze řešit buď dosazením parametrického vyjádření paprsku (rovnice 2.1) do rovnice 2.2 za x, y, z a následného nalezení kořene t nebo geometricky[8], jak uvádí algoritmus 2.1.

Algoritmus 2.1. Geometrické určení pozice koule vůči paprsku. Koule je dána středem S a poloměrem r, paprsek p má počátek v bodě O a jeho směr je dán jednotkovým vektorem $\boldsymbol{d}_n$ (obrázek 2.3).



Obrázek 2.3: *Poloha paprsku vůči kouli — geometrické řešení.*

1. Nechť $\left(\frac{t}{2}\right)^2 = |\boldsymbol{d}_x|^2 - |\boldsymbol{v}|^2 + r^2$, kde $\boldsymbol{v} = S - O$ je vektor směřující z bodu O do bodu S a $|\boldsymbol{d}_x| = \boldsymbol{v} \cdot \boldsymbol{d}_n$ je velikost pravoúhlého průmětu vektoru $\boldsymbol{v}$ do paprsku p.
2. Pokud platí $\left(\frac{t}{2}\right)^2 < 0$, pak paprsek p kouli minul.
3. Nechť $l_2 = |\boldsymbol{d}_x| + \frac{t}{2}$.
4. Je-li $l_2 < 0$, pak počátek O paprsku p leží mimo kouli a paprsek p směřuje pryč od koule.
5. Nechť $l_1 = |\boldsymbol{d}_x| - \frac{t}{2}$.
6. Je-li $l_1 < 0$, pak počátek O paprsku p leží uvnitř koule a l_2 určuje vzdálenost průsečíku paprsku s koulí od počátku O. Jinak počátek O paprsku p leží mimo kouli a vzdálenost mezi průsečíkem paprsku s koulí a počátkem O je l_1.

2.4.2 Rovina

K určení pozice roviny ve scéně stačí znát normálový vektor $\boldsymbol{n}$ na ni kolmý a bod P (obrázek 2.4). Rovinu lze vyjádřit rovnicí

$$ax + by + cz + d = 0, \tag{2.3}$$

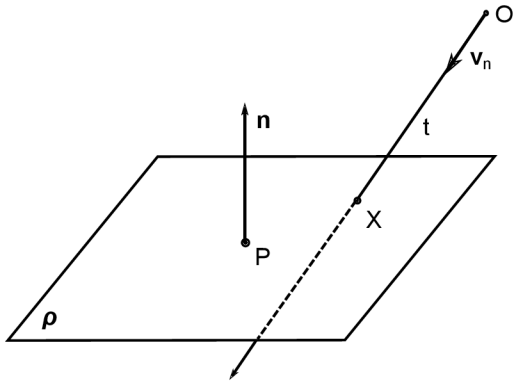
kde koeficienty a, b a c jsou zároveň složkami vektoru $\boldsymbol{n} = (a, b, c)$. Vzhledem k tomu že bod P leží na rovině ρ, lze koeficient d rovnice 2.3 vyjádřit jako

$$d = -\left(ax_P + by_P + cz_P\right) = -\left(a, b, c\right) \cdot \left(x_P, y_P, z_P\right) = -\left(\boldsymbol{n} \cdot P\right). \tag{2.4}$$

Vzdálenost t mezi počátkem paprsku O a jeho průsečíkem X s rovinou ρ určuje vztah[8]

$$t = \frac{\boldsymbol{n} \cdot O + d}{-\boldsymbol{n} \cdot \boldsymbol{v}_n}. \tag{2.5}$$

Jmenovatel $-\boldsymbol{n} \cdot \boldsymbol{v}_n$ určuje polohu paprsku vůči rovině. Je-li kladný, paprsek vchází do roviny z poloprostoru, do kterého směřuje vektor $\boldsymbol{n}$. Je-li záporný, paprsek do roviny vchází z opačného poloprostoru a pokud je roven nule, paprsek je s rovinou rovnoběžný. Pokud vyjde hodnota t záporná, rovina a paprsek nemají žádný společný bod.



Obrázek 2.4: *Znázornění roviny.*

2.4.3 Trojúhelník

Trojúhelník patří v počítačové grafice mezi nejvýznamnější geometrická primitiva. Každé těleso ve scéně lze totiž vyjádřit sítí trojúhelníků. Trojúhelník je určen třemi body A, B a C. Při výpočtu[3, 15] průsečíku paprsku a trojúhelníku je nejprve určena rovina, ve které trojúhelník leží

$$\rho = \{A; \boldsymbol{N} = (B - A) \times (C - A)\} \tag{2.6}$$

a následně je zjištěna souřadnice průsečíku této roviny s paprskem. Informaci, zda průsečík leží uvnitř trojúhelníku nebo mimo něj, lze zjistit pomocí *barycentrických souřadnic*

$$P = wA + uB + vC, \qquad 0 \leq w \leq 1,\ 0 \leq u \leq 1,\ 0 \leq v \leq 1, \tag{2.7}$$

kde pokud platí $w + u + v = 1$, bod P leží uvnitř trojúhelníku. Častěji se ale používá tvar

$$P - A = u\,(B - A) + v\,(C - A)\,, \qquad u \geq 0,\ v \geq 0, \tag{2.8}$$

kde pokud bod P leží uvnitř trojúhelníku, platí $u + v \leq 1$.

2.5 Osvětlovací model

Osvětlovací model se používá při modelování optických jevů, ke kterým dochází na povrchu těles. Existují dva druhy osvětlovacího modelu — *lokální* a *globální*[22]. V ray-tracingu se používá *lokální osvětlovací model*, tj. vypočítává se osvětlení jediného bodu na povrchu tělesa. Sekundární zdroje světla, vzniklé odrazem nebo lomem paprsků pocházejících přímo od světelných zdrojů, se neuvažují. Naproti tomu *globální osvětlovací model*[7] se sekundárními zdroji světla pracuje a lze se s ním setkat u metody *radiozita*.

V následujících několika odstavcích bude jako příklad uveden rozšířený Phongův osvětlovací model, který je poměrně nenáročný na výpočet, snadno se implementuje a je použit v programové části této práce.

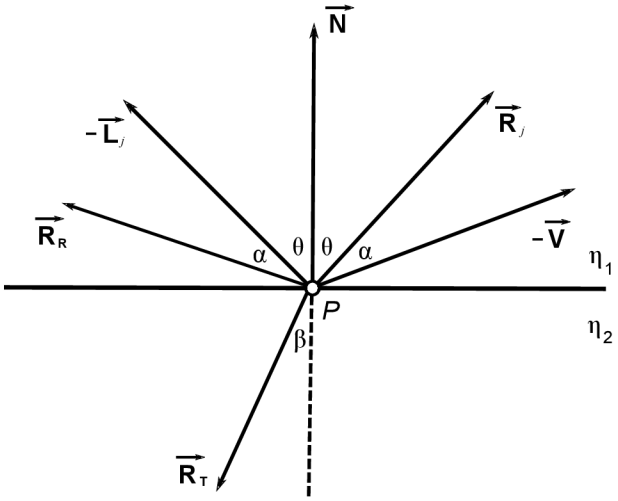
2.5.1 Rozšířený Phongův osvětlovací model

Tento model navrhl a poprvé použil Turner Whitted[20], který vyšel z Phongova osvětlovacího modelu, vyvinutého roku 1975 Bui Tuong Phongem[14]. Phongův osvětlovací model

se používá pro výpočet intenzity odraženého světla od povrchu tělesa a je matematicky vyjádřen vztahem[22]

$$I = I_a + \sum_{j=1}^{n} I_{L_j} \left[k_d \left(\boldsymbol{L}_j \cdot \boldsymbol{N} \right) + k_s (\boldsymbol{V} \cdot \boldsymbol{R}_j)^h \right], \tag{2.9}$$

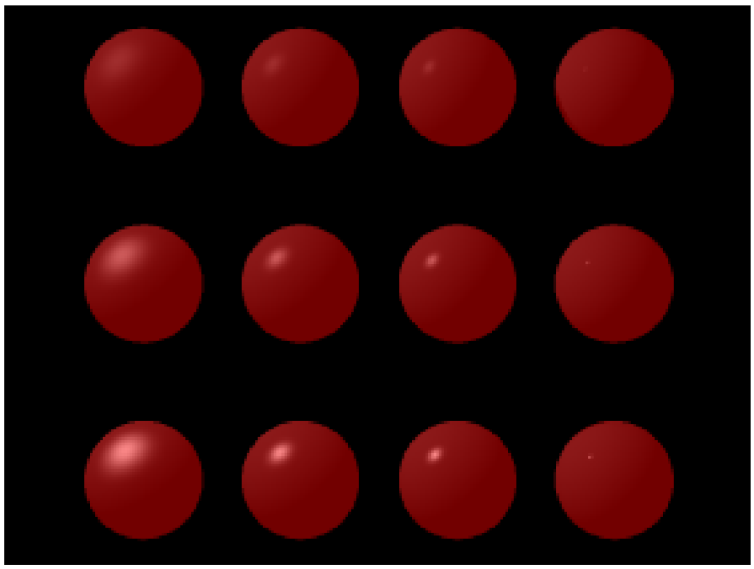
kde I_a označuje *ambientní složku*,
n je celkový počet světelných zdrojů ve scéně,
I_{L_j} označuje barevné složení paprsku přicházejícího od j-tého zdroje světla,
k_d je koeficient difúzního odrazu,
k_s je koeficient spekulárního odrazu,
h je koeficient vyjadřující ostrost zrcadlového odrazu
(čím je h větší, tím jsou odrazy na povrchu tělesa ostřejší),
$\boldsymbol{L}_j$ označuje opačný směr paprsku přicházejícího od j-tého světelného zdroje,
$\boldsymbol{N}$ je normála k povrchu tělesa v bodě P (viz obrázek 2.5),
$\boldsymbol{V}$ označuje opačný směr ze kterého přichází primární nebo sekundární paprsek
a konečně $\boldsymbol{R}_j$ je směr odrazu paprsku z j-tého zdroje světla, určeného
dle vztahu $\boldsymbol{R}_j = 2 \left(\boldsymbol{L}_j \cdot \boldsymbol{N} \right) \boldsymbol{N} - \boldsymbol{L}_j$.



Obrázek 2.5: *Vektory v rozšířeném Phongově osvětlovacím modelu.*

Ambientní složka[22] vyjadřuje množství okolního rozptýleného světla, které přichází ze všech směrů a vzniklo mnohonásobnými odrazy od ostatních těles. Její hodnota bývá pro celou scénu konstantní. Součin $\boldsymbol{L}_j \cdot \boldsymbol{N}$ vyjadřuje kosinus úhlu mezi normálou k povrchu tělesa a paprskem přicházejícím od světelného zdroje. Spolu s I_{L_j} tvoří Lambertův zákon a určuje množství světla v bodě dopadu světelného paprsku $\boldsymbol{L}_j$. Jak je světlo odražené od povrchu tělesa vnímáno pozorovatelem udává vztah $I_{L_j} (\boldsymbol{V} \cdot \boldsymbol{R}_j)^h$. Čím větší je úhel mezi směrem pohledu pozorovatele $\boldsymbol{V}$ a odraženým světelným paprskem $\boldsymbol{R}_j$, tím méně odraženého světla pozorovatel vidí.

Whitted rozšířil vztah 2.9 o další dvě složky, z nichž jedna představuje barevný příspěvek



Obrázek 2.6: *Phongův osvětlovací model v praxi. Hodnoty koeficientů ostrosti zrcadlového odrazu jsou zleva doprava* 5, 20, 50 *a* 1000. *Hodnoty koeficientů spekulárního odrazu jsou shora dolů* 0,1, 0,275 *a* 0,45. *Obrázek byl pořízen programem vytvořeným v rámci této práce dle předlohy v [7] (autorem předlohy je David Kurlander).*

odraženého paprsku v daném bodě a druhá paprsku lomeného[20]

$$I = I_a + \sum_{j=1}^{n} I_{L_j} \left[k_d \left(\boldsymbol{L}_j \cdot \boldsymbol{N} \right) + k_s (\boldsymbol{V} \cdot \boldsymbol{R}_j)^h \right] + k_s I_R + k_t I_T. \tag{2.10}$$

I_R a I_T vyjadřují barevné složení přicházejícího odraženého a lomeného paprsku, koeficient lomu k_t určuje výši barevného příspěvku lomeného paprsku. Pro zlepšení vlastností modelu se doporučuje nahradit koeficienty k_s a k_t funkcemi zahrnujícími Fresnelovy rovnice[20].

K simulaci útlumu paprsku procházejícího transparentním tělesem lze použít *Beerův zákon*[4]

$$I_{out} = I_{in} e^{-dC}, \tag{2.11}$$

kde d vyjadřuje vzdálenost, kterou paprsek tělesem urazí a C je konstanta určující absorbční schopnost materiálu ze kterého je těleso vyrobeno. Čím je C menší, tím menší je i útlum světla.

Směr odraženého paprsku $\boldsymbol{R}_R$ je dán vztahem

$$\boldsymbol{R}_R = \boldsymbol{V} - 2 \left(\boldsymbol{V} \cdot \boldsymbol{N} \right) \boldsymbol{N}, \tag{2.12}$$

který je odvozen ze zákona odrazu (stejně jako vztah pro $\boldsymbol{R}_j$ uvedený výše). Pro směr lomeného paprsku $\boldsymbol{R}_T$ platí vtah

$$\boldsymbol{R}_T = \frac{\eta_1}{\eta_2} \boldsymbol{V} + \left[- \left(\boldsymbol{V} \cdot \boldsymbol{N} \right) \frac{\eta_1}{\eta_2} - \sqrt{1 - \left(\frac{\eta_1}{\eta_2} \right)^2 \left[1 - \left(\boldsymbol{V} \cdot \boldsymbol{N} \right)^2 \right]} \right] \boldsymbol{N} \tag{2.13}$$

odvozený[3] podle Snellova zákona lomu $\eta_1 \sin \alpha = \eta_2 \sin \beta$. Pokud je hodnota výrazu pod odmocninou záporná, dochází k úplnému odrazu.

[3] Vztah 2.13 odvodil Paul S. Heckbert[10] a je uveden též v [8].

2.5.2 Ostatní osvětlovací modely

Rozšířený Phongův osvětlovací model sice dokáže zpracovat odraz a lom, to ale k dosažení fotorealističnosti nestačí. Postupem času se začaly objevovat osvětlovací modely, které pracovaly s dalšími fyzikálními vlastnostmi světla. Například Hallův osvětlovací model[8] pracuje i s vlnovou povahou světla (využívá Fresnelových rovnic), osvětlovací model Torrance-Sparrow[7] uvažuje povrch těles jako síť mikroplošek a je založen na dvousměrové odrazové distribuční funkci. Maximálního stupně realističnosti, kdy je výsledný obraz scény téměř k nerozeznání od reality, lze dosáhnout kombinací ray-tracingu s dalšími zobrazovacími metodami, jako jsou radiozita (vliv sekundárních světelných zdrojů) nebo photon mapping (zobrazení kaustik), to však vyžaduje víceprůchodové zpracování scény[7].

Obrázek 2.7: *Ukázka osvětlovacího modelu použitého v programu POV-Ray. Scéna na obrázku byla renderována třemi průchody a kromě ray-tracingu byly při jejím vytváření použity metody radiozita a photon mapping. Obrázek pochází z* `<http://hof.povray.org/>`.

2.6 Optimalizace metody ray-tracing

Nevýhodou metody ray-tracing, jak už bylo naznačeno v úvodu, je její výpočetní a tím pádem i časová náročnost. Zobrazení komplexní scény (například té na obrázku 2.7) tak může klidně trvat i několik desítek hodin. Jakékoliv urychlení této metody je tedy vítáno.

James Arvo a David Kirk[8] rozdělili urychlovací metody použitelné v ray-tracingu do třech kategorií:

- rychlejší nalezení průsečíku paprsku s tělesy ve scéně,
- použití menšího počtu paprsků,
- jiný pohled na paprsek — paprsek není chápán jako polopřímka o nekonečně malé tloušťce, ale jako ohraničený útvar v prostoru, nejčastěji jehlan nebo kužel.

První kategorii Arvo a Kirk dále dělí na *rychlejší určení průsečíku paprsku s tělesem* a *snížení počtu těles, vůči kterým má být paprsek testován, aby byl nalezen průsečík.*

Do první podkategorie spadá například metoda obálek kolem těles (object bounding volumes) nebo předčasné ukončení výpočtu průsečíku, bylo-li během něj zjištěno, že paprsek těleso mine dříve než dojde na náročnou část výpočtu (příkladem může být krok 2 algoritmu 2.1).

Druhá podkategorie zahrnuje metody dělení prostoru scény na podprostory. Sem lze zařadit například oktalové stromy, kD-stromy nebo hierarchii obálek.

Menšího počtu paprsků lze dosáhnout adaptivním řízením hloubky rekurze či použitím adaptivního antialiasingu.

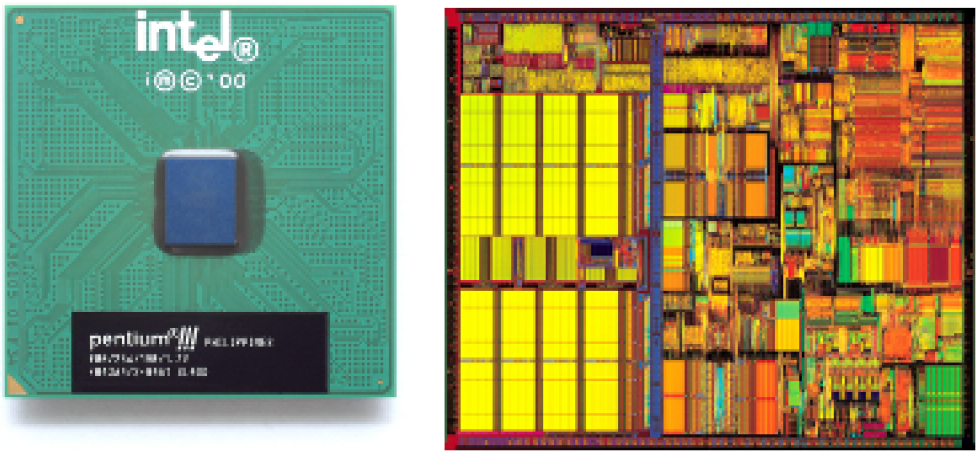
Konečně do třetí kategorie lze zařadit metodu sledování většího počtu paprsků ve svazku (*beam tracing*) nebo metodu *cone tracing*[4].

[4]Tato metoda dává dobré výsledky při antialiasingu, jelikož průsečíkem kuželu a tělesa není bod, ale kruh nebo elipsa. Bohužel, kvůli netriviálním výpočtům při určování nového počátku a rozptylového úhlu kuželu byla tato metoda téměř zapomenuta[8].

Kapitola 3

Instrukční sada SSE

Tato instrukční sada se poprvé objevila roku 1999 u procesoru Intel Pentium III jako odpověď na rozšíření 3DNow! od konkurenční firmy AMD. Zkratka *SSE* v názvu kapitoly znamená *Streaming SIMD*[1] *Extensions*[11] a jedná se tedy o sadu vektorových[2] instrukcí.



Obrázek 3.1: *Procesor Intel Pentium III který jako první podporoval instrukční sadu SSE. Obrázky pochází z* `<http://en.wikipedia.org/wiki/Pentium_III>` *(vlevo) a z* `<http://www.tayloredge.com/museum/processor/processorhistory.html>` *(vpravo).*

3.1 Předchůdci SSE — instrukční sady MMX a 3DNow!

SSE instrukce však nebyly prvními vektorovými instrukcemi, které na svých procesorech firma Intel zavedla. Těmi byly instrukce MMX[3]. Instrukční sada MMX obsahovala pouze instrukce používající celočíselnou aritmetiku a mezi její hlavní nedostatky patřilo používání 80bitových registrů `ST0` až `ST7` matematického koprocesoru[4] jako svých 64bitových registrů

[1] Význam zkratky *SIMD* je uveden v kapitole 1 jako poznámka pod čarou číslo 6.

[2] Pojmenování „vektorová instrukce“ a „SIMD instrukce“ jsou významově ekvivalentní.

[3] Názvu *MMX* není oficiálně přiřazen žádný význam, jedná se o obchodní název společnosti Intel[11]. Běžně je však název *MMX* považován za zkratku od *MultiMedia eXtension*.

[4] Též *x87 Floating-Point Unit*. Dříve se vyskytoval jako samostatný čip, nyní je běžnou součástí procesoru.

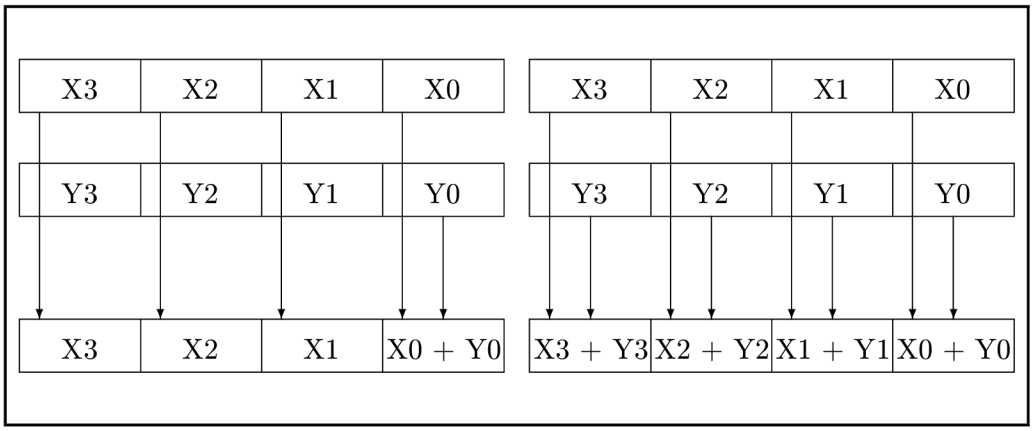
`MM0` až `MM7`. To znemožňovalo současné použití MMX instrukcí a instrukcí matematického koprocesoru. Navíc po ukončení práce s MMX instrukcemi musel být koprocesor uveden do výchozího stavu speciální instrukcí `emms`. Výhodou instrukční sady MMX naopak byla podpora saturační aritmetiky, takže mohla být efektivně použita při různých operacích s rastrovými obrazy (například skládání obrazů s nastavenou transparentní složkou přes sebe).

Nedostatků instrukční sady MMX využila firma AMD uvedením procesoru K6-2, který podporoval instrukční sadu 3DNow!. Instrukce 3DNow![1] používaly, stejně jako MMX, 64bitové registry namapované do registrů matematického koprocesoru. Narozdíl od MMX však umožňovaly pracovat i s reálnými čísly a to se dvěmi hodnotami formátu IEEE 754 s jednoduchou přesností (dále jen *single*). Tím odpadlo přepínání mezi MMX módem a módem matematického koprocesoru, bylo-li nutné pracovat jak s celými tak i s reálnými čísly.

3.2 Charakteristika instrukční sady SSE

Hlavním rozdílem mezi instrukční sadou SSE a instrukčními sadami MMX či 3DNow! je 8 nových 128bitových registrů označených `XMM0` až `XMM7` — do jedonoho XMM registru se tedy vejdou 4 hodnoty typu single. Tyto registry jsou zcela nezávislé na registrech matematického koprocesoru a slouží pouze k provádění početních operací, nelze jimi adresovat paměť. Instrukce SSE jsou rozděleny do následujících kategorií[11]:

1. instrukce pro načítání a přesun dat
2. aritmetické instrukce
3. logické instrukce
4. instrukce pro porovnání dat
5. „prohazovací“ instrukce (*shuffle instructions*)
6. instrukce konverzí



Obrázek 3.2: *Rozdíl mezi skalární instrukcí* `addss` *(vlevo) a její vektorovou variantou* `addps` *(vpravo). Předloha obrázku pochází z* [11].

Instrukce pro načítání a přesun dat umožňují přesouvat data mezi registry a pamětí či mezi registry samotnými. Lze také pracovat pouze s částmi registrů, například instrukce `movlhps` přesune dolní polovinu obsahu zdrojového registru do horní poloviny cílového registru. Při přesunu mezi pamětí a XMM registrem je vyžadováno, aby adresa paměti byla zarovnána na 16bajtové hranici. Je-li tomu jinak, procesor generuje výjimku *General Protection*, což je stejná výjimka jako v případě neplatného přístupu do paměti. Jediná instrukce pro přesun, která umožňuje pracovat s nezarovnanými daty, je `movups`.

Aritmetické instrukce provádí výpočty nad daty uloženými v XMM registrech a to hned ve dvou variantách — vektorové a skalární (obrázek 3.2). SSE aritmetické instrukce pracují podstatně rychleji než jejich ekvivalenty v matematickém koprocesoru a navíc při výpočtech nepoužívají zásobníkový model, takže práce s nimi je o mnoho příjemnější. Bohužel však neobsahují tak široký repertoár matematických instrukcí jako je tomu u koprocesoru a proto musejí být složitější matematické funkce implementovány pomocí jednodušších instrukcí. Do této skupiny instrukcí spadají i instrukce `maxps`/`maxss` a `minps`/`minss`, s jejichž pomocí lze provádět saturační aritmetiku.

Logické instrukce se spolu s instrukcemi porovnání používají především k ovlivňování průběhu výpočtu. Pomocí prohazovacích instrukcí lze měnit pořadí datových složek v rámci jednoho vektoru nebo z vybraných datových složek dvou vektorů sestavit vektor nový. Tento typ instrukcí je vhodný především pro práci z daty uloženými ve formátu *AoS* (popis způsobů, jakými lze ukládat data určená ke zpracování SSE instrukcemi, bude uveden dále). Mezi tyto instrukce patří například `shufps`, `unpcklps` nebo `unpckhps`. Instrukcemi z poslední skupiny lze provádět konverze mezi celými a reálnými čísly, včetně zaokrouhlení nebo odseknutí desetinné části.

3.2.1 Novější verze instrukční sady SSE

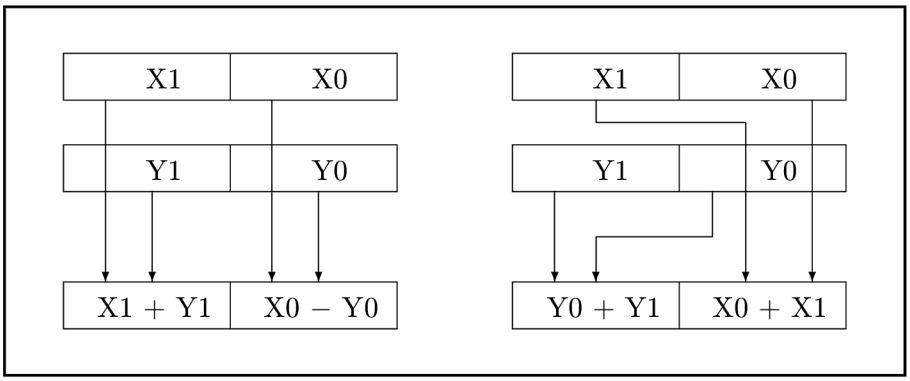
Následující verze instrukční sady SSE, označovaná jako SSE2, přinesla několik užitečných rozšíření. Byly přidány nové instrukce, které

- dokázaly pracovat s čísly formátu IEEE 754 s dvojitou přesností (dále jen *double*),
- uměly provádět s XMM registry celočíselné operace, včetně bitových posunů,
- prováděly celou řadu konverzí.

Instrukční sada SSE2 tak dokázala pracovat až s šesti datovými typy — jeden XMM registr mohl kromě čtyř hodnot typu single obsahovat také dvě hodnoty typu double, z celočíselných datových typů pak 16 slabik, 8 slov, 4 dvojslova nebo 2 čtyřslova. S příchodem této sady bylo možné provádět různé „triky“, které využívaly vlastností normy IEEE 754. Vektorové verze exponenciálních a logaritmických funkcí se tak daly naimplementovat snadněji než by tomu bylo u první verze SSE.

Instrukční sada SSE3 přinesla instrukce pro horizontální součet a rozdíl, které umožňovaly snadnější operace s vektory uloženými jako *pole struktur* (viz dále). Dalším přínosem této sady byly instrukce provádějící asymetrické aritmetické operace (obrázek 3.3). Následující rozšíření SSSE3 pak obohatilo instrukční sadu o celočíselné verze instrukcí pro horizontální součet a rozdíl.

Velmi užitečnými instrukcemi, které přišly spolu s verzí SSE4 (přesněji SSE4.1), byla dvojice instrukcí pro skalární součin dvou vektorů (instrukce `dpps` a `dppd`).



Obrázek 3.3: *Ukázka činnosti instrukce* `addsubpd`*, provádějící asymetrický výpočet (vlevo) a instrukce* `haddpd`*, provádějící horizontální součet (vpravo). Předloha obrázku pochází z* [11].

3.3 Programovací techniky

3.3.1 Podpora SSE instrukcí v jazycích `C` a `C++`

V jazycích `C` a `C++` lze SSE instrukce používat formou

- vloženého assembleru,
- tzv. *intrinsik.*

Prvním způsobem se zapisují SSE instrukce přímo do zdrojového kódu, pomocí `asm (`*řetězcový-literál*`);`[5], kde *řetězcový-literál* obsahuje posloupnost instrukcí, které jsou většinou v nezměněné formě zapsány překladačem na výstup určený pro další zpracování. Tento způsob má řadu nevýhod. Dialekt assembleru, v jakém musí být instrukce zapsány, závisí totiž na programu, který bude výstup překladače dále zpracovávat[6]. Má-li být tedy zdrojový kód přeložitelný na více typech překladačů jazyka `C` či `C++` (v rámci jedné architektury), musí pro části psané přímo v assembleru existovat pro každý dialekt jedna verze. Navíc nad kódem zapsaným přímo v assembleru překladač již většinou neprovádí žádné optimalizace.

Naproti tomu intrinsiky (*intrinsics*[12]) umožňují používat instrukce SSE bez nutnosti zapisovat je do kódu pomocí vloženého assembleru (ve formě intrinsik existují i ostatní SIMD instrukce). Použití intrinsik je v jazycích `C` a `C++` stejné jako použití funkcí. Výhodou intrinsik je, že s nimi překladač může provádět optimalizace a také přenositelnost kódu obsahujícího intrinsiky mezi různými překladači jazyků `C` a `C++` v rámci jedné architektury (za předpokladu, že dané překladače intrinsiky podporují).

3.3.2 Problém vektorizace

Vektorizace je převod kódu zpracovávajícího data sekvenčně na kód, který data zpracovává paralelně[12]. Vektorizovaný kód tak může být tolikrát rychlejší, kolik dat lze jednou vektorovou instrukcí naráz zpracovat. Nevýhodou vektorizace je, že není triviální. V některých případech tak může docházet i ke ztrátám na efektivitě.

[5] Tento způsob zápisu může být na různých překladačích odlišný, závisí na implementaci překladače[6].

[6] Některé překladače mohou podporovat i více dialektů assembleru. V překladači `gcc` lze instrukce assembleru psát jak v syntaxi *AT&T*, tak i v syntaxi *Intel*[9].

V současnosti dokáží vektorizaci vybraných algoritmů provádět některé překladače automaticky (např. Intel[12]). Je však mnohem lepší vektorizaci provádět ručně, jelikož ne všechny překladače umí správně odhalit záměr programátora.

3.3.3 Oranizace dat v paměti

V případě SSE a jiných vektorových instrukcí mohou být data v paměti uložena dvojím způsobem jako

- pole záznamů,
- záznam polí.

Pole záznamů (*Array of Structures — AoS*[12]) uvažuje data jako posloupnost vektorů, každý vektor je pak složen ze složek. Tento pohled na data je přirozený, avšak nepraktický. Například pokud by bylo nutné všechny složky takto uloženého vektoru sečíst, musely by být ještě před provedením této operace z vektoru vyjmuty a přeorganizovány do patřičné formy, což by znamenalo několik operací navíc. A co je další nevýhoda, výsledek takové operace by zabral pouze jednu složku vektoru, což je zbytečné plýtvání výkonem.

Naproti tomu záznam polí (*Structure of Arrays — SoA*), jak už název napovídá, uvažuje datové složky vektoru jako pole. To umožňuje jednodušší a efektivnější využití vektorových instrukcí. Pokud by se například do operandu vektorové instrukce vešly čtyři datové složky, pak s pomocí tohoto uspořádání dat lze sečíst složky čtyř vektorů a výsledkem je vektor čtyř součtů. Tento způsob ukládání dat je tedy pro vektorizaci nejvýhodnější a je také doporučován v [12]. Ovšem při používání tohoto modelu uložení dat mohou nastat i problémy, jak bude uvedeno v kapitolách 4 a 5.

Výpis 3.1: *Rozdíl mezi polem záznamů a záznamem polí.*

```
1  // Pole záznamů - Array of Structures (AoS):
2  typedef struct {
3    float x,y,z,w;
4  } VectorAoS;
5  VectorAoS vecs1[4];
6
7  // Záznam polí - Structure of Arrays (SoA):
8  typedef struct {
9    float x[4];
10   float y[4];
11   float z[4];
12   float w[4];
13 } VectorSoA;
14 VectorSoA vecs2;
```

3.3.4 Podmíněné přiřazení

U skalárních výpočtů probíhá podmíněné přiřazení způsobem, ve kterém se na základě vyhodnocení rozhodovacího výrazu vybere jedna ze dvou možných hodnot, která je následně přiřazena cílové proměnné (příkladem je ternární operátor v jazyce `C` či `C++`).

U vektorových výpočtů však vyhodnocením rozhodovacího výrazu není skalár, ale vektor booleovských hodnot, přesněji maska. Bylo-li porovnání dvou hodnot odpovídajících si složek různých vektorů pravdivé, jsou bity na patřičných pozicích v masce nastaveny na `1`,

v opačném případě na 0. Pomocí masky a logických instrukcí lze pak vybrat ze vstupních vektorů ty správné složky a vytvořit z nich výsledný vektor.

Výpis 3.2: *Podmíněné přiřazení zapsané pomocí SSE intrinsik.*

```
1 __m128 mvec1 = _mm_set_ps(1.0f,-1.0f,-1.0f,1.0f);
2 // mmask = [~0,0,0,~0]
3 __m128 mmask = _mm_cmple_ps(_mm_setzero_ps(),mvec1);
4 // result = [2,0,0,2] | [0,3,3,0] = [2,3,3,2]
5 __m128 result = _mm_or_ps(
6   _mm_and_ps(mmask,_mm_set1_ps(2.0f)),
7   _mm_andnot_ps(mmask,_mm_set1_ps(3.0f))
8 );
```

Za zmínku stojí také instrukce `movmskps`, případně její odpovídající intrinsika `_mm_movemask_ps`. Tato instrukce extrahuje znaménkové bity ze všech čtyř složek vektoru, vytvoří z nich čtyřbitovou masku a uloží ji do cíle. Tímto způsobem lze přeskočit velmi složitý výpočet, bylo-li zjištěno, že všechny čtyři složky vektoru nevyhovují vstupním podmínkám. Je však dobré nejprve vyzkoušet, zda-li je kratší výpočet s neplatnými daty nebo provedení instrukce skoku.

Výpis 3.3: *Využití* `_mm_movemask_ps`.

```
1 if (_mm_movemask_ps(_mm_cmpge_ps(vec1,_mm_setzero_ps())) != 0) {
2   // Zde se provede výpočet, byla-li alespoň jedna složka
3   // vektoru 'vec1' kladná.
4 }
```

3.3.5 Dělení a reciproké instrukce

Stejně jako u instrukcí matematického koprocesoru, tak i u SSE instrukcí je výpočet podílu dvou čísel (použití instrukce `divps`) časově náročná operace. Instrukční sada SSE však nabízí několik možností. Lze použít[12]

- reciproké instrukce, není-li na přesnost výsledku brán velký zřetel,
- reciproké instrukce zpřesněné jedním krokem Newton-Raphsonovy metody, má-li být výsledek uspokojivý,
- instrukci dělení, má-li být výsledek co nejpřesnější.

Reciproké instrukce jsou v kombinaci s násobením rychlejší než použití instrukce dělení, ale také nejméně přesné, jelikož jejich výsledek je v hardwaru procesoru odhadnut. Zpřesnění pomocí Newton-Raphsonovy metody je znázorněno v algoritmu 3.1[7].

Algoritmus 3.1. Newton-Raphsonovova metoda. Je dána funkce $y = g(a)$ a k ní odpovídající funkce $f(x_i) = g^{-1}(x_i) - a$. Dále je znám počáteční odhad x_0 řešení rovnice $g(a) - y = 0$ a hodnota ε.

1. $i = 0$.
2. $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$.
3. Je-li $|x_{i+1} - x_i| < \varepsilon$ je výpočet ukončen.
4. Jinak $i = i + 1$ a pokračuje se krokem 2.

V SSE existují dva druhy reciprokých instrukcí. Jeden pro výpočet $y = \frac{1}{x}$ (`rcpps`, `rcpss`) a druhý pro výpočet $y = \frac{1}{\sqrt{x}}$ (`rsqrtps`, `rsqrtss`). Vztah pro jeden krok Newton-Raphsonovy metody u `rcpps`, `rcpss` má tvar

$$x_1 = 2x_0 - ax_0^2, \tag{3.1}$$

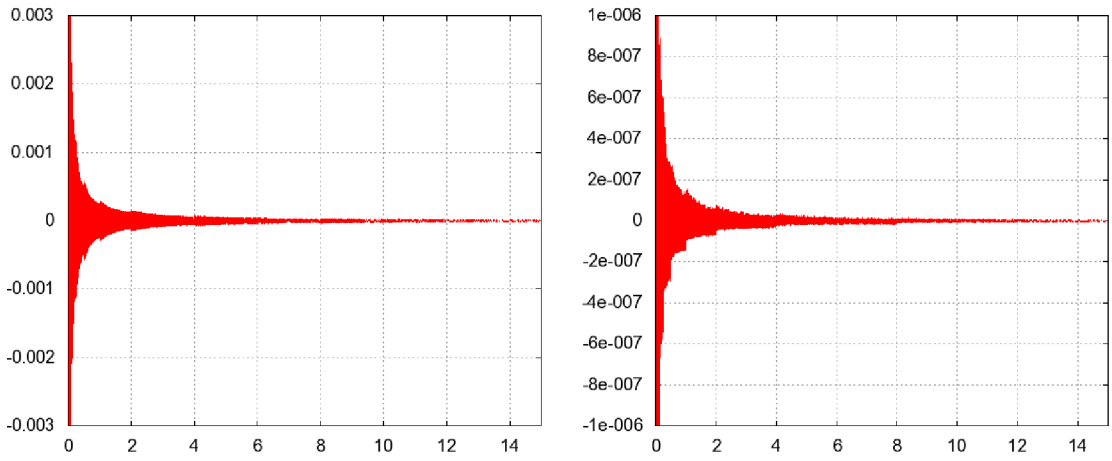
u instrukcí `rsqrtps`, `rsqrtss` je výpočet poněkud náročnější

$$x_1 = \frac{3}{2}x_0 - \frac{1}{2}ax_0^3. \tag{3.2}$$

Proto je lepší pro výpočet reciproké hodnoty druhé odmocniny vypočítat nejdříve druhou odmocninu, odhadnout její reciprokou hodnotu pomocí `rcpps` nebo `rcpss` a tu dále zpřesnit vztahem 3.1.

Výpis 3.4: *Výpočet reciproké hodnoty druhé odmocniny pomocí SSE intrinsik.*

```
__m128 a = _mm_sqrt_ps(_mm_set_ps(1.0f,2.0f,3.0f,4.0f));
__m128 x0 = _mm_rcp_ps(a); // Počáteční odhad.
// Jeden krok Newton-Raphsonovy metody:
__m128 x1 = _mm_sub_ps(
  _mm_add_ps(x0,x0),_mm_mul_ps(a,_mm_mul_ps(x0,x0))
);
```



Obrázek 3.4: *Grafy absolutních chyb mezi hodnotami vypočtenými matematickým koprocesorem jako podíl $\frac{1}{x}$ a hodnotami vypočtenými SSE instrukcí* `rcpps` *bez zpřesnění (vlevo) a se zpřesněním pomocí jednoho kroku Newton-Raphsonovy metody (vpravo). Na ose x jsou vyznačeny hodnoty vstupů výpočtu, na ose y pak k nim příslušné hodnoty absolutních chyb. Grafy byly vytvořeny v rámci experimentování s SSE instrukcemi.*

Kapitola 4

Návrh a implementace

4.1 Možnosti použití SSE instrukcí v ray-tracingu

Při optimalizaci ray-tracingu lze SSE instrukce použít několika způsoby. Tím nejjednodušším je použití skalárních variant SSE instrukcí namísto instrukcí matematického koprocesoru. V současnosti může být tato optimalizace provedena přímo překladačem (v `gcc` ji lze nastavit parametrem `-mfpmath=sse`), programátor se tedy nemusí o nic starat. Tato optimalizace využívá výhod instrukcí SSE oproti koprocesoru (viz předchozí kapitola, část 3.2).

Druhou možností je využití instrukcí SSE ke zpracování dat ve formátu *AoS* (výpis 3.1). Tato úprava se datových struktur použitých v ray-tracingu příliš nedotkne — například strukturu reprezentující bod či vektor v prostoru lze vměstnat do jednoho `XMM` registru. Nevýhodou je však poněkud obtížnější manipulace s takto uloženými daty, jak bylo uvedeno v předchozí kapitole. Při této optimalizaci se stále pracuje s každým paprskem jednotlivě.

Poslední možností je použít uspořádání dat ve formátu *SoA*. Pokud veškeré aritmetické operace probíhají s hodnotami formátu *single*, pak lze s pomocí SSE instrukcí pracovat až se čtyřmi paprsky zároveň uloženými v jednom svazku. To s sebou přináší na jednu stranu výhody ve formě plného využití kapacity XMM registrů[1], na stranu druhou je však potřeba provést vektorizaci algoritmů použitých v ray-tracingu a také je důležité vyřešit rozpad svazku, bylo-li zasaženo několik těles najednou.

Při realizaci programové části této práce byla zvolena právě poslední z výše uvedených možností.

4.2 Existující implementace ray-traycingu využívající SSE instrukce

Většina implementací ray-tracingu využívající SSE instrukce pracuje s rovinnými objekty, především s trojúhelníky. Navíc se při hledání nejbližšího průsečíku používá optimalizace v podobě reprezentace scény formou vhodné datové struktury, většinou kD-stromu. Příkladem takové implementace je již zmiňovaná knihovna *OpenRT*, jejíž jádro[18] je postaveno na optimalizovaném výpočtu průsečíku paprsku s trojůhelníkem a optimalizovaném kD-stromu.

Další zajímavá implementace pochází od Alexandra Reshetova[16]. Reshetov, stejně jako Wald, pracuje pouze s trojúhelníky. V jeho algoritmu je svazek paprsků nejprve ohraničen

[1] V nejlepším možném případě. Dojde-li k úplnému rozpadu svazku, dojde také ke ztrátám na efektivitě.

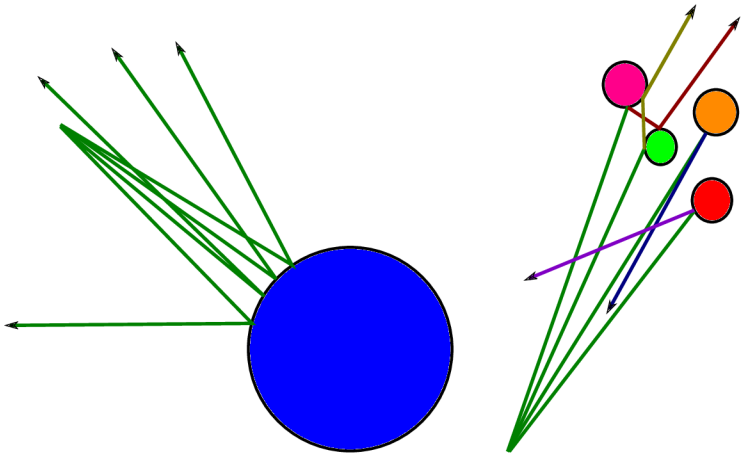
hranolem. Následně je testována poloha čelní stěny hranolu vůči trojúhelníku. Pokud leží mimo trojúhelník, svazek trojúhelník míjí. Pokud leží celá uvnitř trojúhelníku, všechny paprsky ve svazku trojúhelník protínají. K tomuto testu jsou zapotřebí pouze čtyři rohové paprsky, přitom ve svazku jich může být mnohem více (Reshetov použil svazek o 64 paprscích). Jestliže při testu nastala situace, kdy se čelní stěna hranolu a trojúhelník vzájemně překrývají, je poloha trojúhelníku vůči svazku testována postupně pro každý paprsek jednotlivě (přesněji, jsou testovány naráz po čtveřicích, v závislosti na šířce SSE registru).

Alexander Reshetov se též podílel na vývoji *víceúrovňového ray-tracingu*[17]. Zde je použito značného množství paprsků, z nichž největší význam mají rohové paprsky tělesa ohraničujícího svazek (stejně jako tomu bylo i u [16]). Algoritmus použitý v této implementaci se snaží najít společný vstupní bod do kD-stromu pro všechny paprsky ve svazku a tak minimalizovat počet nadbytečných operací při hledání průsečíků s tělesy.

Použití svazku paprsků spolu s prvotním algoritmem ray-tracingu, který vyvinul Whitted, je popsán v [5]. Zde jsou také popsána kritéria podle kterých mohou být svazky paprsků štěpeny na podsvazky (například podle typu objektu, podle materiálu povrchu objektu aj.).

4.3 Řešení rozpadu svazku paprsků

Při sledování svazku paprsků v komplexních scénách, obsahujících odrazivé a transparentí povrchy, může dojít k rozpadu svazku až na jednotlivé paprsky, jak je znázorněno na obrázku 4.1. Wald se o tomto problému zmínil v [18]. V jeho implementaci byly paprsky ve



Obrázek 4.1: *Ukázka rozpadu svazku paprsků. Obrázek vlevo znázorňuje nejlepší situaci, při které všechny paprsky ve svazku zasáhnou jedno těleso. I když se paprsky po odrazu od povrchu tělesa rozeběhnou do prostoru scény, lze s nimi i nadále pracovat v rámci jednoho svazku. Naproti tomu svazek v obrázku napravo se rozpadne na jednotlivé paprsky, se kterými lze pak pracovat už jen samostatně. Jedná se o nejhorší případ, při kterém je v rámci SSE instrukcí využit pouze zlomek z celkového výkonu.*

svazku postupně deaktivovány při průchodu kD-stromem. Wald testoval rozpad paprsků se svazky o rozměrech 2×2, 4×4 a 8×8 na scénách obsahujících 800, 34 000 a 240 000

trojúhelníků. Výsledky uvedl jako počet nadbytečných průchodů kD-stromem neaktivními paprsky v procentech (nejlepší výsledek byl 1,4 %, nejhorší 28,2 %).

Řešení rozpadu, které se snaží vytěžit z výkonu poskytovaného vektorovými instrukcemi co nejvíce, je uvedeno v [19]. Paprsky, které byly při průchodu scénou ze svazku vyřazeny, jsou nahrazeny paprsky jinými. Toto přeuspořádávání paprsků za chodu umožňuje lepší využití SIMD instrukcí. Uváděný algoritmus byl však testován pouze simulací a navíc předpokládá podporu pro operace rozložení a složení (*scatter*/*gather*), která se u SSE instrukcí zatím nevyskytuje.

Implementace rozpadu svazku paprsků použitá v praktické části této práce se řídí myšlenkou uvedenou v [5]. Jako kritérium rozpadu svazku je vybrán stav, při kterém svazek zasáhne dvě nebo více těles (obrázek 4.1). Maximální počet těles, které mohou být svazkem zasaženy, je tedy roven počtu aktivních paprsků v tomto svazku. Vzhledem k tomu, že implementace hledání průsečíku paprsku s nejbližším tělesem je realizována „hrubou silou“ (tj. paprsek je testován vůči každému tělesu ve scéně), je proces vyřazování neaktivních paprsků poněkud složitější, než by tomu bylo například u kD-stromu. Jsou-li totiž u jednoho tělesa některé paprsky vyřazeny, mohou se u jiného opět stát aktivními, pokud je toto těleso blíže počátkům paprsků než tomu bylo u předchozího případu. Při průchodu kD-stromem stačí paprsky pouze vyřazovat[18].

4.3.1 Algoritmus rozpadu svazku na podsvazky

Při realizaci algoritmu rozpadu svazku paprsků bylo použito zásobníku o maximální kapacitě rovné počtu paprsků ve svazku. Vstupem algoritmu je svazek paprsků a prázdný zásobník, jeho výstupem je pak index prvku uloženého na vrcholu zásobníku a zčásti či zcela zaplněný zásobník. Počet prvků na zásobníku je roven indexu vrcholu zvětšenému o jedničku (prázdnému zásobníku tedy odpovídá hodnota indexu -1). Prvek uložený na zásobníku je struktura (v implementaci pojmenovaná `Hit`), která obsahuje

- vektor `t` vzdáleností průsečíků paprsků s tělesem od jejich počátků,
- masku paprsků `hitmask` kolidujících s tělesem (`~0` znamená zásah, `0` že jde mimo),
- typ zásahu `hitinfo` (zda-li paprsky zasáhly těleso zevnitř, zvenčí nebo vůbec),
- ukazatel `obj` na těleso, které bylo paprsky zasaženo.

Položka `hitmask` zde má význam jako předpočítaná data pro bitové operace. K určení, zda paprsky těleso zasáhly nebo minuly, stačí položka `hitinfo`.

Základní kostru algoritmu zapsaného v jazyce `C++` ukazuje výpis 4.1 (spolu s výpisem 4.2 se jedná i o ukázku vektorizace, v tomto případě algoritmu hledajícího průsečík s nejbližším tělesem ve scéně).

Výpis 4.1: *Základní kostra algoritmu rozpadu svazku čtyř paprsků.*

```
1 int Scene::NearestIntersection(const Beam &b,Hit hs[4])
2 {
3   __m128 t,hi,hm,x,filter;
4   int sp = -1; // Index vrcholu zásobníku
5                // (na začátku je zásobník prázdný)
6   // Pro každé těleso ve scéně:
7   for (Object *o = m_obj_first; o != 0; o = o->next) {
8     // - zjisti průsečíky svazku s daným tělesem
```

```
9       t = o->Intersection(b); hi = o->GetHitInfo();
10      // - inicializuj masku (~0 => zásah, 0 => mimo)
11      // - pokud alespoň jeden paprsek ze svazku těleso zasáhl, tak:
12      if (_mm_movemask_ps(hm = _mm_cmpneq_ps(hi,FP_ZERO))) {
13        // + paprsky, které těleso minuly, mají průsečík v nekonečnu
14        t = _mm_or_ps(_mm_and_ps(hm,t),_mm_andnot_ps(hm,FP_MAX.mmf));
15        // + projdi celý zásobník a aktualizuj jeho prvky
16        switch (sp) {
17          case 3:
18            // Kód pro plný zásobník
19          case 2:
20            // Kód pro 3 položky na zásobníku
21          case 1:
22            // Kód pro 2 položky na zásobníku
23          case 0:
24            // Kód pro 1 položku na zásobníku
25          case -1:
26            // Kód pro prázdný zásobník
27            break;
28          default:
29            break;
30        }
31      }
32    }
33    return sp;
34  }
```

Informace o pozici svazku paprsků vůči testovanému tělesu je uložena v proměnných `t`, `hi` a `hm` (jejich význam je podobný položkám struktury `Hit`: `t` odpovídá `t`, `hi` odpovídá `hitinfo` a `hm` odpovídá `hitmask`). Tyto proměnné se účastní aktualizačního procesu prvků zásobníku probíhajícího uvnitř konstrukce `switch`. Záměrně vynechané příkazy `break` ve větvích `case 3` až `case 0` konstrukce `switch` umožňují průchod zásobníkem od jeho vrcholu až na jeho dno.

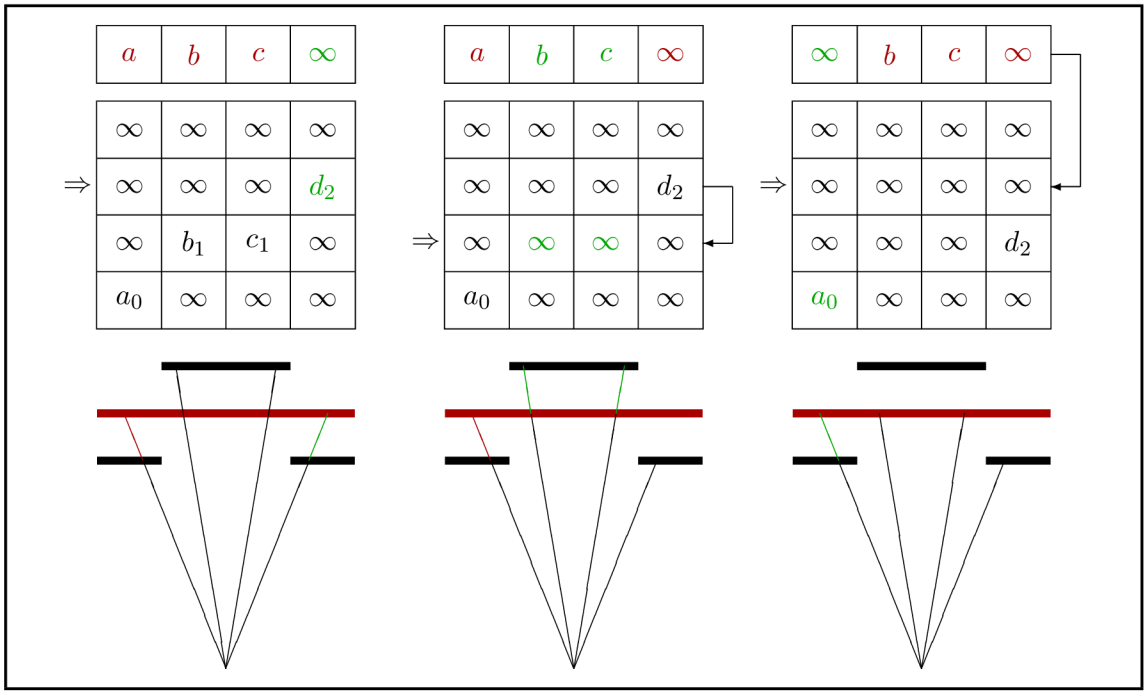
Kód nacházející se ve větvích `case 3` až `case 0` konstrukce `switch` je téměř totožný s kódem ve výpisu 4.2. Makro `N` použité v tomto výpisu může nabývat hodnot `3`, `2`, `1` nebo `0` (podle toho, ve které větvi `case` se právě nachází). Makru `STACK_LIMIT` je přiřazena hodnota `4`. Tato makra se ve skutečné implementaci nevyskytují, jsou zde pouze kvůli přehlednosti. V implementaci je tento kód zapsán v expandovaném tvaru.

Výpis 4.2: *Aktualizace/vyřazení položky na zásobníku.*

```
1  // Porovnej vzdálenosti průsečíků se vzdálenostmi průsečíků na
2  // zásobníku (~0 => jsou blíže, 0 => jsou dále nebo stejné):
3  filter = _mm_cmplt_ps(t,hs[N].t);
4  // Jsou-li všechny průsečíky dále (proběhl-li test s tělesem,
5  // jenž je zastíněno tělesem na zásobníku), nebude s touto
6  // položkou provedena žádná akce
7  if (_mm_movemask_ps(filter)) {
8    // Je-li alespoň jeden průsečík blíže, vyřaď z masky paprsků
9    // prvku na zásobníku starý paprsek
10   if (_mm_movemask_ps(x = _mm_andnot_ps(filter,hs[N].hitmask))) {
11     // Pokud po vyřazení nějaké paprsky prvku na zásobníku
12     // zůstaly, proveď aktualizaci i ostatních složek tohoto
13     // prvku a aktualizuj i proměnné ‘t‘, ‘hi‘ a ‘hm‘
```

```
14      hs[N].hitmask = x;
15      hm = _mm_and_ps(filter,hm);
16      hs[N].hitinfo = _mm_and_ps(hs[N].hitinfo,hs[N].hitmask);
17      hi = _mm_and_ps(hi,hm);
18      hs[N].t = _mm_or_ps(
19        _mm_andnot_ps(filter,hs[N].t),_mm_and_ps(filter,FP_MAX.mmf)
20      );
21      t = _mm_or_ps(
22        _mm_and_ps(filter,t),_mm_andnot_ps(filter,FP_MAX.mmf)
23      );
24    }
25    else {
26      // V případě vyřazení všech paprsků aktuálního prvku na
27      // zásobníku:
28 #if N < STACK_LIMIT-1
29      // - došlo-li k vyřazení hlouběji na zásobníku, pak vzniklé
30      //   prázdné místo obsaď prvkem z vrcholu zásobníku
31      if (sp > N)
32        hs[N] = hs[sp];
33 #endif
34      // - sniž index vrcholu zásobníku o jednu pozici
35      --sp;
36    }
37 }
```

Děje probíhající uvnitř konstrukce `switch` (výpisy 4.1 a 4.2) jsou pro lepší názornost demonstrovány na obrázku 4.2. Před vstupem do konstrukce `switch` má proměnná `sp` hodnotu 2



Obrázek 4.2: *Znázornění dějů probíhajících na zásobníku po vstupu do konstrukce* `switch`.

(na zásobníku jsou tedy uloženy tři prvky) a vektor `t` obsahuje složky (a, b, c, d). Nejprve

je vektor `t` porovnán (první obrázek zleva) s vektorem na vrcholu zásobníku (na obrázku vyznačen šipkou). Dochází k vyřazení složky d složkou d_2, neboť černé těleso vpravo je blíže než těleso červené (celá akce je vyznačena zeleně). Následuje kód ve větvi `case 1` (druhý obrázek zleva). Zde složky b a c vektoru `t` (a, b, c, ∞) vyřadí složky b_1 a c_1. Této situaci odpovídá zakrytí prostředního černého tělesa tělesem červeným. Na zásobníku tak vznikne prázdné místo, které je zaplněno položkou z jeho vrcholu. Proměnná `sp` je snížena o jedničku na hodnotu `1` a zásobník obsahuje nyní dva prvky.

Jako poslední je proveden kód ve větvích `case 0` a `case -1` (obě dvě situace jsou znázorněny na obrázku zcela vpravo). Ve větvi `case 0` je vyřazena složka a složkou a_0. Následuje vstup do větve `case -1`, kde je proměnná `sp` zvýšena na hodnotu `2` a na tuto pozici je uložen i vektor `t` (∞, b, c, ∞). Nakonec je vykonán příkaz `break`, čímž dojde k opuštění konstrukce `switch`. Konečný stav zásobníku je na obrázku 4.3. Poté, co je otestována poloha svazku pa-

∞	∞	∞	∞
∞	b	c	∞
∞	∞	∞	d_2
a_0	∞	∞	∞

Obrázek 4.3: *Konečný stav zásobníku z obrázku 4.2.*

prsků vůči všem tělesům ve scéně jsou informace uložené na zásobníku použity k vytvoření podsvazků.

Výpis 4.3: *Kód větve* `case -1`.

```
1 // Pokud je co a kam přidat, přidej:
2 if (_mm_movemask_ps(hm) && sp < 3) {
3   ++sp;
4   hs[sp].hitmask = hm;
5   hs[sp].hitinfo = hi;
6   hs[sp].t = t;
7   hs[sp].obj = o;
8 }
9 break;
```

4.4 Vlastní aplikace

Vlastní aplikace sestává ze dvou částí oddělených od sebe pomocí konstrukce preprocesoru `#ifdef-#else-#endif`. První část používá k veškerým matematickým operacím SSE instrukce, druhá instrukce matematického koprocesoru. Tento způsob umožňuje zkompilovat dvě verze programu, v závislosti na nastaveném parametru, a ty potom mezi sebou porovnávat.

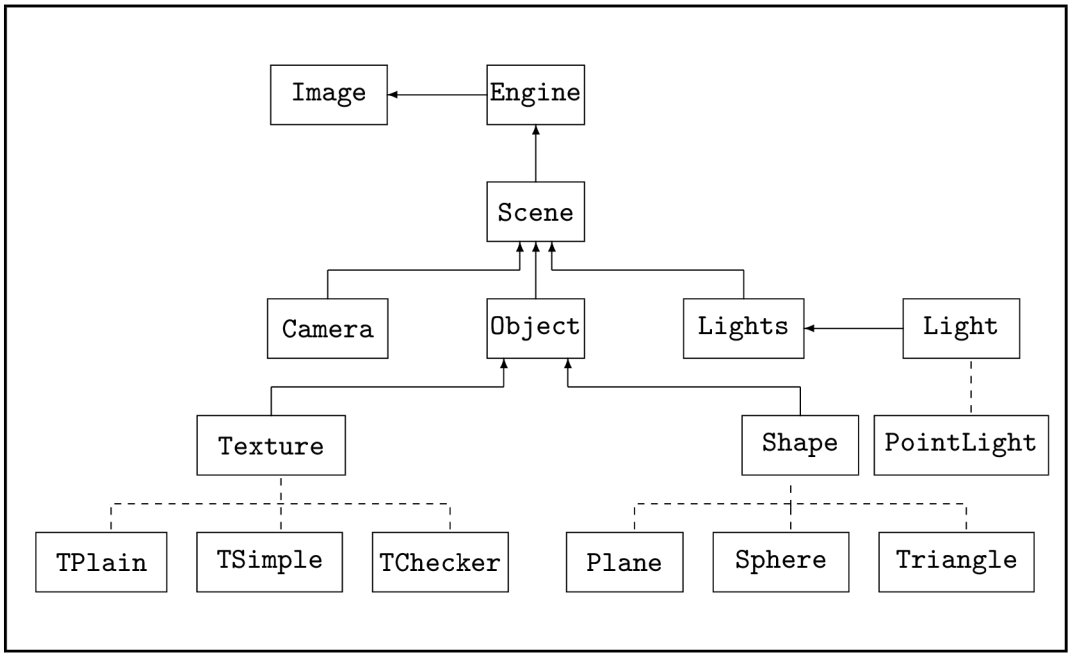
4.4.1 Volba programovacího jazyka a překladače

Vzhledem k výpočetní náročnosti metody ray-tracing a použití specifického rozšíření instrukční sady konkrétní architektury bylo nutné zvolit takový programovací jazyk, který je dostatečně výkonný a zároveň umožňuje využívat výhody dané architektury. Takovými jazyky jsou například assembler, `FORTRAN`, `C` a `C++`. První dva jazyky jsou sice výkonné, zato práce s nimi není nikterak snadná a kód pro ně napsaný je obtížně udržovatelný.

K implementaci programové části této práce byly tedy zvoleny jazyky `C` a `C++`. V jazyce `C` byla napsána matematická knihovna poskytující některé funkce, pro něž neexistují ekvivalentní SSE instrukce. Zbylá část aplikace byla napsána v `C++` a využívá tak výhod objektově orientovaného programování. Jako překladače byly zvoleny svobodný *GCC* a komerční *Microsoft Visual C++*.

4.4.2 Struktura aplikace

Aplikace vytvořená v rámci této práce vychází z programu *Aurelius*, který byl vytvořen Adamem Heroutem pro výukové účely do předmětu *Počítačová grafika* na Fakultě informačních technologí VUT v Brně. Kód Aurelia posloužil jako referenční model, vůči kterému byla porovnávána verze aplikace využívající SSE instrukce. Kromě práce s datovým typem `double`, který Aurelius běžně podporuje, byla přidána i podpora datového typu `float`. Ve verzi používající SSE instrukce pak byla provedena vektorizace většiny algoritmů a byly přidány nové datové struktury. Některé části Aurelia byly odstraněny (CSG, Perlinův šum, namísto válce byl použit trojůhelník) a byla přidána podpora lomu světla. Grafický výstup je ukládán ve formátu BMP. Na obrázku 4.4 jsou znázorněny vztahy mezi třídami v rámci



Obrázek 4.4: *Struktura aplikace.*

toku a zpracování dat. Černé šipky ukazují průtok informací při jejich zpracování, přerušované čáry pak představují dědičnost (odvozené třídy jsou umístěny níže než jejich rodiče). Obrázek neobsahuje všechny třídy použité v aplikaci, ale jen ty, které se účastní zpracování přenášených dat.

Jádro aplikace tvoří třída `Engine`, která obsahuje hlavní algoritmus ray-tracingu. Poté, co je ve funkci `main` vytvořena instance třídy `Scene` (do níž jsou následně uložena primitiva, osvětlení a nastavena kamera) je zavolána metoda `Engine::StartTracing`. Ta si vyžádá od instance třídy `Scene` instanci třídy `Camera`, pomocí které jsou generovány primární paprsky/svazky paprsků. Jakmile je primární paprsek/svazek paprsků vytvořen, je předán me-

todě `Engine::TraceRay/Engine::TraceBeam`, která volá rekurzivně sebe sama v závislosti na nastavené hloubce rekurze. Tato metoda je volána v rámci metody `Engine::StartTracing` pětkrát a na základě získaných barevných informací je proveden antialiasing. Výsledná barva je pak uložena voláním metody `Image::PutPixel/Image::PutFourPixels` do bufferu uvnitř instance třídy `Image`.

Metoda `Engine::TraceRay/Engine::TraceBeam` vytváří a sleduje sekundární paprsky/svazky paprsků a vyhodnocuje osvětlovací model v místě jejich dopadu. Tvorba podsvazků paprsků ze vstupního svazku je realizována v metodě `Engine::TraceBeam` voláním vektorizované verze metody `Scene::NearestIntersection`, jak bylo popsáno v podkapitole 4.3.1. Z každé položky, která byla na zásobník metodou `Scene::NearestIntersection` přidána, mohou být vytvořeny až dva nové svazky paprsků (odražený a lomený). Při renderování scén obsahujících mnoho malých těles z odrazivého nebo transparentního materiálu, kdy dojde k úplnému rozpadu svazků, tak na jeden paprsek může připadnout víc paměťového prostoru, než by tomu bylo u verze pracující pouze s jednotlivými paprsky.

Součástí třídy `Scene` je jednosměrně vázaný seznam objektů, který je při hledání nejbližšího tělesa procházen od začátku až do konce. Objekt je reprezentován třídou `Object`, v níž je uložena instance třídy odvozené od `Shape` a instance třídy odvozené od `Texture`. Ty charakterizují geometrické a optické vlastnosti tělesa.

4.4.3 Datové struktury — svazek paprsků

Podrobný popis datových struktur použitých v aplikaci se nachází v její programové dokumentaci. Na tomto místě bude zmíněna struktura uchovávající svazek paprsků. V momentě, kdy je vygenerován svazek primárních paprsků, mají tyto paprsky stejný počátek, jenž je shodný s pozicí kamery. K uložení svazku primárních paprsků by tedy stačilo uložit pouze jejich směry. Toto platí u perspektivní projekce. Pokud by se jednalo o projekci paralelní, byly by všechny směry primárních paprsků ve svazku shodné se směrem optické osy kamery. K uložení svazku by tak stačilo uložit jen počátky paprsků. V obou případech by struktury uchovávající svazek primárních paprsků měly stejnou velikost.

V ray-tracingu se však nevyskytují pouze paprsky primární, ale také sekundární. Ty mohou mít odlišné počátky a odlišné směry. Navíc sekundárních paprsků vzniká mnohem více než primárních. Z těchto důvodů byla pro uložení svazku paprsků použita struktura ve výpisu 4.4.

Výpis 4.4: *Struktura pro uložení svazku paprsků.*

```
1 struct Beam {
2   Point ps;            // Počátky paprsků (4)
3   Vector dirs;         // Směry paprsků (4)
4   __m128 rays_in_use; // Maska aktivních paprsků
5 };
```

Tato struktura používá formát *SoA* (viz podkapitola 3.3.3) a může obsahovat až 4 paprsky. Za předpokladu, že se svazek paprsků hned nerozpadne, je práce s takto uloženými paprsky efektivní. Jednotlivé složky čtveřice paprsků lze celé načíst do SSE registrů a v tomto stavu s nimi i pracovat (lze například načíst do SSE registrů složky `x`, `y` a `z` čtveřice vektorů, provést normalizaci a výsledky uložit zpět do vektorů). Při úplném rozpadu svazku až na jednotlivé paprsky je však používáno jen 24 bajtů (složka `rays_in_use` ztrácí pro jeden paprsek ve svazku smysl) z celkových 112, tedy pouze 21 % z úplné velikosti struktury.

4.4.4 Emulace složitějších matematických funkcí SSE instrukcemi

Instrukcemi SSE lze pokrýt většinu výpočtů probíhajících při ray-tracingu, hlavně ty, které se týkají určování vzájemné polohy paprsku a tělesa. Existují však i situace, kdy je třeba použít operaci, pro kterou neexistuje ekvivalentní SSE instrukce. Především se jedná o výpočet funkce x^y, použité ve Phongově osvětlovacím modelu, nebo funkce e^x, která je hlavní součástí vztahu pro výpočet útlumu světla při průchodu transparentním materiálem[4]. Z tohoto důvodu bylo nutné implementovat matematickou knihovnu, která by výše uvedené funkce poskytovala.

V matematické knihovně se nachází výše uvedené funkce hned ve třech variantách, oddělené od sebe pomocí konsrukce preprocesoru umožňující podmíněný překlad. První varianta je přepis výše uvedených funkcí z knihovny *Intel Approximate Math Library*[2]. V ní jsou funkce zapsány formou vloženého assembleru a byly proto přepsány do tvaru používajícího intrinsiky. Druhá verze využívá aproximačních polynomů, jejichž koeficienty byly určeny Remezovým algoritmem[13], který je v podobě programu součástí knihovny Boost. Poslední verze používá aproximaci pomocí Taylorových polynomů. Při překladu knihovny je také umožněn výběr stupně aproximačního polynomu, který je používán druhou a třetí verzí. Různými kombinacemi parametrů[3] lze nakonfigurovat knihovnu tak, aby vyhovovala stanoveným potřebám v poměru rychlost/přesnost.

4.4.5 Implementace skalárního součinu dvou vektorů

Skalární součin dvou vektorů je nejčastěji probíhající operace při ray-tracingu. Na jeho implementaci tedy závisí výkon celé aplikace. Výpočet skalárního součinu je uveden ve vztahu 4.1 a k jeho provedení v trojrozměrném souřadném systému tedy stačí třikrát operace násobení a dvakrát operace sčítání.

$$a_x b_x + a_y b_y + a_z b_z \tag{4.1}$$

Implementaci skalárního součinu v prostředí SSE, která je použita v programové části této práce, znázorňuje výpis 4.5.

Výpis 4.5: *Skalární součin dvou vektorů v prostředí SSE.*

```
1 __m128 operator *(const Vector &a,const Vector &b)
2 {
3   return _mm_add_ps(
4     _mm_add_ps(_mm_mul_ps(a.x,b.x),_mm_mul_ps(a.y,b.y)),
5     _mm_mul_ps(a.z,b.z)
6   );
7 }
```

Implementace skalárního součinu ve výpisu 4.5 pracuje s vektory uloženými ve formátu *SoA* a naráz tak dokáže spočítat skalární součin až čtyř dvojic vektorů. S příchodem SSE4 lze skalární součin provést jedinou instrukcí (`dpps` nebo `dppd`), která však vyžaduje data ve formátu *AoS* a proto nebyla v aplikaci použita (data by se musela před a po jejím použití přeorganizovávat, což by vedlo ke zpomalení aplikace).

[2] Existuje jen ve formě archivu dostupného z http://www.intel.com/design/pentiumiii/devtools/AMaths.zip.

[3] Jejich popis je uveden v programové dokumentaci.

4.4.6 Zarovnání dynamicky alokovaných dat

Jak bylo psáno v podkapitole 3.2, instrukce SSE vyžadují aby data, se kterými mají pracovat, byla zarovnána v paměti na 16bajtové hranici. V rámci statických dat, dat uložených na zásobníku a položek uvnitř struktur se o zarovnání postará překladač. S dynamicky alokovanou pamětí je situace horší. Hranice, na které bývají data alokovaná dynamicky v paměti zarovnána, závisí na implementaci použitého paměťového alokátoru (ve většině případů jsou data alokována na 8bajtové hranici). Naštěstí jak překladač *GCC*, tak i *Microsoft Visual C++* poskytují prostředky pro alokaci dat zarovnaných na paměťové hranici, jejíž hodnota je mocninou čísla 2. Jedná se o funkce `_mm_malloc` a `_mm_free`. Pomocí nich pak byly implementovány operátory `new` a `delete`, které jsou součástí třídy `HeapAlign16`. Tato třída je bázová pro všechny třídy, jejichž instance jsou vytvářeny dynamicky v paměti za běhu aplikace a které obsahují data využívaná SSE instrukcemi. Tímto způsobem je zajištěno správné zarovnání dynamicky alokovaných dat v paměti.

4.4.7 Ladění aplikace

Pokud při implementaci aplikace pracující převážně s grafickými daty, se kterými navíc probíhá spousta matematických výpočtů, dojde k chybě, není snadné tuto chybu vyhledat a odstranit. Nejčastěji dochází k chybám při přepisování matematických vztahů či konstant (konstanty, obzvláště jedná-li se o čísla s dlouhým neperiodickým zápisem, je vhodné si nechat vygenerovat a do programu je poté jen zkopírovat). Následky takových chyb jsou pak nežádoucí jevy při zobrazování grafických dat.

Chyby lze při ladění aplikace vyhledat dvěma způsoby: pomocí debuggeru nebo vytvořením sady maker určených k výpisu různých informací a jejich přímou aplikací v předpokládaném místě vzniku chyby. Debuggery jsou aplikace určené přímo pro ladění programů. Lze jimi krokovat aplikaci po jednotlivých instrukcích, sledovat obsah proměnných nebo stanovit místa, na kterých se má běh aplikace zastavit (tzv. *breakpointy*). Použití debuggeru si však vyžaduje, aby aplikace byla zkompilována s ladicími informacemi. Navíc při ladění aplikací typu ray-tracer, kdy je potřeba sledovat obsah proměnných pouze na určitém místě a v určité době (např. při sledování konkrétního paprsku), může být použití debuggeru značně komplikované. Z tohoto důvodu byl pro ladění zvolen druhý způsob. Byla napsána sada maker, která umožňuje vypisovat informace o konkrétním sledovaném paprsku. Pomocí těchto maker byla například odhalena chyba, při které těleso nevrhalo na rovinu stín. Postupnou analýzou výpisu sledovaného paprsku (který byl vyslán směrem k rovině do míst, kde měl být stín vržený tělesem) tak bylo zjištěno zaměněné znaménko „$-$“ ve vztahu pro výpočet průsečíku paprsku s rovinou.

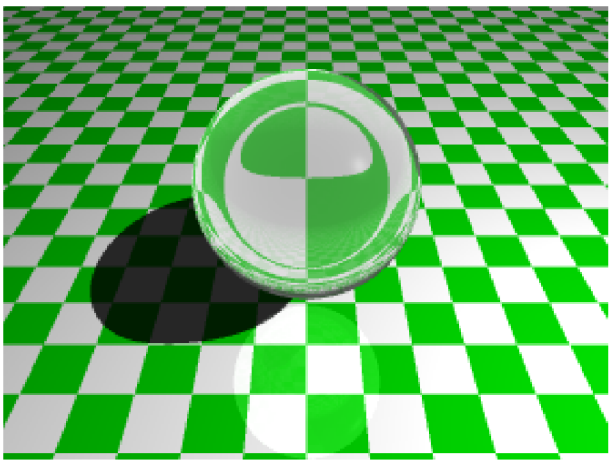
Kapitola 5

Průběh testů a dosažené výsledky

5.1 Použité testovací scény

5.1.1 Scéna 1: „*Whitted sphere*“

Předloha pro tuto scénu pochází z [20]. Jedná se o nejméně náročnou scénu použitou pro testování. Scéna obsahuje rovinu s texturou šachovnice, skleněnou kouli a jeden světelný zdroj.



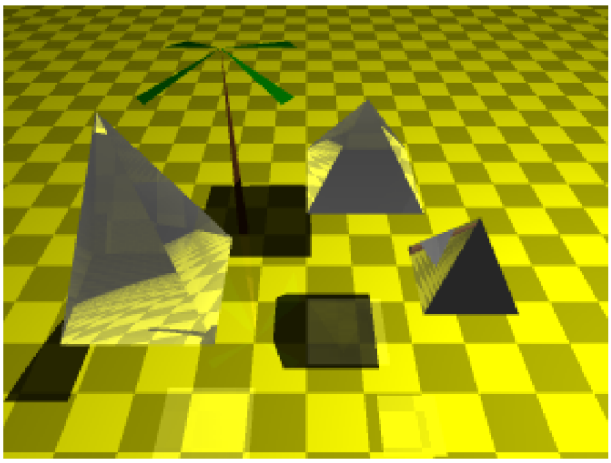
Obrázek 5.1: *Scéna 1: Whitted sphere*

5.1.2 Scéna 2: „*Giza*“

Náročnější scéna, sestavená pouze z rovinných objektů. Ve scéně se nachází tři vznášející se skleněné pyramidy a jedna palma. Dohromady scéna obsahuje jeden světelný zdroj, rovinu a 26 trojúhelníků.

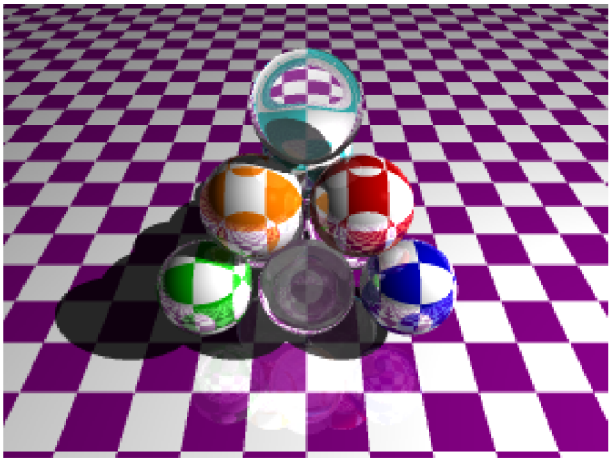
5.1.3 Scéna 3: „*Billiard*“

Poslední scéna se skládá z deseti koulí naskládaných na sebe do tvaru pyramidy. Je použito čtyř skleněných koulí a šesti koulí s texturou šachovnice. Objekty ve scéně byly uspořádány tak, aby došlo k brzkému rozpadu svazků. Účelem této scény je tedy zjistit jak rychle bude



Obrázek 5.2: *Scéna 2: Giza*

ray-tracing probíhat, bude-li ve svazku pouze jeden paprsek. Celá scéna je opět osvětlována pouze jedním světelným zdrojem.



Obrázek 5.3: *Scéna 3: Billiard*

5.2 Parametry a spouštění testů

Informace o prostředí, ve kterém byly provedeny všechny testy a seznam parametrů, se kterými byla aplikace zkompilována[1], jsou uvedeny v tabulce 5.1. Celkem bylo provedeno 540 testů se zaměřením na dobu renderování při zvětšující se hloubce rekurze.

Pod operačním systémem *Microsoft Windows* byly spouštěny sady testů na aplikacích zkompilovaných překladači GCC 3.4.5, GCC 4.4.0 a MSVC 9.0. Pod operačním systémem *GNU Linux* pak byly testovány aplikace přeložené překladačem GCC 4.3.2. Aplikace byly přeloženy ve verzích s datovými typy `float`, `double` a `__m128` (u verze používající instrukce

[1] V tabulce jsou uvedeny jen ty parametry, které by mohly nějakým způsobem ovlivnit výkon aplikace.

Operační systém 1	Microsoft Windows XP Home Edition Service Pack 3
Operační systém 2	Xubuntu Linux 8.10 Interpid Ibex, verze jádra 2.6.27-14-generic
Procesor	Intel Celeron M, 1,7 GHz
Velikost paměti RAM	960 MB
Požité překladače	GCC 3.4.5, GCC 4.4.0, GCC 4.3.2 (Linux), Microsoft Visual C++ 9.0 (15.00.210022.08)

Parametry překladu (GCC)	`-O3 -march=pentium-m -msse2`
	`-mpreferred-stack-boundary=4 -fno-rtti`
	`-fno-enforce-eh-specs -fno-exceptions`
	`-freg-struct-return -fno-stack-check`

Parametry překladu (MSVC)	`/Ox /arch:SSE2 /Qfast_transcendentals` `/GF /GL /GA /GS- /TP /EHs- /EHa- /GR-`

Tabulka 5.1: *Konfigurace prostředí a aplikace.*

SSE). Celkem tedy bylo v rámci testů spuštěno 9 aplikací pod operačním systémem *Microsoft Windows* a 3 aplikace pod operačním systémem *GNU Linux*. Aplikace využívající SSE instrukce používají pro zpřesnění výpočtu reciprokých instrukcí jeden krok Newton-Raphsonovy metody, dále pro výpočet funkcí `pow` a `exp` jsou použity aproximační polynomy generované Remezovým algoritmem (u funkce `pow` je pro $\log_2 x$ použit polynom 6. stupně a pro 2^x polynom 4. stupně, funkce `exp` je pak aproximována polynomem 4. stupně). Všechny scény byly renderovány v rozlišení 640×480. Jedna sada testů obsahuje renderování všech tří scén s hloubkou rekurze 1 až 15.

5.3 Výsledky testů

5.3.1 Scéna 1

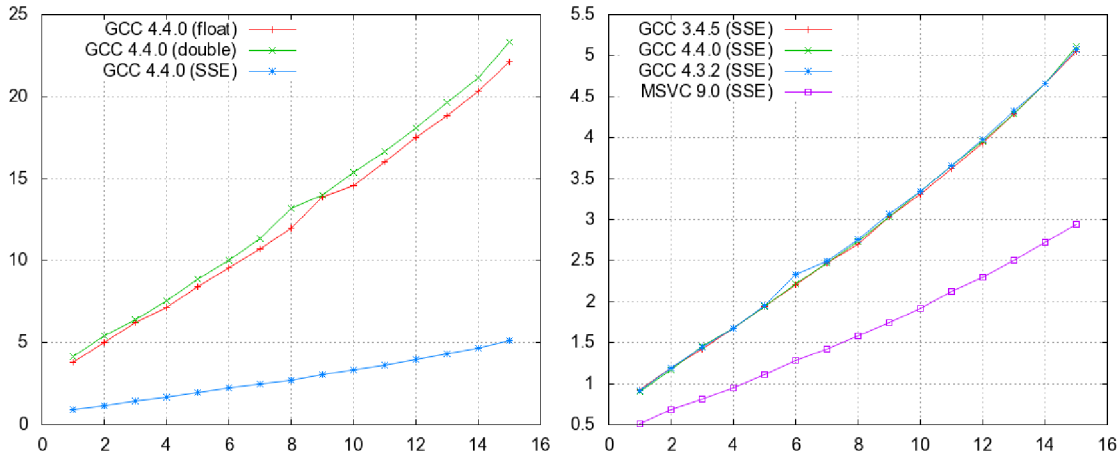
Na obrázku 5.4 jsou znázorněny výsledky testů scény „*Whitted sphere*“ formou dvou grafů pro překladač GCC 4.4.0 a pro všechny použité překladače v rámci výpočetního prostředí SSE. Výsledky všech testů jsou zaznamenány v tabulce A.1.

Na levém grafu obrázku 5.4 a z výsledků uvedených v tabulce A.1 je vidět značný výkonnostní rozdíl mezi aplikacemi používajícími k výpočtům pouze instrukce SSE (modrá křivka) a mezi aplikacemi používající instrukce koprocesoru. Použitím instrukcí SSE byla tato scéna renderována až 4krát rychleji v porovnání k verzi pracující pouze s datovým typem `float`. Toto zrychlení lze zdůvodnit použitím velmi malého počtu těles pro tuto scénu (rovina a koule). Na kouli dochází jak k odrazu paprsků, tak i k jejich lomu. Při odrazu od koule opouští většina odražených paprsků prostor scény a jejich sledování je ukončeno. Svazek paprsků tak zasáhne prakticky jen jedno těleso a tím pádem ho není nutné štěpit na podsvazky. V tomto stavu dochází k plnému využití SSE registrů. Při lomu se koule chová podobně jako spojná čočka — paprsky se lámou blíže k normálám na povrchu koule, čímž se sníží jejich rozbíhání.

Zajímavá situace nastane na okrajích koule. Zde může dojít k rozpadu svazku na dva podsvazky, kdy jeden zasáhne kostkovanou rovinu a druhý kouli. Navíc v tomto místě vstupují lomené paprsky do koule pod ostrým úhlem vzhledem k tečné rovině koule a tak dochází uvnitř koule k úplnému odrazu. Těchto okrajových paprsků však není mnoho, což

se také projeví mírným nárůstem renderovacího času při zvyšování hloubky rekurze.

Pravý graf na obrázku 5.4 ukazuje rychlost běhu aplikace zkompilované odlišnými překladači. Zatímco u aplikací zkompilovaných různými verzemi překladače GCC je doba běhu téměř stejná, u MSVC došlo až k 1,8násobnému zrychlení.



Obrázek 5.4: *Zobrazení výsledků testů pro scénu „Whitted sphere“. Hodnoty na ose x udávají hloubku rekurze, na ose y pak dobu renderování scény v sekundách. Levý graf znázorňuje vývoj rychlosti renderování v závislosti na použité hloubce rekurze pro aplikace zkompilované překladačem GCC 4.4.0 (MinGW) ve verzích pro výpočetní prostředí* `float`, `double` *a SSE. Pravý graf pak porovnává vliv použitých překladačů na výkon aplikace v rámci výpočetního prostředí SSE.*

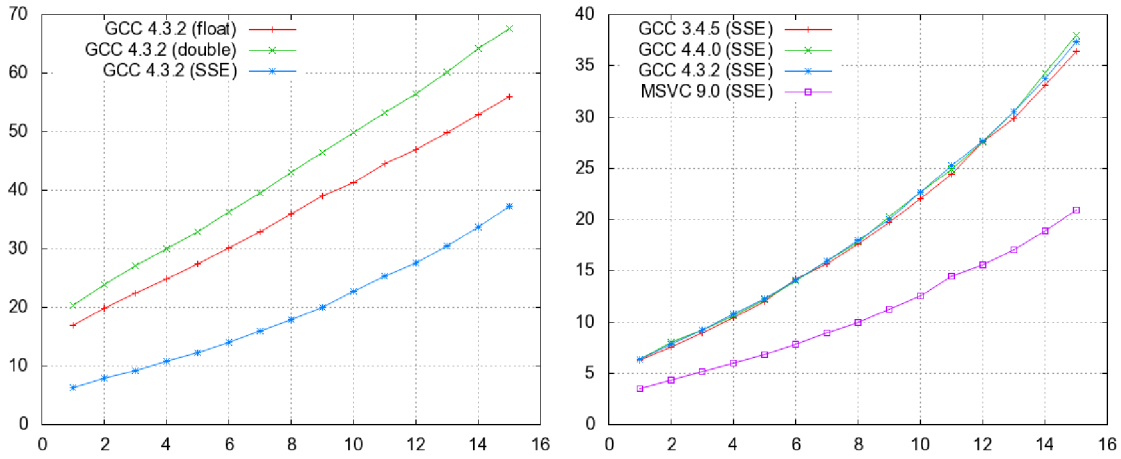
5.3.2 Scéna 2

U scény „*Giza*“ je již vliv rozpadu svazků na rychlost běhu aplikace znatelnější než tomu bylo u předchozí scény. Jak ukazuje levý graf na obrázku 5.5 a hodnoty v tabulce A.2, stále je verze využívající SSE instrukce rychlejší. Se zvyšující se hloubkou rekurze se však snižuje i výkonnostní rozdíl mezi verzí aplikace používající SSE instrukce a aplikací pracující pouze s typem `float` nebo `double`. Při hloubce rekurze 1 je verze s SSE oproti verzi s `float` téměř 2,7krát rychlejší, při hloubce rekurze 15 je to ale už jen 1,5krát. Důvodem je opět rozpad svazků na rozhraní dvou a více těles, kterých je více než u předchozí scény. K úplnému rozpadu svazků však nedojde (nebo dojde jen u zanedbatelného počtu případů), protože je scéna složena pouze z rovinných objektů, navíc vcelku rozměrných.

Pravý graf opět porovnává rychlosti renderování scény aplikacemi zkompilovanými na odlišných překladačích. Aplikace zkompilovaná pomocí MSVC 9.0 je i nadále nejrychlejší a to 1,8krát vůči aplikacím zkompilovaným pomocí GCC, tedy stejně jako i v minulém případě.

5.3.3 Scéna 3

Konečně zajímavé výsledky ukazuje obrázek 5.6 a tabulka A.3. V hloubce rekurze 1 (graf nalevo) je aplikace používající při ray-tracingu SSE instrukce 2,9krát rychlejší než aplikace pracující pouze s datovým typem `float`. Ke zlomu dochází mezi hloubkou rekurze 7



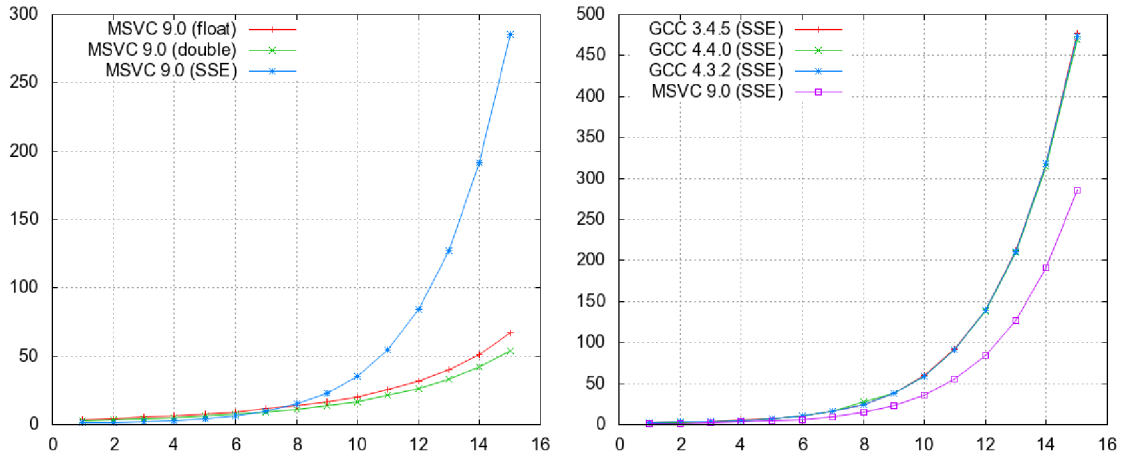
Obrázek 5.5: *Zobrazení výsledků testů pro scénu „Giza“. Hodnoty na ose x udávají hloubku rekurze, na ose y pak dobu renderování scény v sekundách. Levý graf znázorňuje vývoj rychlosti renderování v závislosti na použité hloubce rekurze pro aplikace zkompilované překladačem GCC 4.3.2 (Linux) ve verzích pro výpočetní prostředí* `float`, `double` *a SSE. Pravý graf pak porovnává vliv použitých překladačů na výkon aplikace v rámci výpočetního prostředí SSE.*

a 8. V hloubce rekurze 7 je SSE verze aplikace stále rychlejší, i když jen 1,2krát, než její `float` verze, avšak při úrovni zanoření 8 je již 1,1krát pomalejší. Od tohoto bodu dochází k prudkému nárůstu doby potřebné k renderování scény *„Billiard“* aplikací využívající SSE instrukce. Při spuštění aplikace s maximální úrovní zanoření 15 je SSE verze oproti `float` verzi až *4,3krát pomalejší* (aplikace zkompilovaná překladačem GCC tuto scénu renderovala téměř 8 minut, u MSVC trvalo renderování skoro 5 minut). Toto výrazné zpomalení lze zdůvodnit úplným rozpadem drtivé většiny všech svazků, které se dostaly do kontaktu s kulovými tělesy ve scéně. Část svazků je navíc „chycena do pasti“ průchodem skleněnou koulí v podstavě pyramidy, kde dochází k jejich rozpadu a mnohonásobným odrazům takto vzniklých jednotlivých paprsků od okolních koulí. Tím nastává nejhorší případ, kdy je kvůli jednomu paprsku ve svazku zbytečně manipulováno s rozsáhlými datovými strukturami a tím dojde k celkovému zpomalení renderování obrazu. Naštěstí uspokojivé kvality obrazu lze dosáhnout již při renderování s nastavenou hloubkou rekurze 4 a zde je verze používající svazky paprsků a SSE instrukce 2,2krát rychlejší oproti verzi pracující s `float`. Nejrychlejší byla i v tomto případě aplikace zkompilovaná překladačem MSVC a to 1,7krát.

5.4 Vliv způsobu výpočtu matematických funkcí pomocí SSE instrukcí na kvalitu obrazu

V této části budou provedeny dva testy. Na prvním testu bude znázorněno, jak se změní kvalita obrazu, použijí-li se v aplikaci různé implementace funkcí `pow` a `exp` pro SSE. Druhým testem pak bude ukázáno, co se stane vynechá-li se zpřesnění výsledku reciprokých instrukcí jedním krokem Newton-Raphsonovy metody.

Na obrázku 5.7 uprostřed se nachází výřez ze scény *Whitted sphere*, která byla vyrende-



Obrázek 5.6: *Zobrazení výsledků testů pro scénu „Billiard“. Hodnoty na ose x udávají hloubku rekurze, na ose y pak dobu renderování scény v sekundách. Levý graf znázorňuje vývoj rychlosti renderování v závislosti na použité hloubce rekurze pro aplikace zkompilované překladačem MSVC 9.0 ve verzích pro výpočetní prostředí* `float`, `double` *a SSE. Pravý graf pak porovnává vliv použitých překladačů na výkon aplikace v rámci výpočetního prostředí SSE.*

rována aplikací používající pro výpočet funkce `pow` aproximační polynomy s koeficienty získanými Remezovým algoritmem a výsledek reciprokých instrukcí zpřesňuje jedním krokem Newton-Raphsonovy metody. Obrázek vlevo používá pro výpočet funkce `pow` implementaci

Obrázek 5.7: *Vliv použitých výpočetních postupů na kvalitu obrazu u aplikace používající instrukce SSE.*

této funkce převzatou z *Intel Approximate Math Library*. Tato funkce neposkytuje příliš kvalitní výsledky, což se také projevilo absencí odlesku světelného zdroje na povrchu koule, ale zato její výpočet je nepatrně rychlejší. Nejrychleji probíhá výpočet funkce `pow` pomocí Taylorových aproximačních polynomů a to až o 50 %, přitom po vizuální stránce je kvalita obrazu stejná jako v případě použití aproximačních polynomů získaných Remezovým algoritmem (hlavním důvodem je však použití polynomů nižšího stupně). Na obrázku vpravo je pak vynechán jeden krok Newton-Raphsonovy metody pro zpřesnění výsledku reciprokých instrukcí. Nejenže kvalita takto získáného obrazu je otřesná, navíc došlo k citelnému prodloužení doby jeho renderování.

Kapitola 6

Závěr

Cílem této práce bylo navrhnout a implementovat optimalizaci metody ray-tracing využívající instrukce SSE a tu pak porovnat s neoptimalizovanou verzí. Jako druh optimalizace byl vybrán způsob sledování čtyř paprsků naráz ve svazku, aby bylo možné využít výkon nabízený SSE instrukcemi co nejefektivněji. Tento způsob si vyžádal vektorizaci téměř všech použitých algoritmů. Zvláštní pozornost pak byla věnována vektorizaci algoritmu řešícího kolizi svazku s tělesy ve scéně a nalezení nejbližších těles. Bylo nutné tak vyřešit rozpad svazků na podsvazky, v nejhorším případě až na jednotlivé paprsky. Tento úkol byl o to náročnější, jelikož referenční aplikace používala pro nalezení nejbližšího tělesa ve scéně algoritmus testu paprsku se všemi tělesy scény. Již vyřazené paprsky se tak mohly dostat opět do hry. Tato nepříjemnost byla v prostudované literatuře eliminována použitím některé z akceleračních datových struktur pro reprezentaci scény. Zde s ní však muselo být při implementaci počítáno. Dále bylo nutné vypořádat se s omezeným repertoárem instrukční sady SSE co do poskytování podpory pro vyšší matematické funkce. Tyto funkce byly implementovány pomocí instrukcí SSE pro elementární aritmetické operace a to hned v několika verzích s možností výběru a konfigurace konkrétní verze.

Provedené testy byly navrženy tak, aby pokryly všechny možné případy týkající se soudržnosti paprsků ve svazku. Byl otestován jak stav, kdy jsou všechny paprsky přítomny ve svazku, tak i stav kdy dochází k úplnému rozpadu svazků až na jednotlivé paprsky. Testováním bylo zjištěno, že zde implementovaná metoda je na komplexních scénách při nastavení velké hloubky rekurze několikrát pomalejší než její neoptimalizovaná varianta, což je důsledek rozpadu svazku paprsků a z toho plynoucí nárůst konzumace paměti na programovém zásobníku. Uspokojivé výsledky co do kvality obrazu však lze získat již použitím hloubky rekurze 3 nebo 4 a zde je verze optimalizovaná SSE instrukcemi v průměru dvakrát rychlejší. Z použitých překladačů si nejlépe vedl překladač *Microsoft Visual C++ 9.0*, který produkoval oproti všem použitým verzím překladače *GCC* téměř dvojnásobně rychlejší aplikace.

Jako pokračování projektu se nabízí použití SSE instrukcí s dalšími akceleračními strukturami, jako je například kD-strom a pokusit se tak implementovat ray-tracing v reálném čase. Dále je také možné zaměřit se na produkci více realistického výstupu použitím realističtějšího osvětlovacího modelu či přidáním podpory pro fotonové mapy a radiozitu, samozřejmě vše s využitím SSE instrukcí. Největší výzvou je však pokus o vyvinutí algoritmu, který by umožnil využití plného výkonu SSE instrukcí, aby nedocházelo k plýtvání výpočetní kapacity jako tomu bylo v případě, kdy ve svazku byl pouze jeden paprsek.

Literatura

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual : Application Programming* [online]. November 2009 [cit. 2010-04-15]. xxx, 304 s. Volume 1. Dostupný z WWW ve formátu PDF: http://support.amd.com/us/Processor_TechDocs/24592.pdf

[2] APPEL, Arthur. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring) : Proceedings of the April 30–May 2, 1968, spring joint computer conference*. New York (NY, USA) : Association for Computing Machinery, 1968, s. 37–45. Dostupný komerčně také z WWW (DOI): http://doi.acm.org/10.1145/1468075.1468082.

[3] BADOUEL, Didier. An Efficient Ray-Polygon Intersection. In GLASSNER, Andrew S. (editor). *Graphics Gems*. San Diego (CA, USA) : Academic Press, 1990, s. 390–393. ISBN 0-12-286166-3.

[4] BIKKER, Jacco. Raytracing : Theory & Implementation. Part 3, Refractions and Beer's Law. In *DevMaster* [online]. 2005-10-06 [cit. 2010-04-05]. Dostupný z WWW: http://www.devmaster.net/articles/raytracing_series/part3.php.

[5] BOULOS, Solomon et al. Packet-based Whitted and distribution ray tracing. In *Proceedings of Graphics Interface 2007*. New York (NY, USA) : Association for Computing Machinery, 2007, s. 177–184. Dostupný komerčně také z WWW (DOI): http://doi.acm.org/10.1145/1268517.1268547. ISBN 978-1-56881-337-0.

[6] British Standards Institute; STROUSTRUP, Bjarne (Foreword by). *The C++ Standard : Incorporating Technical Corrigendum No. 1*. 1st ed. Wiley, 2003. 782 s. ISBN 0-470-84674-7.

[7] FOLEY, James D. et al. *Computer graphics : principles and practice*. 2nd ed. in C. Boston (Mass., USA) : Addison-Wesley, 1996. xxiv, 1176 s. Addison-Wesley Systems Programming Series. ISBN 0-201-84840-6.

[8] GLASSNER, Andrew S. (ed.). *An introduction to ray tracing*. London (UK) : Academic Press, 1989. xiv, 330 s. ISBN 0-12-286160-4.

[9] GNU Compiler Collection. *GCC online documentation* [online]. Boston (MA, USA) : Free Software Foundation, 1988- , last modified 2010-04-14 [cit. 2010-04-20]. Dostupný z WWW: http://gcc.gnu.org/onlinedocs/.

[10] HECKBERT, Paul S.; Hanrahan, Pat. Beam tracing polygonal objects. In *SIGGRAPH '84 : Proceedings of the 11th annual conference on Computer graphics*

and interactive techniques. New York (NY, USA) : Association for Computing Machinery, 1984, s. 119–127. Dostupný komerčně také z WWW (DOI): http://doi.acm.org/10.1145/800031.808588. ISBN 0-89791-138-5.

[11] INTEL Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual : Basic Architecture* [online]. December 2009 [cit. 2010-04-15]. xxii, 478 s. Volume 1. Dostupný z WWW ve formátu PDF: http://www.intel.com/Assets/PDF/manual/253665.pdf.

[12] INTEL Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual* [online]. November 2009 [cit. 2010-04-15]. xxiv, 582 s. Dostupný z WWW ve formátu PDF: http://www.intel.com/Assets/PDF/manual/248966.pdf.

[13] MADDOCK, John et al. *Math Toolkit* [online]. 2006–2008, last revised on July 18, 2008 [cit. 2010-05-04]. Dostupný z WWW: http://www.boost.org/doc/libs/1_36_0/libs/math/doc/sf_and_dist/html/index.html. ISBN 0-9504833-2-X.

[14] PHONG, Bui Tuong. Illumination for computer generated pictures. *Communications of the ACM*. June 1975, vol. 18, issue 6, s. 311–317. Dostupný komerčně také z digitálního archivu ACM (DOI): http://doi.acm.org/10.1145/360825.360839. ISSN 0001-0782.

[15] Ray-triangle intersection. In *DmWiki* [online]. Last modified on April 24, 2008 [cit. 2010-04-08]. Dostupný z WWW: http://www.devmaster.net/wiki/Ray-triangle_intersection.

[16] RESHETOV, Alexander. Faster ray packets – triangle intersection through vertex culling. *Symposium on Interactive Ray Tracing*. 2007, vol. 0, s. 105–112. ISBN 978-1-4244-1629-5.

[17] RESHETOV, Alexander; SOUPIKOV, Alexei; HURLEY, Jim. Multi-level ray tracing algorithm. In *SIGGRAPH '05 : ACM SIGGRAPH 2005 Papers*. New York (NY, USA) : Association for Computing Machinery, 2005, s. 1176–1185. Dostupný komerčně také z digitálního archivu ACM (DOI): http://doi.acm.org/10.1145/1186822.1073329.

[18] WALD, Ingo. *Realtime Ray Tracing and Interactive Global Illumination*. Saarbrücken (Germany), 2004. xiv, 297 s. Disertační práce (PhD). Der Universität des Saarlandes, der Naturwissenschaftlich-Technischen Fakultät I, Computer Graphics Group. Dostupný také z WWW: http://www.sci.utah.edu/~wald/PhD/wald_phd.pdf.

[19] WALD, Ingo et al. *SIMD Ray Stream Tracing : SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering* [online]. August 2, 2007 [cit. 2010-04-28]. 9 s. Technical Report No UUSCI-2007-012. Scientific Computing and Imaging Institute, University of Utah. Elektronická kopie zprávy dostupná ve formátu PDF z WWW: http://www.sci.utah.edu/~wald/Publications/2007///Stream/download//sst.pdf.

[20] WHITTED, Turner. An improved illumination model for shaded display. *Communications of the ACM*. June 1980, vol. 23, issue 6, s. 343–349. Dostupný komerčně také z digitálního archivu ACM (DOI): http://doi.acm.org/10.1145/358876.358882. ISSN 0001-0782.

[21] WOOP, Sven; SCHMITTLER, Jörg; SLUSALLEK, Philipp. RPU : a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics*. July 2005, volume 24, issue 3, s. 434-444. Dostupný komerčně také z digitálního archivu ACM (DOI): http://doi.acm.org/10.1145/1073204.1073211. ISSN 0730-0301.

[22] ŽÁRA, Jiří aj. *Moderní počítačová grafika*. Vydání první. Brno : Computer Press, 2004. 612 s. ISBN 80-251-0454-0.

Příloha A

Obsah CD a tabulky s výsledky testů

A.1 Obsah CD

Součástí této práce je i CD, které obsahuje

- zdrojové texty technické zprávy této práce pro program LaTeX,
- zdrojové texty programové části této práce se všemi náležitostmi nutnými k překladu,
- přeložené programy spustitelné v operačním systému *Microsoft Windows*,
- programovou dokumentaci vygenerovanou programem *Doxygen*,
- text této práce ve formátu PDF (interaktivní forma i forma pro tisk),
- vyrenderované testovací scény v rozlišení 1280×800 a 1024×768,
- manuál popisující překlad a ovládání programu,
- plakát formátu A3.

A.2 Tabulky s výsledky všech testů

Na následujících stránkách jsou uvedeny výsledky všech 540 testů rozdělených do tří tabulek (pro každou testovací scénu jedna). Hodnoty výsledků udávají dobu renderování odpovídající scény v sekundách. Všechny testy byly spuštěny jednou, u vybraných hodnot pak vícekrát, pokud tyto hodnoty výrazně narušily tvar posloupnosti generované opakovaným spouštěním testované aplikace s postupně se zvyšující hloubkou rekurze.

Hloubka rekurze	`float`				`double`				SSE			
	GCC			MSVC	GCC			MSVC	GCC			MSVC
	3.4.5	4.4.0	4.3.2	9.0	3.4.5	4.4.0	4.3.2	9.0	3.4.5	4.4.0	4.3.2	9.0
1	3.578	3.828	3.350	2.093	3.796	4.141	3.520	1.750	0.922	0.906	0.920	0.515
2	4.734	5.015	4.390	2.765	4.984	5.390	4.550	2.296	1.187	1.171	1.190	0.687
3	5.718	6.218	5.290	3.265	6.015	6.406	5.510	2.734	1.421	1.453	1.440	0.812
4	6.921	7.171	6.220	3.890	7.046	7.562	6.500	3.234	1.671	1.671	1.670	0.953
5	8.203	8.437	7.270	4.578	8.265	8.843	7.550	3.781	1.937	1.937	1.950	1.109
6	9.046	9.578	8.250	5.171	9.390	10.046	8.570	4.359	2.203	2.218	2.330	1.281
7	10.156	10.718	9.250	5.750	11.140	11.328	09.630	4.906	2.468	2.468	2.490	1.421
8	11.406	11.968	10.010	6.453	11.656	13.187	10.710	5.453	2.703	2.734	2.760	1.578
9	12.593	13.875	11.110	7.203	12.921	14.000	11.830	6.140	3.031	3.031	3.070	1.750
10	13.843	14.593	12.540	7.937	14.171	15.375	12.970	6.734	3.312	3.343	3.350	1.921
11	15.171	16.031	12.430	8.531	15.406	16.625	14.100	7.312	3.625	3.656	3.660	2.125
12	16.406	17.500	15.110	9.312	16.765	18.078	15.270	7.937	3.937	3.953	3.980	2.296
13	17.781	18.812	16.000	10.359	18.046	19.625	16.550	9.171	4.296	4.296	4.320	2.500
14	19.640	20.312	17.410	10.953	19.437	21.140	16.510	9.234	4.656	4.656	4.660	2.718
15	20.671	22.125	18.420	11.750	21.640	23.312	19.080	9.906	5.046	5.110	5.070	2.937

Tabulka A.1: *Výsledky testů pro scénu „Whitted sphere“ (v sekundách).*

Hloubka rekurze	`float`				`double`				SSE			
	GCC			MSVC	GCC			MSVC	GCC			MSVC
	3.4.5	4.4.0	4.3.2	9.0	3.4.5	4.4.0	4.3.2	9.0	3.4.5	4.4.0	4.3.2	9.0
1	18.328	18.750	16.880	13.125	18.640	18.765	20.350	10.781	6.234	6.359	6.330	3.531
2	21.406	21.781	19.790	15.156	22.046	22.016	23.840	12.625	7.578	8.015	7.860	4.343
3	24.406	24.828	22.410	17.203	25.125	24.828	27.070	14.468	8.968	9.187	9.210	5.140
4	27.593	27.671	24.850	19.156	27.703	27.796	29.980	16.046	10.453	10.625	10.760	5.953
5	29.828	30.953	27.360	21.046	31.156	30.812	32.930	17.750	12.015	12.187	12.240	6.843
6	32.968	33.390	30.190	23.687	33.656	33.734	36.370	19.546	14.156	14.000	14.070	7.875
7	35.750	36.593	32.900	25.171	37.093	36.671	39.540	21.921	15.703	15.968	15.910	8.906
8	39.593	39.859	36.040	27.375	40.625	40.312	43.080	23.187	17.562	17.828	17.960	9.984
9	41.890	43.687	38.970	29.312	43.187	43.063	46.400	25.281	19.687	20.265	20.010	11.250
10	45.515	46.375	41.280	32.203	46.296	46.109	49.900	27.078	21.984	22.656	22.680	12.515
11	49.062	50.312	44.500	33.609	50.109	50.093	53.170	29.671	24.390	24.875	25.250	14.515
12	51.484	52.750	46.940	35.750	52.890	52.625	56.480	30.984	27.562	27.546	27.630	15.531
13	55.109	56.843	49.900	38.187	56.703	56.734	60.190	32.859	29.906	30.531	30.490	17.093
14	57.968	59.562	52.880	40.921	59.718	59.375	64.130	34.906	33.109	34.250	33.760	18.875
15	62.359	63.750	56.010	42.921	63.875	63.437	67.610	37.609	36.375	37.984	37.320	20.906

Tabulka A.2: *Výsledky testů pro scénu „Giza“ (v sekundách).*

Hloubka rekurze	`float`				`double`				SSE			
	GCC			MSVC	GCC			MSVC	GCC			MSVC
	3.4.5	4.4.0	4.3.2	9.0	3.4.5	4.4.0	4.3.2	9.0	3.4.5	4.4.0	4.3.2	9.0
1	6.343	6.843	6.040	3.703	6.671	7.109	6.210	3.031	2.250	2.218	2.280	1.296
2	7.578	8.171	7.260	4.437	7.968	8.468	7.520	3.640	2.890	2.875	2.910	1.656
3	9.015	9.546	8.580	5.296	9.406	10.015	8.870	4.359	3.765	3.718	3.790	2.171
4	10.671	11.312	10.090	6.343	11.062	11.781	10.430	5.171	5.640	5.000	5.070	2.921
5	12.671	13.406	11.960	7.640	13.109	13.984	12.350	6.281	7.140	7.046	7.160	4.156
6	15.156	16.140	14.280	9.140	15.781	16.718	14.760	7.546	10.546	10.375	10.750	6.187
7	18.125	19.390	17.180	11.703	18.859	20.046	17.710	9.171	16.015	15.796	16.030	9.437
8	22.609	23.375	20.760	13.500	23.359	24.828	21.380	11.234	24.640	27.828	24.670	15.265
9	26.796	28.984	25.250	16.609	27.953	29.375	26.000	13.843	38.203	37.703	38.320	22.812
10	33.015	34.875	31.550	20.328	34.281	36.312	31.990	16.921	59.859	58.250	59.310	35.500
11	40.781	43.203	38.280	25.484	42.359	45.390	39.560	21.671	91.765	90.485	91.250	54.734
12	51.625	54.593	47.830	31.718	52.875	55.859	49.110	26.453	139.781	137.734	139.370	84.250
13	64.156	67.312	59.820	40.328	65.953	70.140	61.660	33.281	212.187	209.125	211.070	127.140
14	81.421	85.406	75.660	50.906	83.859	88.250	77.680	42.296	318.500	314.343	317.530	191.500
15	103.250	108.203	95.340	67.000	105.734	112.187	98.950	54.109	477.078	469.421	474.010	285.171

Tabulka A.3: *Výsledky testů pro scénu „Billiard“ (v sekundách).*