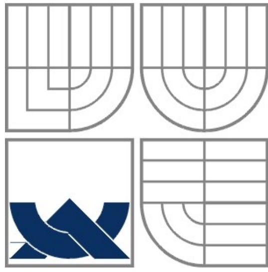
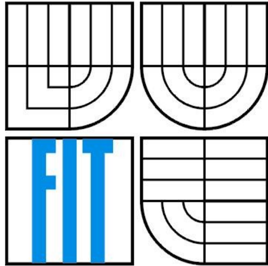


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

RAY-TRACING S VYUŽITÍM SSE

RAY-TRACING USING SSE

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAKUB SKOTÁK
AUTHOR

VEDOUCÍ PRÁCE Ing. JIŘÍ HAVEL
SUPERVISOR

BRNO 2009

Abstrakt

Metoda Ray-tracingu je jedna z realistických metod počítačové visualizace. Tato metoda je vysoce výpočetně náročná a neexistuje pro ni hardwarový akcelerátor. Tato práce popisuje urychlení ray-tracingu za použití instrukčního souboru SSE.

Klíčová slova

Ray-tracing, SSE, Streaming SIMD Extensions, Optimalizace, Průsečík paprsku s tělesem

Abstract

Ray-tracing is one of realistic methods of computer visualization. This method is highly computationally intensive and there is no hardware accelerator for it. This labour describes the acceleration of ray-tracing using the SSE instruction set.

Keywords

Ray-tracing, SSE, Streaming SIMD Extensions, Optimalization , The intersection with the object

Citace

Skoták Jakub: Ray-tracing s využitím SSE, Brno, 2009, bakalářská práce, FIT VUT v Brně.

Ray-tracing s využitím SSE

Prohlášení

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

……………………
Jakub Skoták
20.5.2009

Poděkování

Chtěl bych tímto poděkovat Ing. Jiřímu Havlovi za vedení při vypracování této práce a poskytnutí velkého množství užitečných rad.

© Jakub Skoták, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Obsah

Úvod....6
1Teoretické předpoklady....7
1.1Co je ray-tracing....7
1.2Urychlování ray-tracingu 13
1.3Jak fungují raytracery....14
1.4Volba raytraceru....14
1.5Jednotka SSE (Streaming SIMD Extensions)....16
1.6Optimalizace pomocí SSE....20
1.7Homogenní souřadnice....21
2Implementace....22
2.1Popis implementace Aurelia....22
2.2Profilování kódu....23
2.3Intrinsiky 25
2.4Postup optimalizace....25
2.5Zajímavé techniky....27
2.6Problémy....30
3Testy....32
3.1Porovnání rychlosi SSE....32
3.2Porovnání výsledků....33
4Závěr....39

Úvod

Ray-tracing, neboli sledování paprsku, je základní algoritmus realistické počítačové grafiky. Tato metoda je postavena na sledování světelných paprsků od cíle směrem ke zdroji. Sledování paprsku není v současné době akcelerováno garafickou kartou, tak jak je tomu u OpenGL nebo DirectX. Ray-tracing je vysoce výpočetně náročná metoda a v současné době neexistuje hardware pro její akceleraci. Z důvodu vysoké náročnosti je snaha tuto medodu urychlit. V této práci je popsáno využití instrukční sady SSE pro urychlení metody ray-tracingu.

Instrukční sada SSE je standardní součástí všech moderních procesorů. Tato instrukční sada slouží pro práci s bloky čísel, tyto bloky mají velikost 128b. Nasazení instrukční sady SSE do procesorů byl revoluční krok kupředu. Od uvedení této sady došlo k několika rozšířením. V této práci je použito nejnovější sady SSE4.

Cílem této práce je efektivní nasazení SSE a dosažení maximálního urychlení za pomocí této sady instrukcí. Nedílnou součástí je vyhodnocení výsledků, které přinášejí nové poznatky o vhodnosti používání SSE při implementaci této metody.

V první části práce je vysvětlen princip ray-tracingu a rozbor rozšíření SSE. Dále jsou zde popsána nutná teoretická východiska pro tuto práci. Následující kapitola se zabývá vlastním řešením optimalizace. Je zde popsána implementace a problémy, které je třeba při implementaci řešit. V závěru je uvedeno zhodnocení celé práce a vyhodnocení testů.

1Teoretické předpoklady

V této kapitole jsou popsány nutné teoretické předpoklady. Je provedeno seznámení s metodou ray-tracingu a možnostmi její optimalizace. Dále se kapitola zabývá podstatou SSE a jejího využití pro optimalizaci raytracerů.

1.1Co je ray-tracing

Ray-tracing je vysoce výpočetně náročná metoda počítačové vizualizace. Patří mezi realistické zobrazovací metody, pomocí které lze dosáhnout velmi realistického zobrazení modelu. Tato metoda spočívá v postupném sledování paprsků odražených od modelu směrem k uživateli. Umožňuje zobrazení odrazů a odlesků objektů, lom světla v objektech a dalších jinak velmi těžko dosažitelných jevů.

Pro výpočet ray-tracingu je třeba znát popis 3D scény. 3D scéna se skládá z objektů, které mají svoji pozici, tvar, barvu a další vlastnosti materiálu. Další důležitou součástí popisu 3D scény jsou zdroje světla, které mají svoji pozici a barvu. Dále potřebujeme znát pozici pozorovatele.

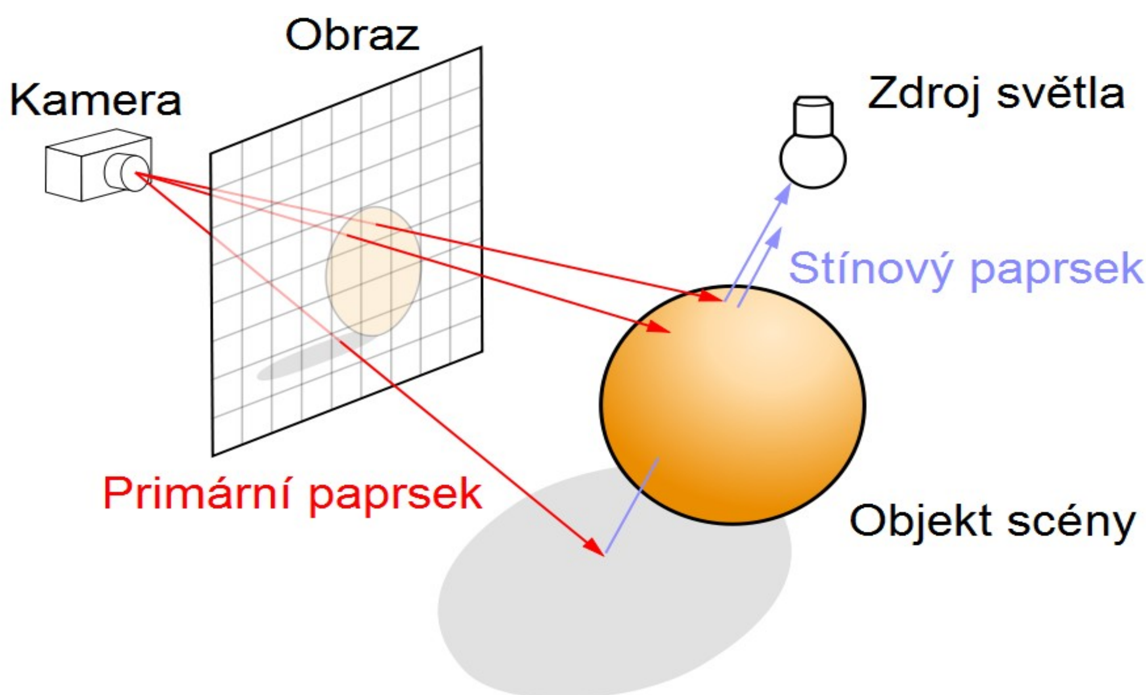
Ilustrace 1: www.svethardware.cz: Příklad obrázku vyrenderovaného pomocí ray-tracingu

Princip metody ray-tracingu spočívá v tom, že sledujeme paprsky, které se šíří od světelných zdrojů do scény. Některé paprsky zasáhnou objekty, kde se podle jejich vlastností lomí, odrážejí a rozptylují. Obraz scény tvoří paprsky dopadlé na projekční plochu.

Metoda ray-tracingu zahrnuje efekty vznikající vzájemnou interakcí objektů ve scéně (tj. například odrazy ostatních těles na povrchu lesklého objektu a lom světla při průchodu průhledným tělesem). Dokáže určit stíny vržené různými tělesy (tyto stíny jsou však ostře ohraničeny). Protože je velmi výpočetně nákladné sledovat paprsky ze zdrojů světla, postupuje se v praxi naopak. Paprsek je vržen ve směru od pozorovatele do scény přes pixely obrazu. Hledáme, co je v daném pixelu vidět, podle toho rozhodneme o výsledné barvě pixelu.

Rozlišujeme několik typů paprsků. Jsou to primární paprsek - vyslaný od pozorovatele scény, sekundární paprsek - vzniká odrazem nebo lomem paprsku a stínový paprsek - vyslaný z místa dopadu paprsku na objekt ke zdrojům světla pro zjištění, leží-li ve stínu. Není-li objekt ve stínu, je pro něj vyhodnocen osvětlovací model. Lom stínových paprsků se zanedbává.

1.1.1Popis metody



Ilustrace 2: cs.wikibooks.org/wiki/Raytracing: Diagram ilustruje algoritmus ray-tracingu pro renderování obrázku.

Při sledování paprsků hledáme vlastně jejich průsečíky s objekty scény. Naivní algoritmus testuje navzájem každý paprsek s každým objektem scény (a se všemi polygony v každém objektu), což vede ke značné časové náročnosti. Každý průsečík paprsku s objektem generuje dva sekundární paprsky a stínový paprsek. V každém dalším průsečíku je zapotřebí provést stejné operace, jedná se tedy o rekurzivní výpočet.

Podmínkou ukončení rekurze je dopad paprsku na difúzní povrch nebo dosažení předem stanovené maximální hloubky rekurze, respektive energie paprsku klesne pod určitý práh.

Rekurzivní výpočet je popsán následující rovnicí:

Rovnice byla převzata z http://cs.wikibooks.org/wiki/Raytracing [9]

$$I(P) = I_{local}(P) + I_{global}(P) = I_{local}(P) + k_{rg}I(P_r) + k_{tg}I(P_t)$$

P - průsečík

P_r - další průsečík odraženého paprsku

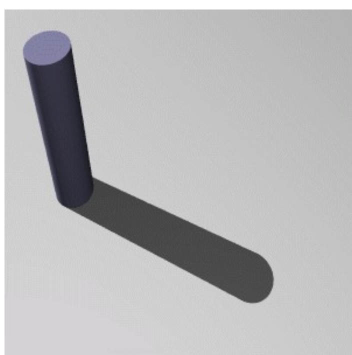
P_t - další průsečík propuštěného paprsku

k_{rg} - globální koeficient odrazivosti (reflexe)

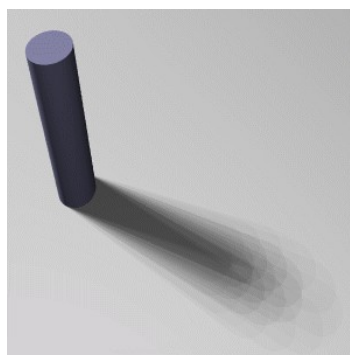
k_{tg} - globální koeficient propouštění (lomu, transmise, refrakce)

1.1.2Nevýhody ray-tracingu

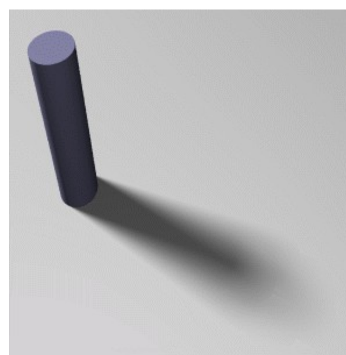
Hlavní nevýhodou metody ray-tracingu je vznik ostrých stínů jako důsledek toho, že se ve scéně vyskytují pouze bodové zdroje světla. Plošná světla se většinou nahrazují mnoha bodovými zdroji, což má výrazný dopad na rychlost výpočtu. Tento postup nevede k dosažení měkkých polostínů, ale k vržení mnoha překrývajících se stínů. Čím více je světel, tím více stínů se překrývá a tím lepší je napodobení měkkého stínu. V praxi se potřebný počet světel často počítá adaptivně vzhledem k tomu, že pro zobrazení stínů velkých ploch je dostačující použít menší počet světel než v případě členitých objektů nebo složitých textur.



Ilustrace 3: herakles.zcu.cz: bodové zdroje světla

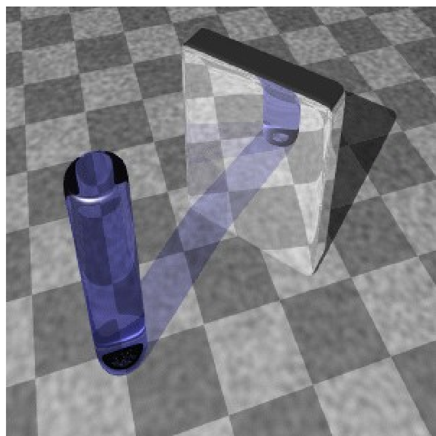


Ilustrace 4: herakles.zcu.cz: aproximace 25 světly

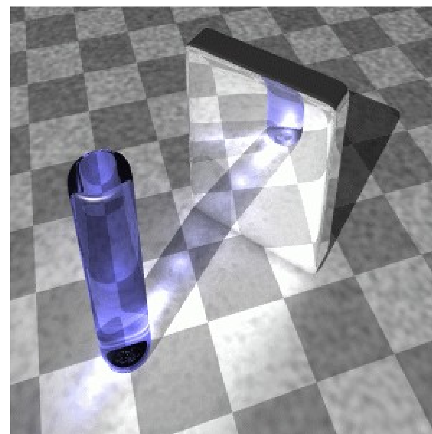


Ilustrace 5: herakles.zcu.cz: aproximace až 2500 světly s adaptivním dělením

Při zjišťování viditelnosti světla pomocí stínového paprsku se bere v úvahu pouze přímá cesta paprsku. Lesklé objekty tedy odrážejí okolí, ale neodrážejí světlo. Tyto objekty se nestávají novými zdroji světla a tudíž nevrhají odlesky.



Ilustrace 6: herakles.zcu.cz: odraz objektu od zrcadla



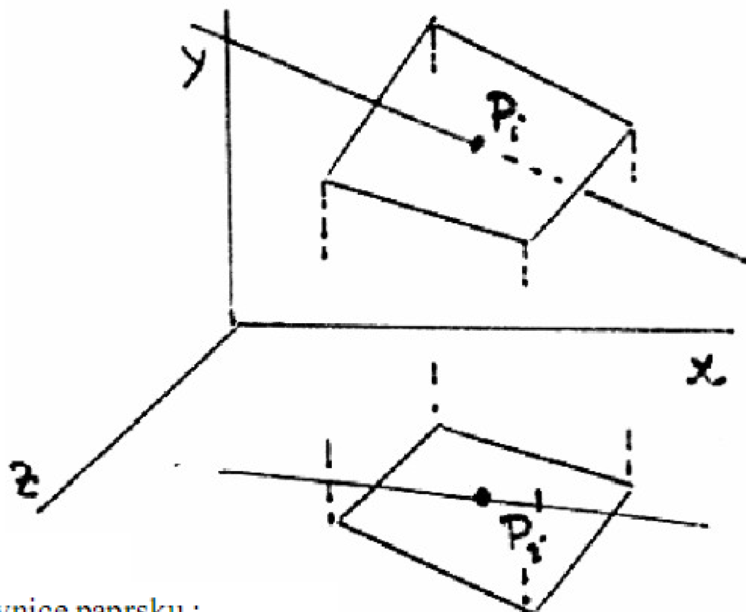
Ilustrace 7: herakles.zcu.cz: odraz světla od zrcadla

Dojde-li na scéně ke změně, např. pozorovatele, objektů, světel či materiálů, je třeba provést celý výpočet ray-tracingu znovu. Nelze přepočítat pouze tu část scény, ve které došlo ke změně, jak to umožňují jiné zobrazovací metody.

1.1.3Průsečík paprsků s tělesy

V počítačové grafice existuje mnoho modelů pro popis objektů. Jsou to například polygonální modely, drátový model a hraniční spline model. V této práci je pro popis objektů využíván model, který je založen na reprezentaci objektů složených z geometrických primitiv. Tyto primitivy vytvářejí tzv. CSG strom. Jedná se o prvky konstruktivní geometrie. V následující části je uveden příklad výpočtu průsečíku přímky s rovinou a koulí.

1.1.3.1 Průsečík paprsku a roviny



Param. rovnice paprsku :

$$x = x_0 + t \cdot (x_1 - x_0) = x_0 + t \cdot \Delta x$$
$$y = y_0 + t \cdot (y_1 - y_0) = y_0 + t \cdot \Delta y$$
$$z = z_0 + t \cdot (z_1 - z_0) = z_0 + t \cdot \Delta z$$

Rovnice roviny :

$$A \cdot x + B \cdot y + C \cdot z + D = 0$$

Rovnice parametru roviny :

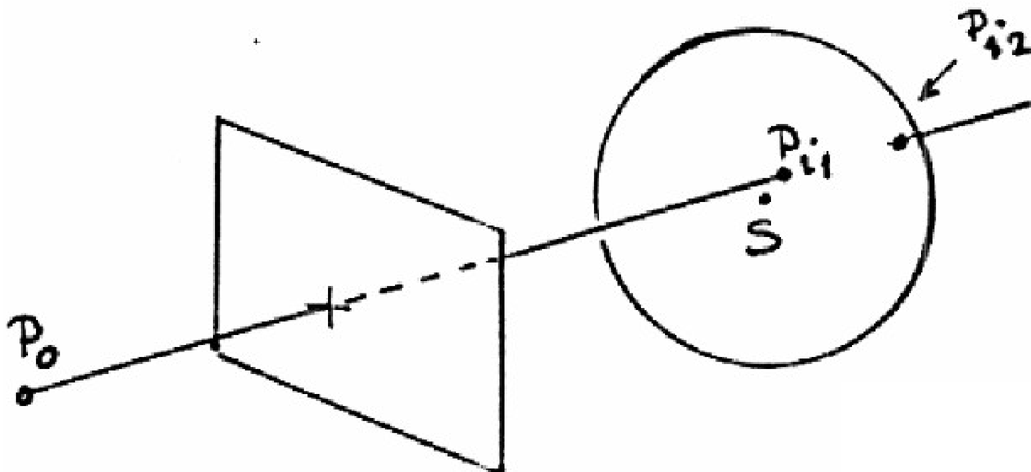
$$t = \frac{A \cdot x_0 + B \cdot y_0 + C \cdot z_0 + D}{A \cdot \Delta x + B \cdot \Delta y + C \cdot \Delta z} = \frac{n}{d}$$

pro d = 0 neprotíná rovinu.

Z průmětu do XY zjistime
jestli paprsek protíná polygon

Převzato ze slaidu do předmětu Základy Počítačové Grafiky [4]

1.1.3.2 Průsečík paprsku a koule



Param. rovnice paprsku :

$$x = x_0 + t \cdot (x_1 - x_0) = x_0 + t \cdot \Delta x$$
$$y = y_0 + t \cdot (y_1 - y_0) = y_0 + t \cdot \Delta y$$
$$z = z_0 + t \cdot (z_1 - z_0) = z_0 + t \cdot \Delta z$$

Rovnice koule :

$S(a,b,c) - střed,\ R - poloměr$

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = R^2$$
$$x^2 - 2ax + a^2 + y^2 - 2by + b^2 + z^2 - 2cz + c^2 = R^2$$

Kvadratická rovnice průsečíků $\rightarrow t_{1,2} \Rightarrow P_i$:

$$t^2 \cdot (\Delta x^2 + \Delta y^2 + \Delta z^2) +$$
$$+ 2t \cdot [\Delta x \cdot (x_0 - a) + \Delta y \cdot (y_0 - b) + \Delta z \cdot (z_0 - c)] +$$
$$+ (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - R^2 = 0$$

Normála v P_i :

$$\bar{N}_i((x_i - a)/R\ ,(y_i - b)/R\ ,(z_i - c)/R)$$

Převzato ze slaidu do předmětu Základy Počítačové Grafiky [4]

1.2Urychlování ray-tracingu

Prostý ray-tracing je velmi časově náročný, urychlovací metody ho mohou urychlit o jeden až dva řády. Následující text obsahuje obecný popis jednotlivých urychovacích metod se zaměřením na principy metod, které jsou používány nejčastěji.

1.2.1Urychlení výpočtu průsečíků

1.2.1.1Rychlejší výpočet průsečíků paprsek – objekt

Pro urychlení výpočtu průsečíku je možné používat speciálních výpočtů pro jednotlivé objekty. Existuje celá řada tzv. vyhodnocovačů průsečíků, tj. programů pro výpočet průsečíku s určitou třídou. Důležitou část těchto vyhodnocovačů průsečíků tvoří testy zásah/mimo, které vyloučí objekt z výpočtu dříve než dojde ke skutečnému analytickému výpočtu.

Dalším přístupem je provedení testu s obálkou kolem tělesa. Principem této metody je, že obálka objektu má podstatně menší složitost než objekt samotný. Výpočet průsečíku se tak stává jednodušším. Pokud je test s obálkou negativní, není třeba se objektem dále zabývat.

1.2.1.2Méně výpočtů průsečíků paprsek – objekt

Obálky objektů je možno skládat hierarchicky za sebe. Pro tyto účely se nejčastěji používají kD-stromy a oktalové stromy.

Vysoké množství stínových paprsků se stalo podnětem pro zavedení paměti překážek (light buffer). Každý bodový zdroj světla se obklopí krychlí pokrytou pravidelnou nebo adaptivně dělenou sítí. Tato kostka se nazývá paměť překážek. Toto předzpracování sény je ale značně náročné.

Výhodné je při výpočtech brát v úvahu informace z předchozích výpočtů Tento postup se nazývá koherence paprsků.

1.2.2Snížení počtu paprsků

Pro snížení počtu paprsků je možno použít adaptivní antialiasing (zředěné vysílání paprsků, interpolace při malé změně) . Tato metoda používá právě koherence paprsků.

Další možnost, jak snížit počet paprsků, je adaptivní řízení hloubky rekurze. S narůstajícím počtem lomů a odrazů paprsku dochází k útlumu intenzity paprsku. Stupeň rekurze se řídí podle dosaženého útlumu energie paprsku.

1.2.3Svazky paprsků

Tato problematika je uvedena v kapitole 1.6 Optimalizace pomocí SSE.

1.2.4Distribuce výpočtů na více částí

Výpočet ray-tracingu je velice snadno paralelizovatelný. Teoreticky je možné vytvořit pro každý paprsek zvláštní vlákno, které běží paralelně. Velkou výhodou této metody je, že mezi vlákny není zapotřebí brát v úvahu žádné synchronizace, protože paprsky jsou na sobě nezávislé. V praxi je lepší rozdělit pozorovací plochu do více částí a pro každou z nich provádět výpočet ve zvláštním vlákně. To přináší výrazné urychlení na multi-core platformách.

1.3Jak fungují raytracery

Algoritmus ray-tracingu popisuje hledání průsečíků paprsků s objekty. Pokud průsečík neexistuje, pixel ma barvu pozadí. Z průsečíků se posílají stínové paprsky směrem ke zdrojům světla. Pro všechny nezakryté zdroje světla se vyhodnotí součet osvětlení. Pokud nedošlo k překročení hloubky rekurze, vyšle se odražený paprsek a lomený paprsek z průsečíku. Barva paprsku je dána součtem barev osvětlení, odraženého a lomeného paprsku.

Počet rekurzivních volání procedury sledovaní paprsku je řízen maximální hloubkou rekurze. Je-li hloubka rekurze rovna 1, jedná se o tzv. ray-casting – vyhodnocují se pouze primární paprsky a jejich dopady na nejbližší objekt, při použití stínových paprsků jsou vyhodnoceny stíny. Pro zobrazení odrazů musí být hloubka rekurze minimálně 2 a pro řešení průhledných těles je minimum hloubky rekurze 3.

1.4Volba raytraceru

Pro provedení optimalizace bylo nutno zvolit vhodný raytracer. Při volbě raytraceru jsem prostudoval řadu dostupných raytracerů a snažil se vybrat jednoduchý a přehledný raytracer s vhodným výstupem. Jelikož jsem se rozhodl pro optimalizaci vektorových výpočtů, hledal jsem raytracer, který zpracovává různé objekty, nikoliv pouze trojúhelníky. Popis prostudovaných raytracerů je uveden v následujícím textu.

1.4.1POV-Ray

POV-Ray (The Persistence of Vision Ray-Tracer) byl vyvinut z DKBTrace 2,12 (autoři: David K. Buck a Aaron A. Collins). Scéna je popsána textovým souborem. Je to velmi zajímavý raytracer s grafickým rozhraním. Pro popis scény slouží interní jazyk. Tento raytracer zvládá různé osvětlovací modely, různé reprezentce objektů. Je tudíž příliš komplexní a složitý, proto jsem se rozhodl jej nepoužívat.

1.4.2 Simple Ray Tracer

Simple Ray Tracer je raytracer napsaný v C++, kód je objektově orientovaný, velmi jednoduchý a intuitivní. Poskytuje tak dobré základy, jak se naučit ray-tracing. Zásadní nevýhodou Simple Ray Traceru je, že zpracovává pouze trojúhelniky a výstupem je obrázek ve formátu ppm, který není běžně používaný.

1.4.3 RayWatch

Ray Watch je jednoduchý raytracer napsaný v C++, který slouží pro studijní účely. Pro nahrávání obrázků je zde použita knihovna SDL. Ray Watch zpracovává trojúhelníky, čtverce a roviny. Po spuštění se vybere jedna ze tří možných scén, která bude renderována. Při mém testování došlo v průběhu renderování k chybě, proto jsem se rozhodl tímto raytracerem dále nezabývat.

1.4.4 Minilight

Jde o command-line aplikaci, parametrem je soubor s popisem scény. Podporované platformy jsou Windows, Mac a Linux. Zajímavostí je, že používá Monte-carlo path-tracing transport, tj. metodu, kdy se osvětlení scény vypočítá pomocí sochastické metody Monte-carlo. Důsledkem této metody je zlepšení kvality osvětlení scény. Jedná se velice malý a přehledný raytracer.

Výstupem je ppm obrázek. Jedná se o další z řady raytracerů, který zpracovává pouze trojúhelníky. To jsou důvody, proč jsem se rozhodl vybrat jiný raytracer.

1.4.5 Aurelius

Velice přehledný raytracer napsaný v C++. Tento raytracer byl vyvinut Adamem Heroutem v roce 2006. Objekty scény nejsou složeny z trojúhelníků. Scéna se skládá ze základních objektů, roviny, koule a válce. Paprsky jsou jeden po druhém vrhány do scény a hledají se průsečíky s objekty scény.

Raytracer využívá knihovny GLUT pro vykreslení vyrenderované scény do okna. Je to studijní raytracer, ze kterého je dobře pochopitelná samotná podstata ray-tracingu. Právě jeho jednoduchost, přehlednost a způsob výpočtu průsečíků s objekty jsou důvody, proč jsem si vybral tento raytracer k optimalizaci.

1.5Jednotka SSE (Streaming SIMD Extensions)

Jedná se o rozšírení architektury x86 o vektorové, nebo také SIMD instrukce. Jsou to instrukce, které dokáží zpracovat jednou instrukcí více dat najednou. Odtud také plyne zkratka SIMD, neboli Single Instruction, Multiple Data. V současné době existují verze SSE, SSE2, SSE3, SSE4.1, SSE4.2. Mezi verzemi je zaručena zpětná kompatibilita.

Před uvedením SSE se používala jiná technologie SIMD, s názvem MMX. Hlavním omezením technologie MMX je skutečnost, že pracuje pouze s celými čísly. Tento nedostatek prolomila multimediální technologie 3Dnow! konkurenční firmy AMD. Další omezení vyplývá ze skutečnosti, že registry MM0 až MM7 jsou vlastně registry pro zpracování instrukcí v pohyblivé řadové čárce FPU ST0 až ST7. Z toho důvodu není možné zároveň provádět MMX instrukce a floating point. Pokud programátor chce využít v kódu oba typy operací, musí hodnoty registrů nejprve uložit do paměti a po skončení práce MMX nebo FP je zase obnovit. K tomu slouží instrukce FSAVE a FRSTOR. V dnešní době SSE plně nahrazuje MMX.

3DNow! je od počátku zamýšlena tak, aby překonala do té doby silně problematické místo v 3D grafických aplikacích a to operace v plovoucí řádové čárce. Právě vznik 3DNow! byl impulzem pro Intel, aby vytvořil instrukční sadu SSE. Technologie 3DNow! obsahuje 21 nových instrukcí, které podporují SIMD operace s plovoucí desetinnou čárkou včetně celočíselných operací a výrazně rychlejší přepínání mezi MMX a matematickými FPU operacemi.

1.5.1Verzování

Streaming SIMD Extensions přináší nové instrukce (ne všechny se týkají SIMD) a nové datové typy. Verze SSE 2 tyto datové typy rozšiřuje. Následující verze přináší pouze nové instrukce. V dalších kapitolách této práce jsou jednotlivé verze probrány podrobněji.

1.5.1.1Verze SSE

První verze SSE je implementována od procesorů řady Pentium 3.

Rozšíření SSE přidává následující vlastnosti k architektuře IA-32, osm 128bitových datových registrů XMM, 32Bitový kontrolní registr MXCSR, který poskytuje kontrolu řízení a stavu operací prováděných s registry XMM. Do XMM registru je možné ukládat data v podobě 128bitového bloku reálných čísel s jednoduchou přesností (čtyři hodnoty čísel IEEE 754 v plovoucí řádové čárce s jednoduchou přesností). Tato verze obsahuje instrukce provádějící operace typu SIMD na hodnotách v XMM registrech a instrukce ukládající a obnovující stav registru MXCSR a rozšíření instrukce CPUID.

1.5.1.2Verze SSE2

Druhá verze SSE se objevila s příchodem procesoru Pentium 4 (Willamette).

Verze SSE2 rozšiřuje stávající SSE o podporu bloku reálných čísel v plovoucí řádové čárce s dvojitou přesností (double, standard IEEE 754) a o podporu bloků celých čísel. Jsou zde rozšíření o možnost pracovat se 128bitovymi daty ve formátu 16 slabik a 8 slov. Dále zde jsou přidány nové instrukce pro práci s novými datovými typy, instrukce pro práci se skaláry a bloky reálných čísel s dvojitou přesností, nové SIMD instrukce pro práci s celými čísly, 128bitová verze SIMD instrukcí MMX a nové možnosti kontroly cachování a řízení kódu.

1.5.1.3Verze SSE3

Tato verze je zařazena od procesorů Pentium 4 s jádry Prescott .

Rozšíření SSE3 nepřináší nové datové typy. Rozšíření se týká především instrukcí, kterých je navíc 13. Deset instrukcí slouží pro podporu a vylepšení stávajících SIMD instrukcí rozšíření SSE a SSE2. Jsou to například horizontalni operace (HADD). Jedna SSE3 instrukce akceleruje programování x87 FPU díky zavedení konverze reálných hodnot na čísla celá. Poslední dvě instrukce (MONITOR a MWAIT) urychlují synchronizaci vláken.

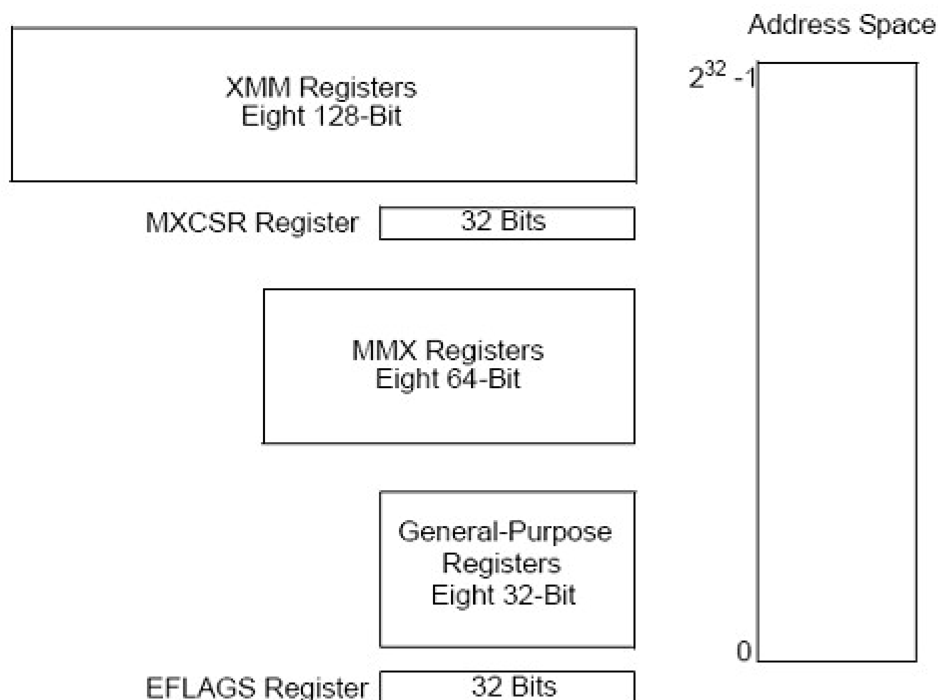
1.5.1.4Verze SSSE3

Toto rozšíření obsahuje 32 nových instrukcí, resp. 16 instrukcí, kde každá je ve verzi 64b MMX a 128b XMM. Poprvé se verze SSSE3 objevila s procesory Xeon 5100 Series, pro stolní počítače Pentium Dual Core.

1.5.1.5Verze SSE4

Verze SSE4 obsahuje 54 nových instrukcí, z toho ve verzi 4.1 je obsaženo 47 instrukcí a její implementaci obsahují procesory řady Core 2 Duo s jádry Penryn a vyšší. Zbývajících 7 instrukcí přichází na řadu až s procesory Core i7 ve verzi 4.2. Zajímavostí je, že AMD si vytvořilo vlastní klon SSE4a. SSE4a je alternativou SSE 4.1, obsahuje ale jen 4 nové instrukce. Nejzajímavější instrukci DPPS bohužel neobsahuje. Před uvedením SSE4, se někdy mylně označovala sada SSSE3 jako SSE4. Existuje verze SSE5, což je produkt AMD. Nejedná se o novou verzi, nybrž o konkurenční produkt. Verze SSE5 nebyla doposud implementována do žádného z procesorů firmy AMD.

1.5.2Programátorské prostředí SSE



Ilustrace 8: Intel® 64 and IA-32 Architectures Software Developer's Manual: Prostředí pro programování SSE

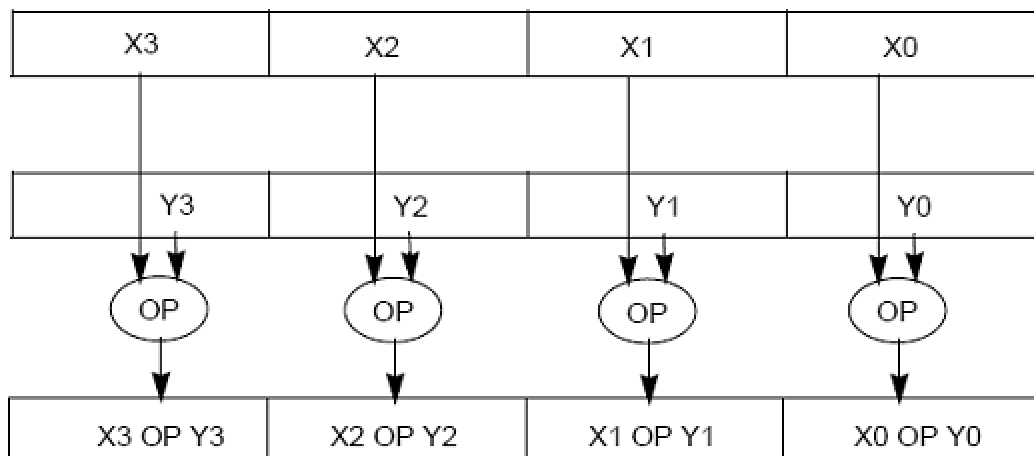
Programátorské prostředí SSE zahrnuje nové registry XMM, kontrolní registr MXCSR, registry MMX, registry pro obecné využití, registr EFLAGS a adresový prostor procesoru.

Registry XMM je možné použít pouze k výpočtům a nelze je využít k adresování operandů v paměti. K adresování operandů slouží registry obecného využití. Data se do registrů XMM nahrávají a ukládají do paměti po 32, 64 nebo 128 bitových blocích, nejnižší slabika se ukládá do paměti jako první. Registr MXCSR je 32bitový registr poskytující informace o stavu a prostředky pro řízení práce se SIMD instrukcemi. Obsah tohoto registru je možné nahrát z paměti instrukcí LDMXCSR nebo FXRSTOR a uložit do paměti instrukcí STMXCSR nebo FXSAVE.

Aby bylo možné rozšíření SSE využít, musí existovat podpora ze strany operačního systému a procesoru. Podporu SSE lze zjistit prostřednictvím instrukce CPUID.

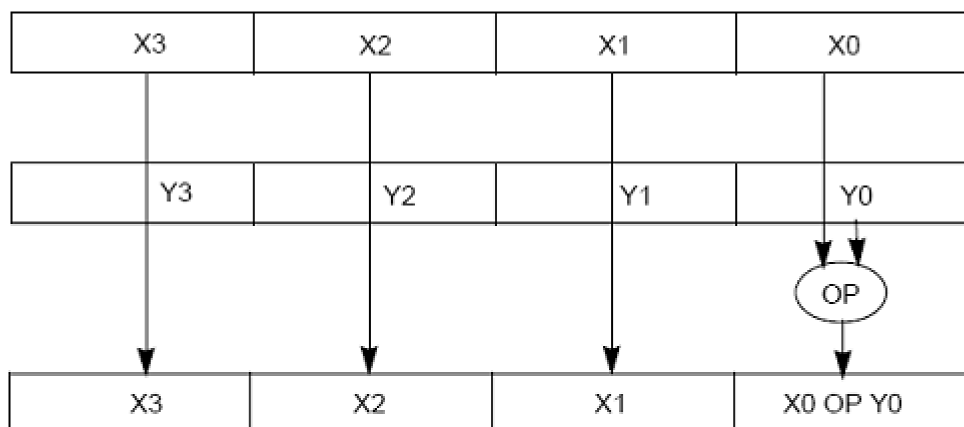
1.5.3Instrukce skalárů a bloků reálných čísel

Instrukce pracující s blokovým datovým typem provádějí operace s bloky reálných čísel s jednoduchou přesností (float). Každý zdrojový a cílový operand obsahuje čtyři reálná čísla. Cílový operand obsahuje výsledek operace, která je prováděna paralelně se všemi čtyřmi hodnotami. Jsou to instrukce končící PS, např. ADDPS.



Ilustrace 9: Intel® 64 and IA-32 Architectures Software Developer's Manual: Operace nad dvěma bloky reálných čísel s jednoduchou přesností

Skalární instrukce pracují s nejnižšími (nejméně významnými) dvojslovy zdrojových a cílových operandů. V nejnižším dvojslově cílového operandu se ukládá výsledek operace a do ostatních tří dvojslov cílového operandu se kopírují hodnoty z prvního zdrojového operandu. Skalární operace jsou podobné operacím koprocesoru x87 FPU. Tyto instrukce končí SS, např. ADDSS.



Ilustrace 10: Intel® 64 and IA-32 Architectures Software Developer's Manual: Operace nad dvěma skalárními reálnými čísly s jednoduchou přesností

Stejná situace je u instrukcí zpracovávajících čísla s dvojitou přesnosí. Jediným rozdílem je to, že se jedná o blok dvou čísel místo čtyř. Tyto instrukce jsou zakončeny SD pro skalár a PD pro vektorové operace.

1.5.4Instrukce DPPS (Dot Product of Packed Single Precision Floating-Point Values)

Instrukce DPPS je součástí rozšíření SSE4.1. Slouží pro výpočet skalárního součinu vektorů, což je velmi častá operace. Má tvar DPPS xmm1, xmm2/m128, imm8, kde xmm1 a xmm2 registry obsahují vektory. Osmibitové číslo imm8 je maska, vyšší 4 bity určují, se kterými čísly z XMM se bude provádět výpočet, zbývající 4 bity určují cíl, kam se uloží výsledek. Tato instrukce je i ve verzi DPPD pro výpočet dvousložkového vektoru obsahující čísla s dvojitou přesností.

Příklad výpočtu skalárního součinu třísložkového vektoru pomocí fpu vypadá následovně:

```
float DotProduct  = u.x*v..x + u.y*v.y + u.z*v.z;
```

Podíváme-li se na výpis assembleru, uvidíme, že se jedná o poměrně složitý výpočet, skládající se téměř z 40 instrukcí.

```
00061393  fld         dword ptr [edx+4]
00061396  fstp        dword ptr [esp+4]
0006139A  fld         dword ptr [ecx+8]
…. . .
000613F4  fmulp       st(2),st
000613F6  fmulp       st(2),st
000613F8  fsubrp      st(1),st
000613FA  fstp        dword ptr [eax+8]
```

Za pomoci SSE docílíme stejného efektu za použití jedné instrukce, plus případný přístup do paměti:

```
__m128 DotProduct  = _mm_dp_ps(u.SseData, v.SseData, 0x7f);
```

Výpis assembleru pak je následující:

```
002D1823  dpps        xmm0,xmm2,7Fh
```

1.6Optimalizace pomocí SSE

K optimalizaci ray-tracingu pomocí SSE existují dva přístupy. V této kapitole jsou o každém z těchto přístupů uvedeny stručné informace.

1.6.1Zpracování svazků paprsků

Paprsky vrhané do scény mají společný počátek a svírají mezi sebou malý úhel. To je výhodné jak pro průchod scénou, tak pro větvení výpočtu. Svazky se zpracovávají v rámci SSE registrů, kdy jeden

svazek tvoří čtveřice paprsků. Toto použití je poměrně jednoduché, vyžaduje ale značné zásahy do původního kódu raytraceru. Dále je třeba ošetřit případ rozpadu svazku na jednotlivé paprsky, jako je tomu třeba při odrazu od kulových ploch.

1.6.2Urychlení vektorových operací

Jednotka SSE je vytvořena pro práci s vektory a urychlování operací s nimi. Je vhodné implemetovat vektorové výpočty pomocí sse, kdy jednotlivé operace provádí jedna instrukce, na rozdíl od bloku instrukcí matematického koprocesoru. Tento přístup nevyžaduje takové změny v kódu, jako první přístup. Z těchto důvodů jsem ve své práci SSE použil pro urychlení vektorových operací při výpočtu průsečíků s tělesy. Nevýhodou je však nevyužití plné kapacity XMM registrů. Výpočet se provádí pouze pro tříprvkový vektor. Plného využití XMM registů lze docílit za předpokladu použití homogenních souřadnic.

1.7Homogenní souřadnice

Pro snažší provádění všech lineárních transformací bodů a vektorů v prostoru je vhodné všechny body (včetně koncových bodů úseček a vektorů) specifikovat ve čtyřrozměrném prostoru, přičemž poslední souřadnice (označovaná písmenem *w*) je nastavena na hodnotu 1.0 pro body a 0.0 pro vektory. Poslední, tj. čtvrtá souřadnice se podle své úlohy při výpočtech nazývá váha (*weight* - z toho vyplývá i její jednopísmenné označení). Po tomto rozšíření původních 3D souřadnic do 4D prostoru je možné pro specifikaci lineárních transformací a současně i perspektivní projekce používat transformační matice o velikosti 4x4 prvky.

Při provádění perspektivní projekce, která se v 4D chová jako běžná lineární transformace , pro provedení transformace je pouze nutné zachovat poslední souřadnice bodů na jedničce. Homogenní souřadnice nám usnadňují veškeré 3D sesp 2D transformace, jako jsou scale, rotation, translation a podobně. Právě homogenní souřadnice jsou ideální pro práci v SSE, využije se tak plná velikost registrů. V implemenaci raytraceru Aurelius k žádným transformacím nedochází, proto v mnou vybraném raytraceru nejsou použity.

2 Implementace

Součástí této kapitoly je rozbor raytraceru Aurelius a postupy jeho optimalizace. Text také obsahuje vysvětlení, jakým způsobem se postupuje při hlednání stěžejních míst pro optimalizaci. V závěru kapitoly jsou uvedeny problémy, které nastaly při provádění optimalizace raytraceru a které je třeba řešit.

2.1 Popis implementace Aurelia

Aurelius je objektově orientovaný raytracer napsaný v C++. Popis scény je tvořen instancemi tříd `Camera`, `ObjectGroup` a `LightSet`. Postup vytváření scény je následující. Vytvoříme materiály, které budeme ve scéně používat, vytvoříme instance `Material` a přidělíme jim příslušné vlastnosti. Pomocí `ObjectGroup::Add`, přidáváme do scény nové ojekty. Metodou `Camera::LookAt` nastavíme pozici a směr kamery, což postačuje pro kompletní popis scény.

Základem pro vykreslení je třída `AtomicRayTracer`, která obsahuje implementaci ray-tracingu. Samotný výpočet se provede pomocí metody `AtomicRayTracer::Render`, pro vykreslení obrázku je použito knihovny GLUT.

2.1.1 Třída `AtomicRayTracer`

Jádrem celého výpočtu jsou dva vnořené cykly `for`, které postupně zpracovávají všechny zobrazované pixely. Pro každý pixel se vytvoří paprsek `ray`, o sledování paprsku se stará rekurzivní metoda `AtomicRayTracer::TraceRay`. Má-li paprsek se scénou průsečík, `ObjectGroup::Intersect`, provede se výpočet. Příslušné dílčí součty celkové barvy jsou zaznamenávány ve třídě `Color`. Nedošlo-li k protnutí, výsledná barva je černá.

2.1.2 Struktura `Vector`

V této stuktuře je definován vektor jako trojice bodů `x`,`y` a `z`. Je zde definována celá řada operací, jako je součet `+`, rozdíl `-`, skalární součin `*`, vektorový součin `cross` a normalizovaný vektor `Normalize`.

2.1.3 Třída `Shape`

Tato třída slouží pro popis tvarů objektů. Jednotlivé tvary přebírají vlastnosti této třídy. Nejdůležitější součástí je struktura `Intersection`, která slouží pro popis průsečíku s tělesem.

2.1.3.1Koule

Koule je popsána ve třídě `Sphere : public Shape`, je popsána pomocí středu a poloměru. Průsečík spočítáme vyřešením kvadratické rovnice.

2.1.3.2Válec

Válec je popsán ve třídě `Cylinder : public Shape`, je popsán pomocí počátečního a směrového vektoru a poloměru. Výpočet průsečíku je velice podobný jako u koule. Opět řešíme kvadratickou rovnici.

2.1.3.3Plocha

Plocha má svoji definici ve třídě `Plane : public Shape`, jako vektor a bod. Výpočet průsečíku je podstatně jednodušší, než je tomu u předešlých dvou objektů.

2.1.4Nutné úpravy

Aurelius byl původně překládán překladačem GNU GCC. Protože v době tvorby této práce byla pro Windows dostupná pouze beta verze, která by dokázala přeložit SSE4.1, rozhodl jsem se využívat prostředí MS Visual studio 2008. To s sebou přineslo drobné úpravy. Nejdůležitější změnou je nahrazení datového typu `double` za typ `float`. Tato změna neměla žádný vizuální vliv na výsledný obraz. Tato změna samozřejmě přináší lepší využití SSE.

2.2Profilování kódu

Pro zjišťování, ve kterém místě tráví program nejvíce času, se používají programy zvané profilery. Tyto programy vytvářejí statistické vyhodnocení času stráveného v jednotlivých funkcích. Do MS Visual Studia je pofiler importován až od verze team, kterou jsem však neměl k dispozici. Rohodl jsem se tedy použít některého volně dostupného profileru. Použil jsem profileru AMD CodeAnalyst.

2.2.1AMD CodeAnalyst

AMD CodeAnalyst je zdarma ke stažení na stránkách AMD. Existuje ve verzích jak pro Windows, tak pro Linux. Verze pro Windows se automaticky integruje do MS Visual Studia, je jej ale možné používat i samostatně. Ačkoliv je CodeAnalyst vyvíjen pro platformu AMD, pracuje bez problémů i s procesory firmy Intel. Pracuje na principu náhodného přístupu do programu a zjišťování v jaké se funkci se zrovna nachází program. Takto potom vypadá vyprofilovaný výstup.

System Data | System Graph | Processes | Aurelius_msv.exe - Data | Aurelius_msv.exe - shape.h

CS:EIP	Symbol + Offset	64-bit	Timer samples - C0	Timer samples - C1
0x8d1ed0	Plane::IntersectFirst		26.89	31.25
0x8d2250	ObjectGroup::Intersect		14.71	11.88
0x8d1710	Sphere::IntersectFirst		13.38	18.75
0x8d2d30	AtomicRayTracer::TraceRay		11.24	8.75
0x8d1b90	Cylinder::Intersect		9.34	6.25
0x8d23f0	ShapeObject::Intersect		8.84	8.75
0x8d1600	Shape::IntersectFirst		3.6	2.5
0x8d2720	AtomicRayTracer::Render		2.59	1.88
0x8d1320	Vector::Normalize		1.33	1.88
0x8d14e0	Camera::GetRay		1.07	1.88
0x8d24b0	PointLight::GetShadowRay		0.95	0.62
0x8d4070	Ranges<Shape>::FirstEdgeGreater		0.88	1.25
0x8d4e60	std::vector<Range<Shape>,std::allo...		0.82	0.62
0x8d1180	Image::GetPixmap		0.76	0
0x8d15c0	PhongMaterial::GetPhongInfo		0.63	0
0x8d1dd0	Cylinder::GetNormal		0.57	0
0x8d4a90	std::vector<Range<Shape>,std::allo...		0.38	0.62
0x8d4b00	std::vector<Range<Shape>,std::allo...		0.38	1.25
0x8d41d0	Ranges<Shape>::Add		0.32	0
0x8d574e	operator delete		0.32	0.62
0x8d1b50	Sphere::GetNormal		0.19	0
0x8d1110	Image::Image		0.13	0
0x8d2120	Plane::GetNormal		0.13	0
0x8d2530	PointLight::GetColor		0.13	0.62
0x8d5120	std::allocator<Range<Shape> >::allo...		0.13	0
0x8d57a0	operator new		0.13	0
0x8d56c0	std::_Uninit_fill_n<Range<Shape> *,...		0.06	0.62

1 function, 54 instructions, Total: 242 samples, 13.88% of samples in the module, 1.36% of total session samples

Ilustrace 11: Příklad vyprofilovaného Aurelia

System Data | System Graph | Processes | Aurelius_msv.exe - Data | Aurelius_msv.exe - shape.h

Search for: All Go Asm

58 0 51 67 Function + -

C:\Users\skoty\Documents\Bakalarka\Aurelius_msv\Release\Aurelius_msv.exe

Address	Line	Trac	Source	Cod	Timer samples - C0	Timer samples - C1
	52		`const Vector &a = ray.Start();`		0	0
0x8d1713	53		`const Vector &p = ray.Direction(...`		6.13	6.67
0x8d1717	54		`float aa = p*p;`		25.47	23.33
0x8d1759	55		`float bb = 2* (p * (a - c));`		27.36	36.67
0x8d17be	56		`float cc = (a-c)*(a-c) - r*r;`		9.91	3.33
0x8d17f9	57		`float D = bb*bb - 4*aa*cc;`		16.51	20
0x8d183f	58		`if (D > 0) {`		8.49	6.67
0x8d184f	59		`float sD = sqrt(D);`		0	0
0x8d1860	60		`float t1 = (-bb - sD) / (2*aa);`		2.83	3.33
0x8d187e	61		`if (t1 > 0) return Intersectio...`		0	0
0x8d18af	62		`float t2 = (-bb + sD) / (2*aa);`		0	0
0x8d18bd	63		`if (t2 > 0) return Intersectio...`		0	0
	64		`}`		0	0

1 file, 1 function, 1 line, 0 instructions, Summary: 69 samples, 3.96% of module samples, 0.39% of total samples

Ilustrace 12: Příklad vyprofilovaného Aurelia – Sphere::IntersectFirst

2.3 Intrinsiky

Většina funkcí je obsažena v knihovnách. Některé funkce jsou však zabudovány do kompilátoru. Tyto funkce jsou označovány jako *intrinsic functions* nebo *intrinsics*. Intrinsic funkce jsou obvykle implementovány jako inline funkce. Tím je eliminována režije na volání funkcí a výsledný kód se tak stává rychlý a kompaktní. Použití intrinsik ovlivňuje přenositelnost kódu, protože intrinsiky, které jsou k dispozici ve Visual C++, nemusí být k dispozici v případě jiných kompilátorů. Naopak některé intrinsiky, které by měly být k dispozici pro některé cílové architektury, nejsou k dispozici na všech architekturách. Nicméně, intrinsiky mají obvykle větší přenositelnost než inline assembler. Dále jsou intrinsiky potřebné pro 64-bitové architektury, kde inline assember není podporován. Některé intrinsiky jsou dostupné pouze jako intrinsiky, jiné i jako klasické funkce. Ve své práci jsem používal pouze intrinsiky týkající se rozšíření SSE. Pro tyto intrinsiky neexistuje ekvivalent ve tvaru funkce.

Intrinsiky se používají jako klasické funkce, mají své parametry a návratové hodnoty. Příklad intrinsic funkce je následující, `__m128 _mm_add_ss(__m128 a, __m128 b);`, jedná se o implementaci instrukce `ADDSS`. Podobný princip je i u ostatních instrukcí. Aby bylo možno intntrinsiky v kódu programu používat, je nutno vložit knihovnu *intrin.h*. Je možno také vložit pouze určité části, jako jsou například *xmmintrin.h* (SSE) nebo *pmmintrin.h* (SSE3).

2.4 Postup optimalizace

Ze všeho nejdříve bylo potřeba upravit strukturu `Vector` tak, aby byla zarovnána v paměti na 16 bitů. Podle profileru jsem vyprofiloval, ve kterých funkcích tráví program nejvíce času. Postupně jsem začal jednotlivé metody optimalizovat. Začal jsem nahrazovat jednotlivé výpočty za pomocí SSE intrinsik. Tento proces je poměrně zdlouhavý, protože je třeba optimalizaci provádět postupně, optimalizovat jeden nebo dva výpočty a zbylý výpočet ponechat na překladači. Dále je třeba otestovat, zda byl přepis správný a v případě úspěchu pokračovat. Kdyby byla celá metoda přepsána pomocí intrinsik, bylo by pak hledání chyb velice náročné. V této fázi nedochází k žádnému urychlení, je tomu spíše naopak. Výsledné urychlení je znatelné až po optimalizaci celé metody, kdy nedochází k přesunům mezi registry FPU a registry XMM.

```
//float bb = 2* (p * (a - c));
 mac = _mm_sub_ps(a.SseData,c.SseData);//ulozi do mac (a-c)
__m128 mtmp = _mm_dp_ps(p.SseData, mac, 0x7f);
 mbb = _mm_add_ss(mtmp,mtmp);//2x
//cc = (a-c)*(a-c) - r*r;
mcc = _mm_sub_ss(
        _mm_dp_ps(mac, mac, 0x7f), _mm_mul_ss(
            _mm_load_ss(&r),_mm_load_ss(&r)));
```

Text 1: Příklad optimalizovaného kódu

Narazil jsem na problém při dynamické alokaci paměti, kdy nedocházelo k zarovnávání v paměti. Bylo třeba vytvořit přetížený opreátor new, který přiděluje zarovnanou paměť ukazateli. Více se tomuto problému budu věnovat v sekci Problémy.

Po vyřešení tohoto problému jsem mohl pokračovat v optimalizací ostatních funkcí. Poučen z předchozích chyb, byla práce značně rychlejší. Po dokončení optimalizace všech výpočtů průsečíků jsem vyhodnotil zrychlení. Toto zrychlení již bylo znatelné. Nyní se převážná většina výpočtů odehrávala mezi XMM registry. Bylo ale třeba odhalit slabá místa.

Po prostudování kódu v assembleru jsem zjistil, že výpočty mezi vektory, kde bylo použito implementovaných operátorů a metod struktury `Vector`, se provádí v FPU. Tato skutečnost značně brzdila celý výpočet. V dalším textu se tomuto problému věnuji podrobněji.

Ve své práci jsem se pokoušel eliminovat některé náročné operace. Velmi náročnou operací je při optimalizaci dělení. Je několik způsobů, jak se dělení vyhnout nebo jej omezit. Jeden ze způsobů je odsunutí dělení, které se vypočítává před podmínkou, dovnitř podmínky. Příklad tohoto odsunutí je následující.

```
__m128 mt1 = _mm_div_ss(citatel,jmenovatel);
      if (_mm_comigt_ss(mt1,_mm_setzero_ps()))
                                         {
             float t1;
             _mm_store_ss(&t1,mt1);
             return Intersection(Intersection::ikInto, this, t1);
   }
```

Text 2: Příklad dělení vně podmínky (pomalejší)

```
int mmask = _mm_movemask_ps(_mm_xor_ps(citatel,jmenovatel)) & 1;
  if (!mmask)
  {
     float t1;
      _mm_store_ss(&t1,_mm_div_ss(mt1cit,jmenovatel));
      return Intersection(Intersection::ikInto, this, t1);
   }
```

Text 3: Příklad dělení uvnitř podmínky (rychlejší)

Výše uvedené řešení je velice vhodné a přináší znatelné urychlení. Problém však nastává, je-li podmínka větší nebo rovno namísto větší. V tomto případě nelze takovéto konvence použí. Pokud ji chceme použít, musíme se přesvědčit, jaký má tato změna vliv na výsledek.

Dalším způsobem, jak omezit dělení, je použití inverzní hodnoty a následného násobení. Podrobnější informace o této metodě jsou uvedeny v sekci Zajímavé techniky.

Doposud jsme se ve své práci zabýval úpravami ve zdrojovém souboru *Shape.h*. Dalším souborem, vhodným pro optimalizaci, je *Vector.h*. Tento soubor obsahuje již zmiňovanou strukturu `Vector` a operace definované nat touto strukturou. Pokusil jsem se optimalizovat veškeré

elementární operace. Dospěl jsem k závěru, že jedinou funkcí vhodnou pro optimalizaci je funkce Normalize, která provádí výpočet normalizovaného vektoru. V případě optimalizace všech funkcí je třeba přetížit operátor new u třídy Light. Zde naroste režije při vytváření objektů natolik, že dochází ke zpomalení. Pokud je třeba provést výpočet v SSE, použijeme místo operátoru intrinsiku reprezentující příslušnou instrukci.

Při optimalizaci metody Normalize jsem nejdříve klasickým způsobem nahradil výpočet pomocí intrinsik a porovnal výsledek. Došlo k výraznénu urychlení. V tomto výpočtu vystupují dvě po sobě jdoucí instrukce, které jsou velice časově náročné. Jedná se o odmocninu a dělení. Pokusil jsem se tyto instrukce nahradit za inverzní odmocninu (postup známý jako Carmack invers sqare root) a násobení. Setkal jsem se ale s neúspěchem. Místo očekávaného zrychlení došlo se zpomalení a zhoršení výsledné kvality obrazu. Z tohoto důvodu jsem ponechal implementaci jako odmocninu a dělení. Více se tomuto problému budu věnovat v následující kapitole.

2.5Zajímavé techniky

V této kapitole jsou popsány různé zajímavé techniky, které jsem při vypracování této práce vyzkoušel. Některé z nich jsem úspěšně implementoval, od jiných jsem upustil. Většina těcho technik je úzce spojena s používáním SSE, ostatní se týkají optimalizace obecně.

2.5.1Zarovnaný vektor

Při práci s SSE je potřeba mít paměť, do které chceme přistupovat, zarovnanou na 16b. Toho docílíme directivou __declspec(align(n)), kde n je počet bitů na které bude paměť zarovnána. Tuto directivu používá překladač Visual Studia. Pro překladač GCC to je __attribute__((aligned(n))). Tyto directivy se bohužel neuplatňují při dynamické alokaci. Podrobnější informace o tomto problému jsou uvedeny v dalším textu.

Původní struktura vypadala následovně:

```
struct Vector {
  float x;
  float y;
  float z;
 ....
}
```

Text 4: Původní struktura Vector

Tento způspb je pro použití SSE nevhodný, proto jsem tuto strukturu upravil:

```
struct Vector {
  __declspec(align(16))float v[4];
  .....
}
```

Text 5: Upravená struktura Vector

Tato změna měla za následek nutnost provedení dalších úprav kódu. Tyto úpravy by ale nebyly nutné, pokud bych objevil již dříve následující techniku:

```
_declspec(align(16)) struct Vector
{
      union
      {
            struct { float X, Y, Z, w; };
            float Raw[4];
            __m128 SseData;
      };
};
```

Text 6: Vhodná implementace vektoru

Protože jsem již měl strukturu i náležitosti s tím spojené hotovy, použil jsem tuto techniku ve tvaru:

```
__declspec(align(16)) struct Vector {
      union
      {
            float v[4];
            __m128 SseData;
      };
    ....
}
```

Text 7: Výsledná implementace

Používání výše uvedené unie přináší zjednodušení práce. Není třeba pokaždé, když chceme načítat nebo ukládat do vektoru, používat instrukce pro přesun dat. Postačuje přístup k `SseData` přes tečkovou notaci, o instrukce přesunu se překladač postará sám.

2.5.2Inverzní odmocnina

Tato metoda výpočtu byla hojně využívána v minulosti. Použil ji legendární programátor John Carmack ve svém Quake 3. Testy prokázaly, že tato metoda je přibližně 4x rychlejší než klasický `(float)(1.0/sqrt(x))` a maximální relativní chyba je 0.00175228. Zdá se, že tato metoda je velmi užitečná. Je však popsána poměrně nepochopitelným kódem:

```
float InvSqrt(float x)
{
  float xhalf = 0.5f*x;
  int i = *(int*)&x; // get bits for floating value
  i = 0x5f375a86- (i>>1); // gives initial guess y0
  x = *(float*)&i; // convert bits back to float
  x = x*(1.5f-xhalf*x*x);//Newton step, repeating increases accuracy
  return x;
}
```

Text 8: Inverzní odmocnina

Sám jsem otestoval, že tato metoda opravdu funguje. Pokusil jsem se ji proto přepsat pomocí SSE intrinsik:

```
inline __m128 InvSqrt(__m128 x)
{
  __m128 xhalf = {0.5,0.5,0.5,0.5};
  xhalf = _mm_mul_ps(xhalf,x);
  __declspec(align(16)) float tmp[4];
  _mm_store_ps(tmp,x);
  __m128i i = _mm_load_si128((__m128i*)tmp);
  i = _mm_srli_epi32(i,1);
  __declspec(align(16)) int mem_magic[4]={0x5f375a86, 0x5f375a86,
                                          0x5f375a86,
0x5f375a86};
  __m128i magic = _mm_load_si128((__m128i*)mem_magic);
  i = _mm_sub_epi32(magic,i);
  _mm_storeu_si128((__m128i*)tmp,i);
  x = _mm_loadu_ps(tmp);
  __m128 mtmp = _mm_mul_ps(x,x);
  mtmp = _mm_mul_ps(mtmp,xhalf);
  __m128 jedenapul = {1.5,1.5,1.5,1.5};
  mtmp = _mm_sub_ps(jedenapul,mtmp);
  x = _mm_mul_ps(x,mtmp);
 return x;
}
```

Text 9: Inverzní odmocnina pomocí SSE

Nasazením této techniky jsem žádné zrychlení nezískal, naopak došlo ke zpomalení a zhoršení kvality obrazu. Z těchto důvodů jsem se rozhodl techniku inverzní odmocniny vypustit a držet se standardní instrukce `_mm_sqrt_ps`. Tato metoda není pro inplementací de SSE vhodná, protože musíme často přistupovat do pamětí a provádět přetypování mezi *iny* a *floaty*.

2.5.3 Inverzní hodnota

Ačkoliv v SSE operace dělení je obsažena, inverzní hodnota se obvykle počítá jiným způsobem.

Jedná se o odhad pomocí instrukce `rcp`, který je dále zpřesněn jedním krokem Newtonovy metody, tedy podle vzorce $y_{i+1} = 2y_i - xy_i^2$. Tento postup na rozdíl od operace dělení nepotřebuje načítat jedničku z paměti. Také bývá rychlejší, protože je možné jej rozložit mezi okolní instrukce.

```
static inline __m128 _mm_inv_ps (__m128 x)
{
	const __m128 rcpi = _mm_rcp_ps(x);
	return _mm_sub_ps(
			_mm_add_ps(rcpi, rcpi),
			  _mm_mul_ps(
			    _mm_mul_ps(rcpi, rcpi),x
			  )
		);
}
```

Text 10: Inverzní hodnota v SSE

Ve své práci jsem metodu inverzní hodnoty používal jako náhradu operace dělení. Vypočítal jsem zvlášť čitatele a jmenovatele. Ze jmenovatele jsem vypočítal pomocí této metody inverzní hodnotu a výsledky vynásobil. Došel jsem k závěru, že tento přístup je vhodný pouze, pokud dělíme dva a více různých čitatelů stejným jmenovatelem. V případě dvou čitatelů je urychlení velmi malé, je však změřitelné. Pro jednoho čitatele se setkáme s výrazným zpomalením.

2.6 Problémy

Největším problémem při práci s SSE je to, že paměť se kterou pracujeme, musí být zarovnaná na 16b. Je sice možné používat instrukce pro práci s nezarovnanou pamětí, to nám ale přístup do paměti značně zpomaluje. Pokud používáme staticky alokovanou paměť, není nic jednoduššího, než použít directivu překladače. Pokud však používáme paměť alokovanou dynamicky, situace se komplikuje. Nejlepším způsobem je použít zarovnanou alokaci `void *_mm_malloc (size_t size, size_t alignment)` z knihovny `mm_malloc.h`. Pokud již máme nainkludováno `intrin.h`, `mm_malloc.h` inkudovat nemusíme, je již obsažen. Existuje i `void _mm_free (void * ptr)`, jedná se však o klasické uvolnění pomocí free.

Pokud využíváme dědičnosti, nevyhneme se vytváření dynamických objektů. K tomu se používá operátoru `new`. Zde opět dochází k problému, protože operátor `new` nezarovnává vytvářený objekt v paměti. Řešením je přetížení operátoru pro danou třídu. Implementace vypadá následovně:

```
void *__CRTDECL operator new(size_t size) _THROW1(_STD bad_alloc)
{       // try to allocate size bytes
  void *p;
  p = _mm_malloc(size,16);//zarovnana pamet na 16b
  if (!p)
  {       // report no memory
    static const std::bad_alloc nomem;
    _RAISE(nomem);
  }
  return (p);
}
```

Text 11: Přetížení oprátoru `new`

Hledání této chyby je velice problematické, protože ji nelze odhalit v průběhu překladu. Chyba se objeví až za běhu programu, při přístupu do paměti. Objeví se hlášení o pokusu přístupu na nepovolenou adresu v paměti.

Dalším výrazným problémem je rozhodování, zda použít jednotku SSE nebo FPU. Ne vždy totiž SSE přináší očekávaný užitek. K výraznému zpomalení dochází, přesouváme-li data mezi registry SSE a FPU. Pokud chceme pracovat s SSE jednotkou, je daleko lepší počítat veškeré výpočty, a to jak blokové, tak skalární v rámci SSE registrů. Menší a méně náročné výpočty je lepší počítat na FPU. Implementace je daleko jednodušší a potencionální užitek SSE je minimální. Někdy je nejlepším řešením vyzkoušet, která z metod je rychlejší a k té či oné se přiklonit. Je však třeba si uvědomit, zda úsilí vynaložené na tento vývoj, přinese prospěch celému projektu.

3Testy

Následující text popisuje testy, které jsem s výsledným programem prováděl a hodnocení, k nimž jsem dospěl. Jako první je uvedeno porovnání jednotlivých verzí SSE, následují konkrétní testy týkající se výsledného programu.

3.1Porovnání rychlosi SSE

Jak jsem již uváděl v části 1.5.4 Instrukce DPPS, výpočet skalárního součinu je velmi častou operací. Rozhodl jsem se proto otestovat, jaký přínos tato instrukce má.

Naplnil jsem pole náhodnými čísly a prováděl jsem s nimi opakovaně skalární součin. Pro ilustraci uvádím kód jednotlivých výpočtů.

```
_mm_store_ps(out[i],
     _mm_dp_ps(
        _mm_load_ps(&in[i][0]), _mm_load_ps(&in[i+1][0]), 0xff));
```

Text 12: Výpočet pomocí SSE4.1

Použijeme-li instrukci DPPS, výpočet sestává z jediné instrukce, tudíž je velice jednoduchý.

Za použití horizontálních operací z SSE3 se kód lehce komplikuje a prodlužuje. Je však pořád dostatečně přehledný.

```
_mm_store_ps(out[i],
   _mm_hadd_ps(
      a=_mm_hadd_ps (
        a=_mm_mul_ps(
           _mm_load_ps(&in[i][0]), _mm_load_ps(&in[i+1][0])),
        a),
     a));
```

Text 13: Výpočet pomocí SSE3

Pokud používáme jen základní instrukce z SSE, jsme značně omezení. Musíme použít množství shufflů, protože zde nejsou obsaženy žádné horizontální instrukce.

```
a = _mm_load_ps(&in[i][0]);   // nahraji prvni cislo
b = _mm_load_ps(&in[i+1][0]); // nahraji druhe cislo
// vypocitam skalarni soucin
c = _mm_mul_ps(a, b);
a = _mm_shuffle_ps(c,c,0xb1);
b =  _mm_add_ps(a, c);
a = _mm_shuffle_ps(b,b,0x0a);
a =  _mm_add_ps(a, b);
// ulozim cislo zpet do pameti
_mm_store_ps(out[i],a);
```

Text 14: Výpočet pomocí SSE

Takto vypadá klasický postup, kdy je celý výpočet počítán na FPU.

```
float result= in[i][0]*in[i+1][0] + in[i][1]*in[i+1][1] +
              in[i][2]*in[i+1][2] + in[i][3]*in[i+1][3];
out[i][0] = result;
```

Text 15: Výpočet pomocí FPU

Vyhodnotil jsem naměřené výsledky, které byly následující: 2714s pro versi SSE4.1, 4758s pro SSE3, 4899s pro SSE a 6318s bez použití SSE. Z toho vyplývá, že mezi verzemi SSE a SSE3 je hlavní rozdíl v jednoduchosti zápisu. Zrychení verze SSE4.1 je téměř 60%.

3.2Porovnání výsledků

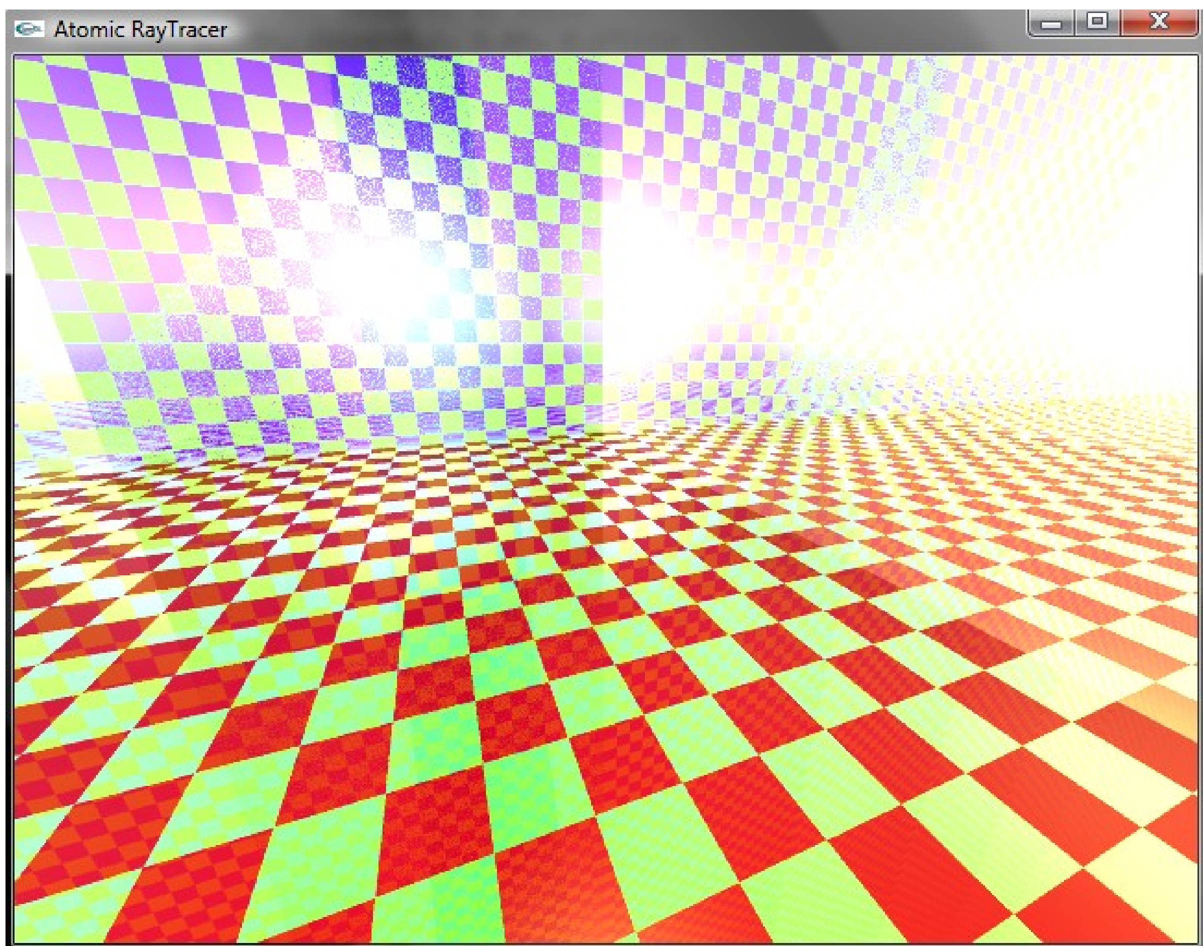
Vytvořil jsem sadu testů a vyhodnotil jejich výsledky. Testoval jsem jak jednotlivé objeky samostatně, tak kompletní scény.

Pro zajímavost jsem provedl srovnání překladačů Vicual C++ a Intel C++. Přeložil jsem původní verzi Aurelia oběma překladači. Intel C++ vygeneroval kód, který byl o 32% rychlejší, než tomu bylo u Vicual C++. Optimalizovaný kód se rychlostí lišil pouze o 8%. To znamená, že pokud do kódu jsou vnášeny intrinsiky, můžeme tím výsledný program urychlit. Na druhou stranu omezujeme překladač v optimalizaci. Musíme tedy užívat intrinsiky s rozmyslem a jen tehdy, pokud má toto řešení opravdu smysl.

Veškeré testy zde uvedené byly provedeny ma platformě Intel Core 2 Duo T9300 v prostředí Windows Vista Home.

3.2.1Roviny

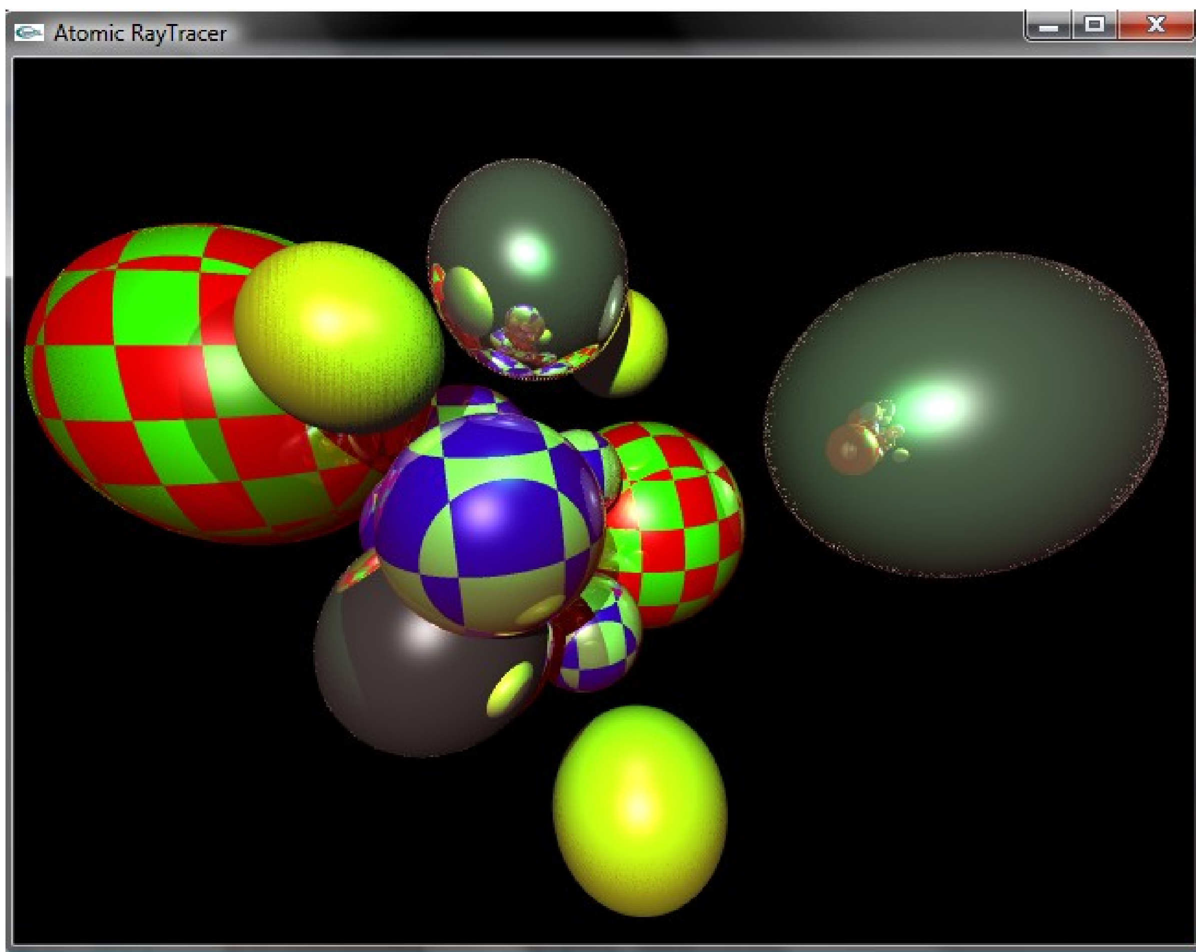
Vytvořil jsem scénu, kterou tvoří 4 roviny a 2 zdroje světla. Tento test slouží k otestování optimalizace výpočtu průsečíku paprsku s rovinou.



Po přeložení překladačem Vicual C++ byl čas bez optimalizace 26,1s, po optimalizaci se čas snížil na 17,3s. To znamená zrychlení o 34%. Považuji tedy tuto optimalizaci za přínosnou.

3.2.2Koule

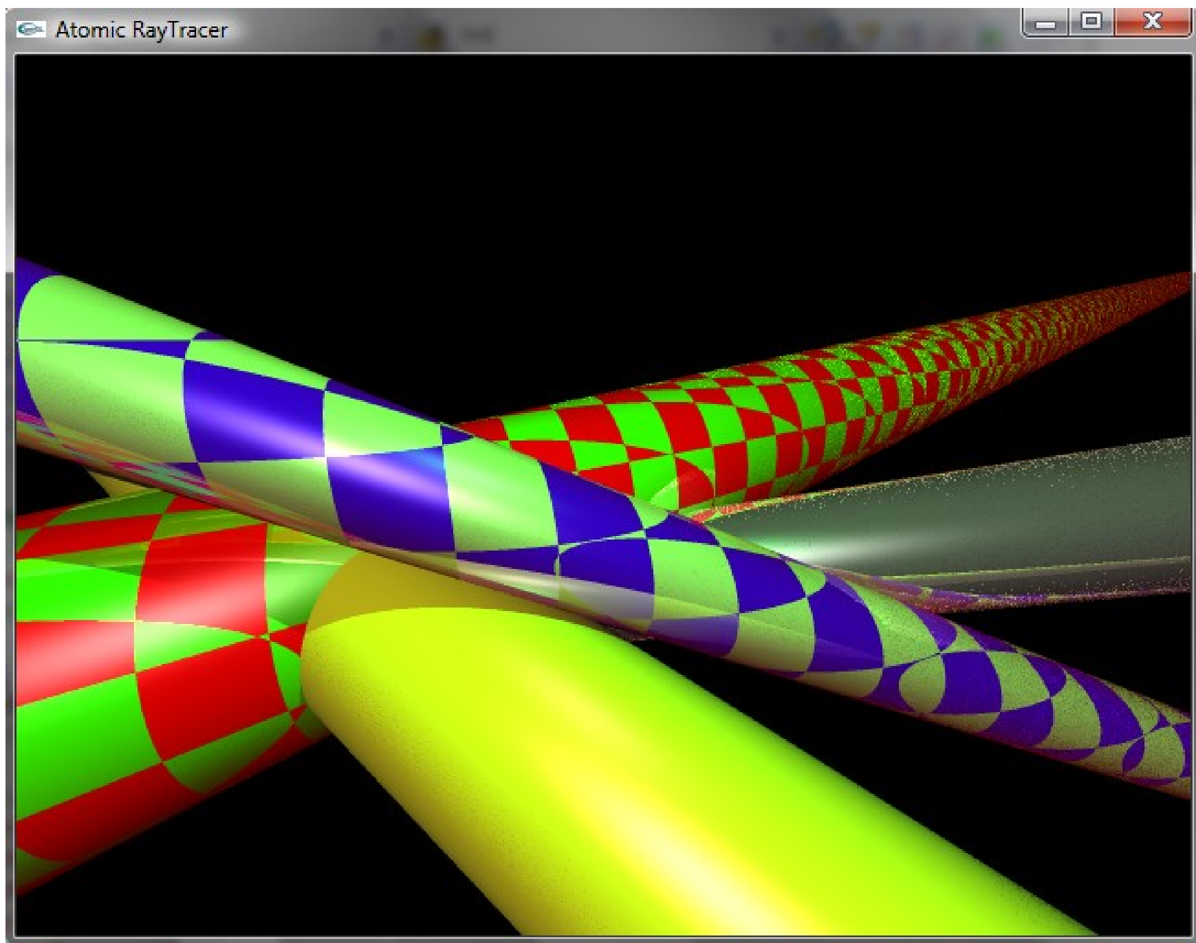
Pro tento test jsem sestavil scénu o 15 koulích různých rozměrů a 2 zdroje světla. Testoval jsem, jak dalece se projevuje optimalizace výpočtu průsečíku paprsku s koulí.



Program jsem přeložil překladačem Vicual C++, čas renderování bez optimalizace byl 3.17s, po optimalizaci se čas snížil na 2.04s. Tím jsem docílil zrychlení o 36%. Toto zrychlení je opravdu znatelné a přináší tak přínos celému programu.

3.2.3Válce

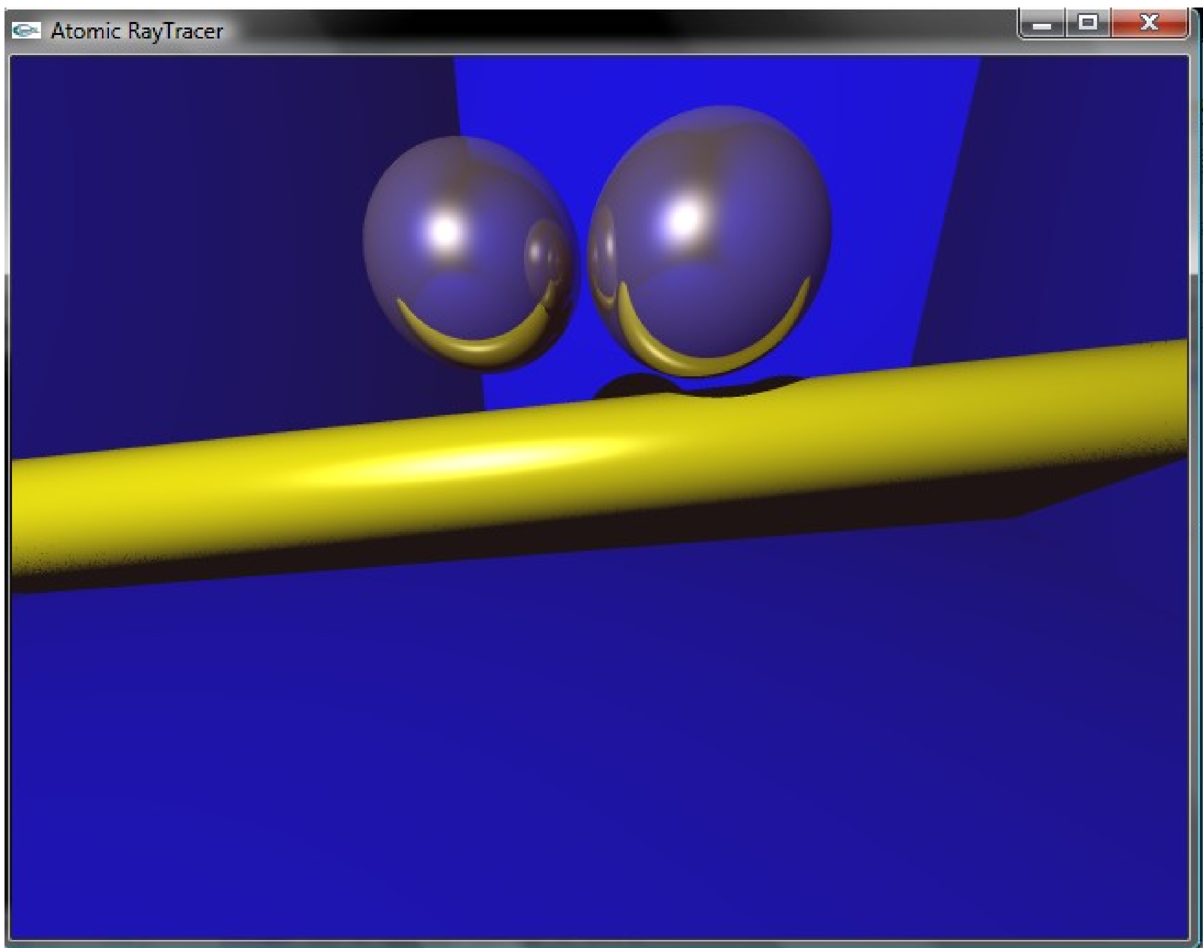
Vytvořil jsem scénu, kterou tvoří 4válce a 2 zdroje světla, abych otestoval, jak se projevuje optimalizace výpočtu průsečíku paprsku a válce.



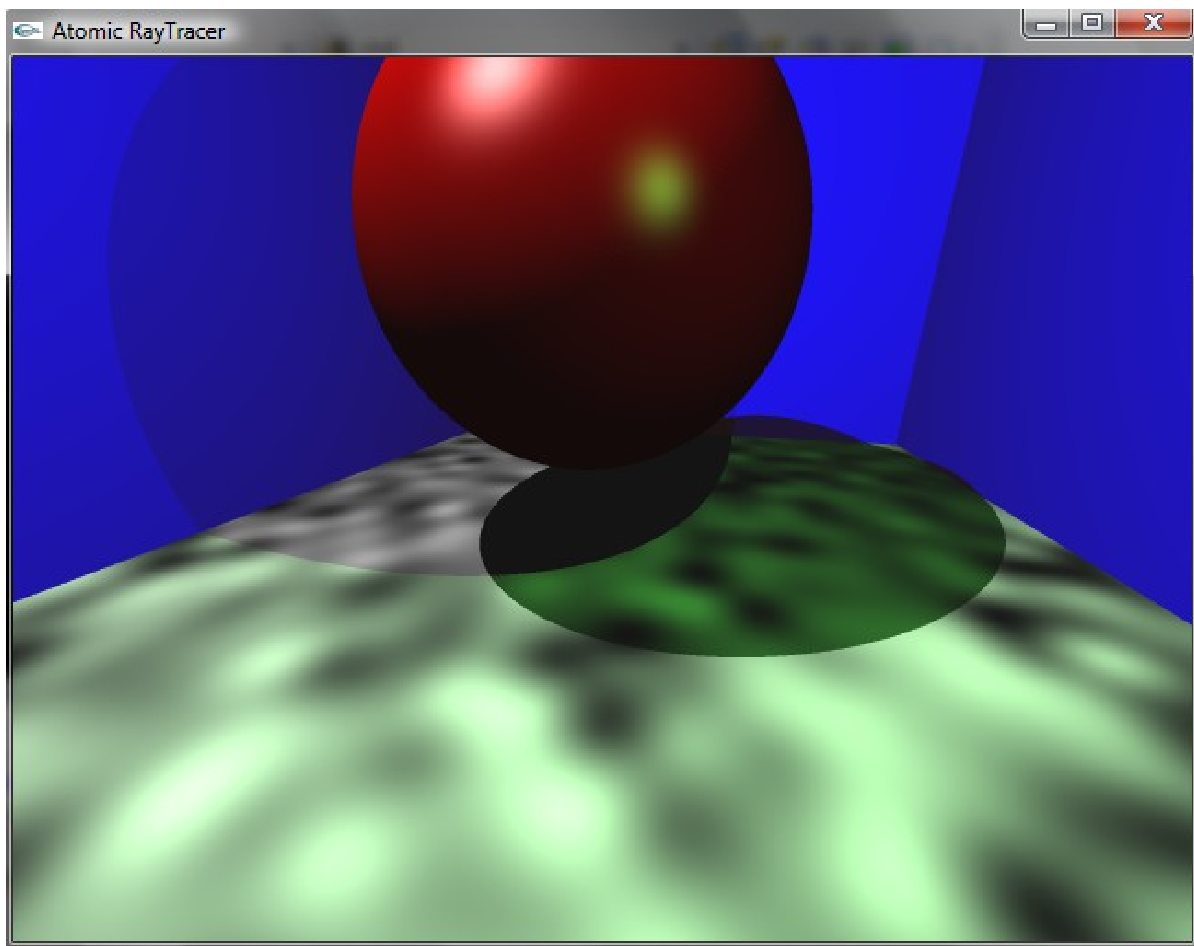
Po přeložení překladačem Vicual C++ byl čas bez optimalizace 5.62s, po optimalizaci se čas snížil na 5.57s. To znamená zrychlení o pouhé 1%. Toto ubohé urychlení je zapříčiněno tím, že metoda Intersect vrací celý kontejner Ranges<Shape>. Tento kontejner v sobě zaobaluje kontejnej vector, z důvodu dynamické alokace paměti je práce s kontejnery velice pomalá a pro raytracer nevhodná. V ostatních dvou případech existují dvě metody, Intersect a IntersectFirst. Metoda IntersectFirst vrací jen strukturu Intersection. U výpočtu průsečíku s rovinou je sice metoda Intersect implementována, používá se ale jen IntersectFirst. U koule jsou také obě metody implementovány, ale v naprosté většině případů je použito metody IntersectFirst. Pokud bychom práci s kontejnerem odstranili, docílili bychom výrazného zrychlení a většího využití SSE.

3.2.4Kompletní scény

Použil jsem dvě scény, které již v Aureliu byly implementovány, abych otestoval celkový přínos optimalizace. Jsou to následující scény:



Opět jsem přeložil program překladačem Vicual C++. Renderování scény v původní implementaci trvalo 2.31s. Optimalizovaný kód po překladu renderoval scénu 1.83s. Docílil jsem tedy urychlení o 21%.



Ilustrace 17: Test-2. Scéna

Tato scéna se po překladu Vicual C++ renderovala 2.22s bez optimalizace a 1.78s s optimalizací. To znamená 20% zrychlení.

Prováděl jsem testy i na dalších scénéch a došel jsem k závěru, že výsledné zrychlení je přibližně 25%. Toto zrychlení je určitě významé a nezanedbatelné. Zrychlení se mění v závislosti na složení scény.

3.2.5Přínos jednotlivých optimalizačních kroků

Pokusil jsem se během optimalizace zaznamenávat dílčí kroky a vyhodnocovat jejich přínos. Změřil jsem, jak velký přínos má odsunutí dělení dovnitř podmínky u výpočtu průsečíku s koulí. Výsledný program se při testu s koulemi zrychlil o 2,6%, což považuji za nezanedbatelné zrychlení.

Dále jsem zaznamenal rozdíl mezi pouhým nahrazením výpočtů za SSE instrukce a kompletní optimalizací u roviny. Konečná optimalizace se týkala především přenesení výpočtů metod struktury Vektor do SSE a odsun jednoho dělení dovnitř podmínky. Tyto činnosti měly za následek urychlení testu rovin o 4,3%. Z toho vyplývá, že tyto činnosti jsou významnou součástí optimalizace.

Odhadovat urychlení u válců je značně problematické. V tomto případě jsem použil funkci inverzní hodnoty pro dvě po sobě jdoucí dělení se stejným jmenovatelem. Výsledek je prozazatelně lepší, ale hodnota v procentech je nezměřitelná. Přínos pro optimalizaci je tedy dokázán. Problém s předáváním kontejneru ale přetrvává.

4Závěr

Prostudoval jsem metodu ray-tracing a došel jsem k závěrům, že tato metoda je velice užitečná ve filmovém průmyslu. Jako důkaz jsou filmy Toy Story, Hledá se Nemo nebo Cars. Ve světě počítačových her tato metoda memá přiliš velké uplatnění. Existuje více přístupů k ray-tracingu, firma nVidia je naprostým odpůrcem ray-tracingu jako metody pro vývoj grafiky her. Zato přístup AMD a ATI je jiný. Tyto firmy pracují na vývoji her s fotorealistickou grafikou, kde ray-tracing hraje význačnou roli.

V dnešní době hodně prosazovaná metoda paralelizace pro multi-core a narůstající počet jader je jasné plus této metody. Intel pracuje na vývoji speciálního procesoru pro urychlení výpočtu vektorových operací, tento proceror vystupuje pod názvem Intel Larrabee. Vize do budoucna tudíž hraje pro ray-tracing. Na to, jak to vše dopadne si budeme muset počkat.

Nastudoval jsem současné implementace sledování paprsku pomocí SSE. Jak je popsáno výše, jsou dva přístupy tohoto řešení. Svazky paprsků bylo vhodnější používat v minulosti ve starších verzích SSE, které v sobě neměly zakompovány horizontální operace. Dnes se spíše prosazuje nasazení SSE na práci s vektory. Nové verze jasně hrají pro optimalizaci vektorových operací.

Provedl jsem tedy návrh optimalizace jako optimalizace vektorových operací. Neno přístup považuji za vhodnější z důvodu menších zásahů do kódu. Kód tak neztrácí na své čitelnosti a je nadále přehledný.

Tento návrh jsem úspěšně implementoval. V kapitole testy jsou popsány jednotlivé testy a sejichch výsledky. Největšího urychlení jsem docílil na scéně s koulemi, toto urychlení bylo 34%. Nejhorších výsledků jsem docílil na scéně s válci, zde bylo zrychlení pouhé 1%. Je vidět, že optimalizace pomocí SSE přináší nesporné urychlení výpočtů. Musí však být správně použito tam, kde je potenciál SSE možné využít.

Bezvýznamného urychlení u válců bylo způsobeno tím, že metoda, která se stará o výpočet průsečíku, vrací kontejner. Práce s kontejnery je velice pomalá a pro implementaci raytraceru naprosto nevhodná. Z tohoto důvodu by další vývoj měl směřovat k nahrazení kontejneru za jiné efektivnější řešení.

Výsledná implementace vyžaduje ke svému spuštění verzi SSE4.1. Hned v úvodu běhu programu je přítomnost této instrukční sady kontrolována pomocí CPUIP. Dalším vylepšením by mohlo být rozšíření pro starší procesory, na kterých není SSE4.1 podporována.

Výsledný kód byl překládán v prostředí Windows překladačem Visual C++. Bylo zde použito jistých directiv, které omezují přenositelnost pro jiné překladače. Toto omezení je možné odstranit podmíněným překladem.

Program využívá pouze jednoho jádra procesoru, bylo by tedy možné do implementace vnést podporu multicore. Došlo by pak k efektivnějšímu využití hardware a snížila by se celková doba výpočtu.

Dalším námětem na vylepšení je rozšíření základních geometrických primitiv. Aurelius zpracovává pouze roviny, koule a válce. Mohli bychom provest rozšíření o nové primitivy, jako například jehlan nebo kvádr. V Aureliu je vytvořena třída pro CSG reprezentace objektů. V této třídě ale chybí metody základních logických operací, jako jsou průnik, sjednocení a rozdíl. Doimplementováním těchto metod bychom docílili možnosti sestavení daleko složitějších scén.

Dále jsem vytvořil stručný plakát demonstrující moji práci. Tento plakát obsahuje určení cílů této práce a vyhodnocení výsledků s příklady. Plakát je přibalen na DVD se zdojovými kódy.

Při vypracování této práce jsem se naučil pracovat s intrinsiky. Práce s intrinsiky je daleko lepší než práce v holém assembleru. S SSE jsem již přišel do styku v předmětu Pokročilé assemblery, kde jsem vytvářel kód přímo v assembleru. Přístup pomocí intrinsik nám usnadnuje ladění a zlepšuje přenositelnost kódu.

V úplném závěru bych tuto práci vyhodnotil jako úspěšnou. Bylo docíleno znatelné optimalizace, která je pro metodu ray-tracingu mesmírně důležitá. Při vypracování teto práce jsem čerpal mnoho užitečných znalostí z předmětů Pokročilé assemblery a Základy počítačové grafiky. Výsledky této práce by mohly být užitečné pro ty, keří se zabývají optimalizací raytacerů, tak optimalizací pomocí SSE obecně.

Literatura

[1] J. Žára, B. Beneš, J. Sochor, P. Felkel *Moderní počítačová grafika* Computer Press 2004.

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1*,November 2008

[3] *Intel® SSE4 Programming Reference*, July 2007

[4] Ing. Přemysl Kršek, Ph.D. Ing. Michal Španěl, *Slaidy do předmětu Základy Počítačové Grafiky*

[5] Chris Lomont, Fast Inverse Square Root www.lomont.org/Math/Papers/2003/InvSqrt.pdf, 2003

[6] Ing. Filip Orság, Ph.D., Pokročilé asemblery, Studijní opora, 2006

[7] AMD CodeAnalyst, www.amd.com

[8] *Microsoft Developer Network* , msdn.microsoft.com

[9] *Raytracing*, cs.wikibooks.org/wiki/Raytracing

[10] *Raytracing*, herakles.zcu.cz

[11] *OpenGL a nadstavbová knihovna GLU*, www.root.cz

[11] Články o raytracingu, www.svethardware.cz

[11] *PovRay* raytracer, www.povray.org

[12] Raytracer *Aurelius*

[13] *RayWatch* a *Simple Ray Tracer*, sourceforge.net

[14] Raytracer *Minilight* www.hxa.name/minilight

[15] Jiří Havel, *Rychlý výpočet průsečíku paprsku s trojúhelníkem*, Brno, 2008, diplomová práce FIT

Seznam příloh

Příloha 1. DVD se zdrojovými kódy