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

SLEDOVÁNÍ PAPRSKU V REÁLNÉM ČASE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE VÍT BLECHA
AUTHOR

BRNO 2007

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

SLEDOVÁNÍ PAPRSKU V REÁLNÉM ČASE

REAL-TIME RAY TRACING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — VÍT BLECHA
AUTHOR

VEDOUCÍ PRÁCE — Ing. ADAM HEROUT, Ph.D.
SUPERVISOR

BRNO 2007

Abstrakt

Ve své práci se zabývám metodou zobrazování počítačové 3D grafiky, která se nazývá sledování paprsků (anglicky ray tracing). Zaměřuji se především na zkoumání možností zobrazování trojúhelníků touto metodou. Cílem práce je studium existujících algoritmů pro výpočet průsečíku trojúhelníku a paprsku, jejich vzájemné srovnání, jejich výpočetní náročnost a rychlost na různých počítačových architekturách. Tyto algoritmy také implementuji do existujícího ray traceru pracujícího v reálném čase.

Klíčová slova

Sledování paprsku, výpočet průsečíku paprsku a trojúhelníku, počítačová grafika.

Abstract

My thesis deal with method of rendering computer graphics called ray tracing. I will aim on possibility of rendering triangles using this method. Goal of my work is study of existing algorithms to test ray-triangle intersection, their comparison, their computing demandingness and speed on different pc architectures. I will also implement these algorithms to existing ray tracer working in real-time.

Keywords

Ray tracing, calculating ray-triangle intersection, computer graphics.

Citace

Vít Blecha: Sledování paprsku v reálném čase, bakalářská práce, Brno, FIT VUT v Brně, 2007

Sledování paprsku v reálném čase

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Adama Herouta, Ph.D. a uvedl jsem veškeré materiály, které jsem při psaní práce použil.

. . . . . .
Vít Blecha
14. května 2007

Poděkování

Děkuji tímto panu Heroutovi za odborné rady a konstruktivní připomínky, které mi při psaní práce velmi pomohly.

Zadání

Sledování paprsku v reálném čase

Real-Time Ray Tracing

Vedoucí:
Herout Adam, Ing., Ph.D., UPGM FIT VUT
Oponent:
Sumec Stanislav, Ing., Ph.D., UPGM FIT VUT

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

Kategorie:
Počítačová grafika
Literatura:
* dle pokynů vedoucího

Licenční smlouva

Licenční smlouva je uložena v archívu Fakulty informačních technologií Vysokého učení technického v Brně.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá zobrazováním počítačové 3D grafiky metodou sledování paprsků (dále ray tracing). Jedná se o metodu, která je velmi náročná, ale dosahuje velmi dobrých vizuálních výsledků, které jsou jinými metodami težko dosažitelné. Velmi zjedodušenému popisu této metody, jejímu úskalí i hlavním výhodám, věnuji druhou kapitolu své práce.

Stěžejní předmět této práce je zkoumání zobrazování trojúhelníků metodou ray tracingu. V dnešní počítačové grafice je trojúhelník dominantní zobrazovaný prvek, případně objekty, které jsou z trojúhelníků složeny. Možnost jejich zobrazení je proto pro použitelnost této metody velmi důležitá. Aby bylo možné pomocí ray tracingu trojúhelníky vykreslovat, je třeba určit, zda paprsek vyslaný do scény protne daný trojúhelník. Existuje několik možných přístupů, různě efektivních, jak tento průsečík určit. Vybral jsem si dva existující algoritmy: algoritmus popsaný Tomasem Möllerem a Benem Trumborem [6] a algoritmus Didiera Badouela [1], které průsečík paprsku s trojúhelníkem počítají. Tyto algoritmy popíšu ve druhé kapitole.

Třetí kapitola je pak zaměřena na srovnání těchto dvou metod a dále s algoritmem Nicka Chirkova [2]. Tuto kapitolu považuji za nejdůležitější, protože obsahuje nejrůznější testy a porovnání jednotlivých algoritmů. Výsledkem práce bude zhodnocení algoritmů, určení který je výkonnější za jakých podmínek. Ze získaných vědomostí půjde říct, co by ještě více mohlo přiblížit metodu sledování paprsků běžnému používání v real-timovém zobrazování.

Závěrečná kapitola bude věnována implementaci těchto algoritmů do existujícího ray traceru, takže bude možné ověřit získané znalosti i v praxi.

Kapitola 2

Ray tracing

Jedná se o metodu zobrazování 3D scén v počítačové grafice. Tato metoda je založena na principu sledování světelného paprsku, který prochází danou scénou. Díky tomuto přístupu je tak možné zobrazit jevy, které jsou jinými metodami velmi těžko zobrazitelné, jako například odrazy objektů nebo lom světla. Bohužel velmi vysoká kvalita výsledku je úměrná velké náročnosti této metody. Přesto díky vysoké výkonnosti dnešních počítačů je již možné zobrazovat některé scény ray tracingem v reálném čase. Případně je možné scénu počítat současně na několika počítačích a zde je již dosaženo velmi kvalitních obrazových výsledků ze složitých scén spočítaných v reálném čase. Touto problematikou se zabývá celá řada vývojových týmů a na výzkumu se podílí mnoho významných světových universit. Scénu vykreslenou pomocí ray tracingu si můžeme prohlédnout na obrázku 2.1

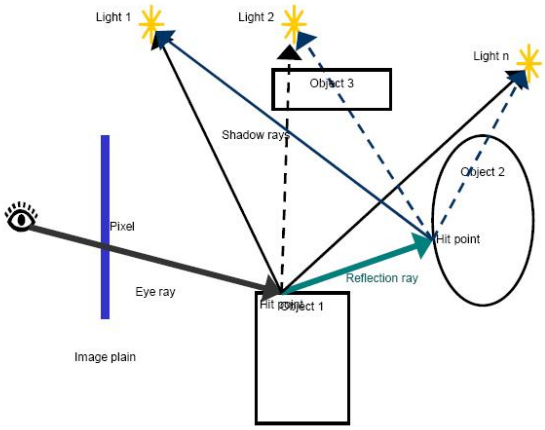
2.1 Základní princip

Myšlenka této metody spočívá ve sledování světelných paprsků scénou. To obnáší z každého světelného zdroje vyslat paprsky do všech směrů a sledovat jejich let. Paprsky interagují s objekty ve scéně, při jejich zasažení nabírají barevnou informaci a mohou se na jejich povrchu odrážet, lámat nebo zcela zanikat. Výsledný obraz je pak získán sečtením paprsků, které vstupují do kamery. Toto řešení je pro praxi ale velice náročné, neboť není předem možné určit, které paprsky se dostanou do kamery a ovlivní tak výsledný obraz. Vyslání všech paprsků by bylo výpočetně velice náročné. Proto se v praxi používá takzvané zpětné sledování paprsků, které bude v dalším textu označováno čistě jako sledování paprsků nebo ray tracing.

Základní princip této metody spočívá v tom, že se do scény pro každý zobrazovaný pixel vysílá paprsek a je sledována jeho dráha a interakce s objekty ve scéně. Tyto paprsky se nazývají primární. Pro každý takto vyslaný paprsek je třeba určit, zda protíná nějaký objekt. Pokud do nějakého objektu narazí, je z tohoto místa vyslán paprsek ke každému světlu ve scéně, aby bylo možné určit, zda je objekt tímto světlem osvětlen nebo nikoliv. Na základě tohoto zjištění je pro daný bod zaznamenána barevná hodnota, která je dána vlastnostmi povrchu objektu a případného osvětlení. Dále se z tohoto bodu generují dva nové paprsky, takzvaně paprsky sekundární. Jsou to paprsky odrazu a lomu. Vznik těchto paprsků může být povrchovými vlastnostmi objektu potlačen. Tyto paprsky se dále zpracovávají stejně jako paprsek primární. Aby se takto paprsky negenerovaly donekonečna, je třeba zvolit určitou hloubku rekurze, po jejímž dosažení se již pro paprsek sekundární paprsky nebudou generovat. Každý paprsek tak přinese část barevné informace, která se

Obrázek 2.1: Ray tracing, [7]

postupně skládá a nakonec určí jaká barva se má pro daný pixel zobrazit. Základní princip sledování paprsků je dobře vidět na obrázku 2.2, který jsem převzal z [3].



Obrázek 2.2: Sledování paprsků

2.2 Objekty ve scéně

Abychom mohli zobrazovat nějakou scénu, musíme mít danou její reprezentaci. Protože metoda sledování paprsků sleduje dráhu jednotlivých paprsků v prostoru, je nejvhodnější mít objekty popsané matematicky. Tak je jednoduché počítat průsečík tohoto objektu s paprskem. Například koule může být popsána jednou rovnicí a vyřešení průsečíku je vcelku jednoduché. Pokud však chceme zobrazovat složitější scény a objekty různých tvarů, je nemožné tyto objekty nějak jednoduše popsat matematickými rovnicemi. Je však možné tyto objekty složit z mnoha trojúhelníků, což je postup, který je v dnešní grafice zcela standardní. Trojúhelník ale nemá tak jednoduché matematické vyjádření jako třeba koule, proto je počítání průsečíku paprsku a trojúhelníku o něco těžší. Existuje celá řada algoritmů, které se touto problematikou zabývají, protože sledování paprsků se věnuje celá řada světových universit a jiných výzkumných týmů. Za předmět svojí práce jsem si zvolil vyzkoušet některé z těchto algoritmů a provést jejich srovnání.

2.3 Výhody a nevýhody

Jednoznačně největší výhodou této metody zobrazování je její vysoká realističnost. V podstatě metoda simuluje, jak se světelné paprsky šíří scénou a může tak zobrazovat optické jevy jako například lom světla nebo odrazy. Velmi dobře také umožňuje vykreslovat stíny objektů. Nevýhodou je, že je tato metoda velmi výpočetně náročná, protože se musí vytvářet velké množství paprsků a počítání jejich průsečíků se všemi objekty ve scéně není také jednoduché. Třeba se ale brzy dočkáme grafických karet, které budou hardwarově sledování paprsků podporovat a tato metoda se tak stane běžnou zobrazovací metodou pracující v reálném čase.

Kapitola 3

Určování průsečíku paprsku s trojúhelníkem

Pokud chceme zobrazovat trojúhelníky metodou sledování paprsků, je nejdůležitější vypočítat průsečík paprsku s trojúhelníkem a zjistit vzdálenost tohoto průsečíku, případně také určit bod na trojúhelníku, ve kterém k průsečíku došlo. Tento výpočet není tak jednoduchý jako výpočet průsečíku paprsku s koulí, protože trojúhelník nemá tak jednoduchou geometrickou reprezentaci. Postupů řešících tuto problematiku je více a liší se jak výkonností (rychlostí výpočtu), tak také paměťovými nároky. Já jsem si pro svoje zkoumání vybral dva algoritmy, o kterých jsem četl, že by měly být oba velmi efektivní. Dalším důvodem proč jsem si vybral právě tyto dva, byl jejich odlišný přístup k problematice.

3.1 Metoda Möller, Trumbore

Tato metoda byla popsána v [6]. Jedná se o metodu, která je málo náročná na paměť. K určení průsečíku je třeba znát pouze polohu tří vrcholů trojúhelníku a počáteční bod a směrový vektor paprsku. Narozdíl od většiny ostatních metod nezkoumá jako první, zda paprsek vůbec protíná rovinu, ve které trojúhelník leží.

3.1.1 Algoritmus

Nechť je paprsek $R(t)$ s počátečním bodem O a normalizovaným směrovým vektorem D definován jako

$$R(t) = O \ + tD, \tag{3.1}$$

t je parametr paprsku, který určuje vzdálenost průsečíku od počátku paprsku. Trojúhelník je určen svými třemi vrcholy $V0, V1, V2$.

Bod $T(u, v)$ na trojúhelníku je dán jako

$$T(u, v) = (1 - u \ - v)V_0 + uV_1 + vV_2, \tag{3.2}$$

kde (u, v) jsou barycentrické souřadnice, pro které musí platit $u \geq 0, v \geq 0$ a $u + v \leq 1$. Souřadnice (u, v) mohou být použité například pro mapování textur nebo interpolaci barvy. Spočítání průsečíku paprsku $R(t)$ a trojúhelníku $T(u, v)$ je ekvivalentní $R(t) = T(u, v)$, což znemená

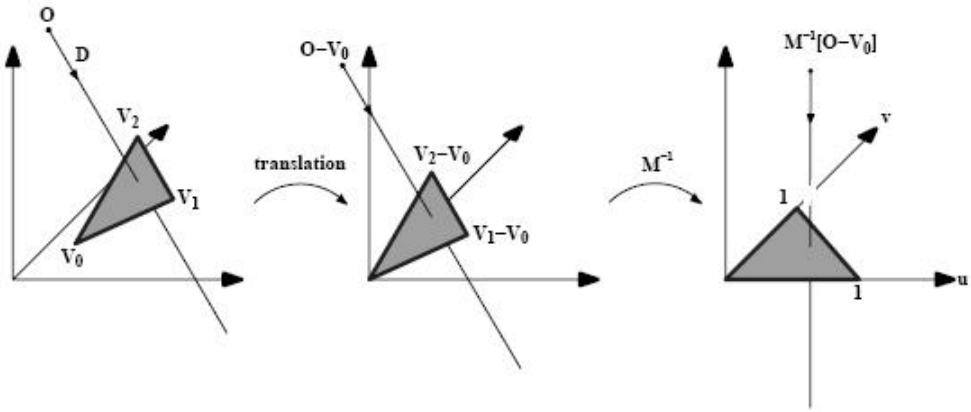
$$O \ + tD = (1 - u \ - v)V_0 + uV_1 + vV_2, \tag{3.3}$$

po úpravě rovnice 3.3 dostáváme

$$[-D, V_1 - V_0, V_2 - V_0] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = 0 - V_0. \tag{3.4}$$

Parametr t a barycentrické souřadnice (u, v) tedy mohou být nalezeny vyřešením těchto rovnic.

Geometricky to můžeme chápat jako přesunutí trojúhelníku do počátku souřadného systému a jeho transformaci do jednotkového trojúhelníku v osách y a z, jak znázorňuje obrázek 3.1 (kde $M = [-D, V_1 - V_0, V_2 - V_0]$).



Obrázek 3.1: Přesunutí a transformace trojúhelníku

S nahrazením $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ a $T = O - V_0$ v rovnici 3.4 získáme řešení použitím Cramerova pravidla

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-D, E_1, E_2|} \begin{bmatrix} |T,\ E_1,\ E_2| \\ |-D, T, E_2| \\ |-D, E_1, T| \end{bmatrix}. \tag{3.5}$$

Ze znalosti že $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$. Naše rovnice 3.5 tak může být přepsána jako

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (D \times E_2) \cdot T \\ (T \times E_1) \cdot D \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}, \tag{3.6}$$

kde $P = (D \times E_2)$ a $Q = T \times E_1$.

Existují různé modifikace tohoto základního algoritmu, které mohou zvýšit jeho výkon. O variantě a konkrétní implementaci, kterou jsem se rozhodl použít já, napíši víc v kapitole 4.5.1, kde je možné nalézt kód zapsaný v jazyce C.

3.2 Metoda Badouel

Tento algoritmus D. Badouela byl prezentován v [1]. Narozdíl od předchozího algoritmu je tento náročnější na paměť, mimo tří vrcholů trojúhelníku používá také charakteristiky roviny, ve které se trojúhelník nachází.

3.2.1 Algoritmus

Stejně jako u předchozí metody mějme paprsek $R(t)$ s počátkem v O a normalizovaným směrovým vektorem D definovaný jako

$$R(t) = O \ + tD. \tag{3.7}$$

Trojúhelník je popsán svými třemi vrcholy V_0, V_1, V_2. Každý vrchol V_i je určen souřadnicemi x_i, y_i a z_i. Normála roviny obsahující tento trojúhelník je spočítána vektorovým součinem

$$\overrightarrow{N} = \overrightarrow{V_0V_1} \times \overrightarrow{V_0V_2} \tag{3.8}$$

a uložena jako jedna z charakteristik trojúhelníku. Vektor $\overrightarrow{V_0V_1}$ reprezentuje hranu trojúhelníku (obdobně druhý). Pro každý bod P roviny platí, že skalární součin $P \cdot N$ je konstatní. Tato konstanta je spočítána skalárním součinem $d = -V_0 \cdot N$. Implicitní reprezentace roviny

$$N \cdot P + d = 0, \tag{3.9}$$

je spočítána jednou a také uložena jako jedna z charakteristik trojúhelníku.

Pokud je paprsek rovnoběžný s rovinou trojúhelníku, je skalární součin jeho normály a směru paprsku roven nule

$$N \cdot D = 0, \tag{3.10}$$

v takovém případě paprsek trojúhelník neprotíná a výpočet končí.

Vyjádření parametru t pro průsečík z předchozích rovnic dostaneme jako

$$t = -\frac{d + N \cdot O}{N \cdot D}. \tag{3.11}$$

Nyní, když víme zda paprsek protíná rovinu, můžeme určit, zda je tento průsečík uvnitř trojúhelníku. Mějme bod P dán jako (viz. obrázek 3.2)

$$\overrightarrow{V_0P} = \alpha\overrightarrow{V_0V_1} + \beta\overrightarrow{V_0V_2}. \tag{3.12}$$

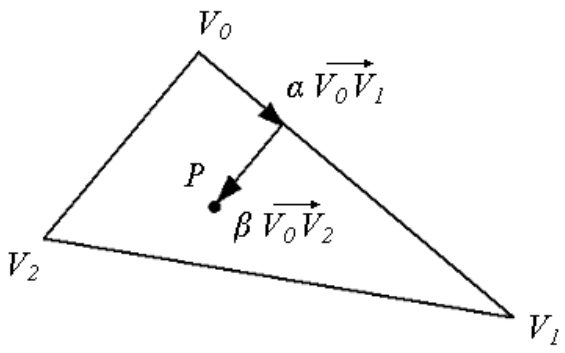
Bod P bude ležet uvnitř trojúhelníku, pokud

$$\alpha \geq 0, \beta \geq 0 \text{ a } \alpha + \beta \leq 1. \tag{3.13}$$

Rovnice 3.12 má tři komponenty:

$$\begin{cases} x_P - x_0 = \alpha(x_1 - x_0) + \beta(x_2 - x_0) \\ y_P - y_0 = \alpha(y_1 - y_0) + \beta(y_2 - y_0) \\ z_P - z_0 = \alpha(z_1 - z_0) + \beta(z_2 - z_0) \end{cases}. \tag{3.14}$$

Tato soustava má řešení, které je jedinečné. Abychom soustavu zjednodušili, zobrazíme trojúhelník na jednu ze základních rovin (buď xy, xz nebo yz). Pokud by trojúhelník byl kolmý na jednu z těchto rovin a zobrazil by se na ni, zobrazil by se jako úsečka. Abychom



Obrázek 3.2: Parametrické vyjádření bodu P v trojúhelníku

se tomuto možnému problému vyhnuli, nalezneme dominantní osu normálového vektoru a použijeme rovinu kolmou k této ose.

$$i_0 = \begin{cases} 0 & \textit{pokud } |N_x| = \max(|N_x|, |N_y|, |N_z|) \\ 1 & \textit{pokud } |N_y| = \max(|N_x|, |N_y|, |N_z|) \\ 2 & \textit{pokud } |N_z| = \max(|N_x|, |N_y|, |N_z|) \end{cases} . \tag{3.15}$$

Považujme i_1 a i_2 (i_1 a $i_2 \in \{0, 1, 2\}$) za příznaky různé od i_0. Představují rovinu, na kterou bude trojúhelník promítán. Nechť jsou (u, v) dvourozměrné souřadnice vektoru v této rovině. Souřadnice vektorů $\overrightarrow{V_0P}, \overrightarrow{V_0V_1}a\overrightarrow{V_0V_2}$ zobrazené na tuto rovinu budou

$$\begin{matrix} u_0 = P_{i_1} - V_{0i_1} & u_1 = V_{1i_1} - V_{0i_1} & u_2 = V_{2i_1} - V_{0i_1} \\ v_0 = P_{i_2} - V_{0i_2} & v_1 = V_{1i_2} - V_{0i_2} & v_2 = V_{2i_2} - V_{0i_2} \end{matrix} . \tag{3.16}$$

Soustava 3.14 se tak zjednodušší na

$$\begin{cases} u_0 = \alpha \cdot u_1 + \beta \cdot u_2 \\ v_0 = \alpha \cdot v_1 + \beta \cdot v_2 \end{cases} . \tag{3.17}$$

Řešením tedy jsou

$$\alpha = \frac{\det \begin{pmatrix} u_0 & u_2 \\ v_0 & v_2 \end{pmatrix}}{\det \begin{pmatrix} u_1 & u_2 \\ v_1 & v_2 \end{pmatrix}} \quad \text{a} \quad \beta = \frac{\det \begin{pmatrix} u_1 & u_0 \\ v_1 & v_0 \end{pmatrix}}{\det \begin{pmatrix} u_1 & u_2 \\ v_1 & v_2 \end{pmatrix}}. \tag{3.18}$$

Tím získáme barycentrické souřadnice α a β a můžeme určit 3.13, zda paprsek protíná trojúhelník. Konkrétní implementace v jazyce C je uvedena v kapitole 4.5.2.

Kapitola 4

Experimenty a srovnávání algoritmů

V této kapitole se budu zabývat srovnáním a testováním dvou prezentovaných algoritmů. Při zkoumání různých zdrojů, jsem v testovacím programu [5] narazil na metodu, která byla rychlejší než jedna z metod (druhá nebyla v tomto testu prezentována), kterými jsem se zabýval. Je to algoritmus Nicka Chirkova [2]. Rozhodl jsem se, že tento algoritmus do svého testovacího programu implementuji také, aby byly poznatky zajímavější. Pro zjednodušení budu v této kapitole jednotlivé algoritmy pojmenovávat Möller, Badouel a Chirkov.

Abych mohl algoritmy srovnávat, napsal jsem si vlastní testovací program (je obsažen na přiloženém cd ve složce ray_test). Program je psán v jazyce C++ ve vývojovém prostředí Microsoft Visual Studio 2005. Pro měření času jsem využil knihovnu `windows.h`, konkrétně funkci `GetTickCount ()`, která funguje s přesností v řádu milisekund. Přesto aby bylo možné takto funkce měřit, musejí být volány v cyklu mnohokrát za sebou. Všechna měření budou prováděna v operačním systému Windows XP. Aby bylo dosaženo co největší přesnosti a zabráněno vlivu ostatních procesů, jsou veškeré testovací procesy spouštěny s prioritou real-time. Pro generování trojúhelníků a paprsků, je využita knihovna `cstdlib`. Všechny hodnoty jsou generovány pomocí funkce `rand ()` v rozsahu $\langle 0, 1 \rangle$. Před každou skupinou testů jsou vygenerované trojúhelníky a paprsky uloženy do pole a pro všechny tři algoritmy použity stejně, takže mají stejné podmínky.

S přihlédnutím k tomu, co budu potřebovat k implementaci do ray traceru, jsem zvolil varianty algoritmů, které počítají pouze parametr t. Trochu v nevýhodě se tak ocitá algorimus Badouel, který počítá i souřadnice průsečíku.

Při uvádění typu pc je uvedeno: označení procesoru (takt jádra, označení jádra, velikost L1 cache (data / kód), velikost L2 cache), velikost operační paměti (typ, frekvence).

V popisu měření je udáno, kolik je opakováno výpočtů pro jeden test s kolika různými trojúhelníky. 25 000 trojúhelníků a 20 000 opakování tedy znamená 500 000 000 výpočtů pro jedno měření. Hustota zasažení (v tabulkách pouze zasažení) udává, kolik procent trojúhelníků z náhodně generované skupiny bylo paprskem zasaženo (0,25 znamená že bylo paprskem zasaženo právě 25 % trojúhelníků), protože to, jak uvidíme, má vliv na výsledek měření.

Sloupec tabulky Efektivnější udává (pokud je uveden), o kolik procent je údaj v daném řádku tabulky pomalejší než nejlepší varianta. Řádek s 0 % je v daném testu nejlepší.

Většina testů je doplněna znázorňujícím grafem, někdy je však kvůli sazbě až na další stránce.

4.1 Výběr varianty algoritmu Möller

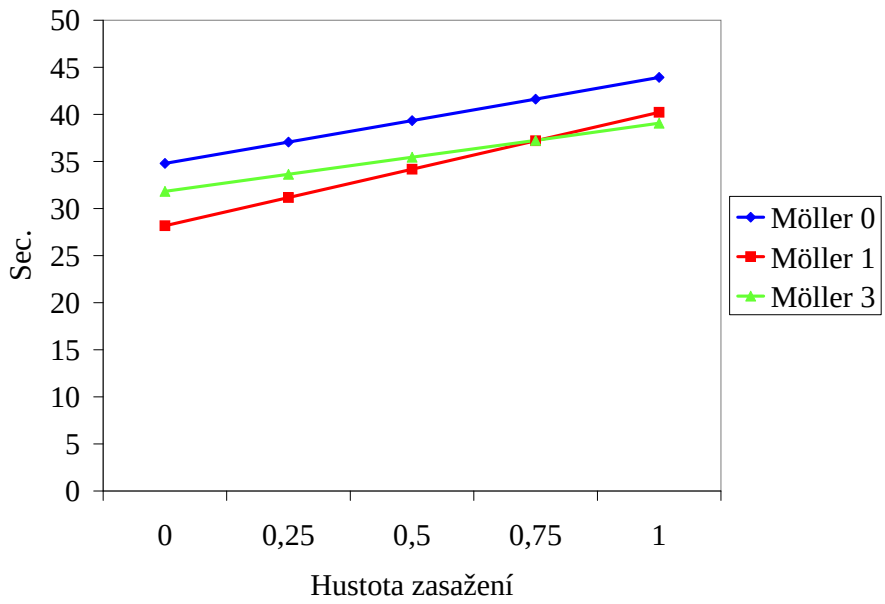
Motivace: Protože variant algoritmu Möller jsem objevil několik, provedu jako první test jejich srovnání a do dalších testů zahrnu pouze variantu nejlepší. Varianty se liší hlavně umístěním dělení.

Použité pc: AMD Athlon 64 3000+ (1,8 GHz, Winchester, L1 (64 KB / 64 KB), L2 512 KB), 1 GB RAM (PC3200, 200 MHz).

Měření: 25 000 trojúhelníků při 20 000 opakování. Sloupce tabulky udávají různé hustoty zasažení. Výsledné hodnoty jsou počet sekund potřebný k danému počtu výpočtů.

Varianta	0,0	0,25	0,5	0,75	1,0	Efektivnější
Möller0	34,797	37,062	39,344	41,625	43,922	15,1 %
Möller1	28,172	31,172	34,172	37,203	40,219	0 %
Möller3	31,828	33,640	35,453	37,236	39,062	3,7 %

Tabulka 4.1: Různé varianty algoritmu Möller (4.1)



Obrázek 4.1: Graf testu (4.1)

Výsledek

Jak vidíme, tak většinou nejlepší výsledky dosáhla varianta 1, která provádí dělení až úplně na konci algoritmu. Rozhodl jsem se proto do dalších testů imlementovat metodu tuto (více o konkrétní implementaci v části 4.5.1). Zajímavé je, že varianta 3 je více efektivní s rostoucí hustotou zasažení. Rychlejší než varianta 1 je ale až v posledním případě. Pokud se chcete na jednotlivé varianty podívat, jsou k nalezení v [5].

4.2 Vliv hustoty zasažení na výkon algoritmu

Motivace: Přišel čas konečně vyzkoušet tři zvolené algoritmy mezi sebou. Jak jsme mohli vidět z předchozího testu, je výkon algoritmu závislý na procentu zasažených trojúhelníků. V tomto testu tedy provedeme měření pro různé hustoty s krokem 0,1.

Před testem jsem očekával, že algoritmus Badouel se ukáže jako výkonnější, protože používá některé předpočítané hodnoty, zatímco algoritmus Möller si vystačí s minimem zadaných hodnot.

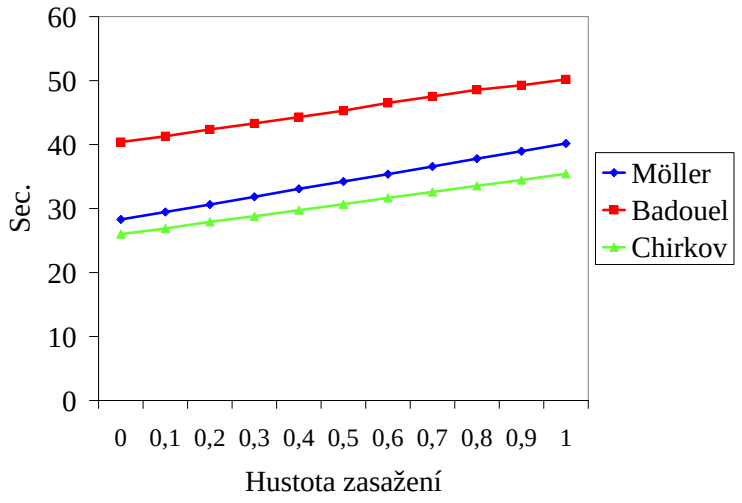
Použité pc: AMD Athlon 64 3000+ (1,8 GHz, Winchester, L1 (64 KB / 64 KB), L2 512 KB), 1 GB RAM (PC3200, 200 MHz).

Měření: 25 000 trojúhelníků při 20 000 opakování. Sloupce tabulky udávají různé hustoty zasažení. Výsledné hodnoty jsou počet sekund potřebných k danému počtu výpočtů.

Algorimus	0,0	0,1	0,2	0,3	0,4	0,5
Möller	28,297	29,453	30,625	31,828	33,078	34,234
Badouel	40,391	41,297	42,359	43,313	44,297	45,297
Chirkov	26,016	26,890	27,921	28,797	29,750	30,672

Algorimus	0,6	0,7	0,8	0,9	1,0	Efek.
Möller	35,375	36,562	37,796	38,969	40,172	11,4 %
Badouel	46,500	47,516	48,563	49,265	50,172	47,7 %
Chirkov	31,672	32,593	33,562	34,469	35,438	0 %

Tabulka 4.2: Různé hustoty zasažení (4.2)



Obrázek 4.2: Graf testu (4.2)

Výsledek

Vcelku značný propad algoritmu Badouel byl pro mě překvapením. Algoritmus Chirkov se ukázal jako opravdu nejrychlejší. Přičemž s rostoucí hustotou zasažení zvyšoval svůj náskok oproti metodě Möller. Jinak je vidět, že rychlost algoritmů klesá přibližně konstantně.

4.3 Různé počítače

Motivace: Porovnáme opět všechny tři algoritmy, tentokrát mi jde o zjištění, zda se výsledky předchozího testu potvrdí i výpočtem na jiných počítačích.

Použité pc1: AMD Athlon 64 3000+ (1,8 GHz, Winchester, L1 (64 KB / 64 KB), L2 512 KB), 1 GB RAM (PC3200, 200 MHz).

Použité pc2: AMD Athlon 64 3500+ (2,2 GHz, Venice, L1 (64 KB / 64 KB), L2 512 KB), 512 MB RAM.

Použité pc3: Intel Core 2 Duo E6300 (1,6 GHz, Conroe, L1 (2 x 32 KB / 2 x 32 KB), L2 2 MB, 2 GB RAM (PC5300, 333 MHz).

Použité pc4: Intel Core 2 Duo E6600 (2,4 GHz, Conroe, L1 (2 x 32 KB / 2 x 32 KB), L2 4 MB, 1 GB RAM.

Měření: 25 000 trojúhelníků při 20 000 opakování na jedno měření. Testovány byly hustoty zasažení 0, 0,25, 0,5, 0,75 a 1. Tabulka ukazuje průměrnou hodnotu sekund potřebných daným pc k výpočtu těchto skupin trojúhelníků.

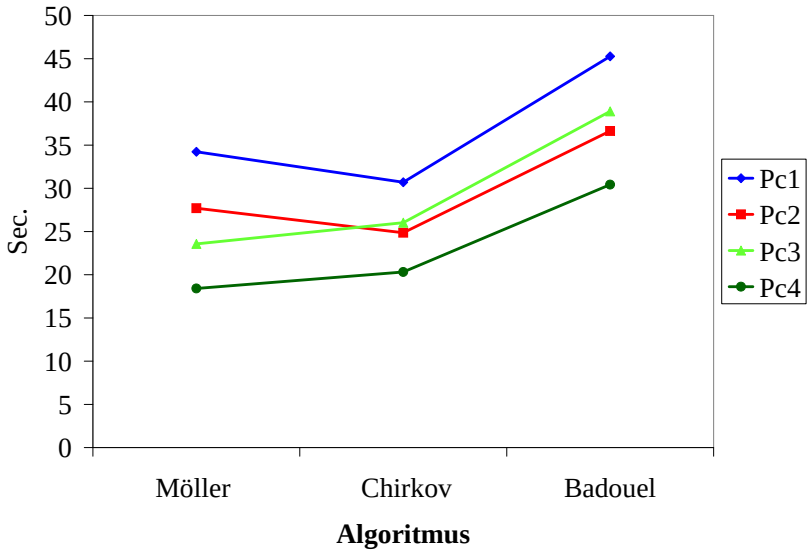
Poznámka: V grafu jsem přehodil sloupce tak, aby algoritmy Möller a Chirkov byly vedle sebe, proto aby zjištěná skutečnost byla lépe patrná.

Pc	Möller	Badouel	Chirkov
Pc1	34,228	45,268	30,703
Pc2	27,694	36,628	24,856
Pc3	23,562	38,907	26,024
Pc4	18,409	30,422	20,319

Tabulka 4.3: Srovnání na různých pc (4.3)

Výsledek

Tento výsledek pro mě byl ještě větším překvapením než výsledek předchozího testu. Očekával jsem, že na různých pc budou výpočty různě rychlé. Ale ukázalo se, že na počítačích Intel je nejrychlejší jiná metoda než na počítačích AMD. Dokonce vidíme, že na pc3 je metoda Möller rychlejší než na pc2, které má druhé dvě rychlejší. V následujícím testu se pokusím zjistit proč.



Obrázek 4.3: Graf testu (4.3)

4.4 Příčina rozdílných výsledků z testu 4.3

Motivace: Z předchozího testu se ukázalo, že na počítačích Intel je nejrychlejší metoda Möller, kdežto na počítačích AMD algoritmus Chirkov. Proč? Možných příčin mě napadá několik. Buď je to tím, že procesory od Intelu jsou vybaveny větší L2 cache pamětí, rychlejší operační pamětí, nebo je možné, že zvládají podstatně rychleji nějakou operaci.

Použité pc1: AMD Athlon 64 3500+ (2,2 GHz, Venice, L1 (64 KB / 64 KB), L2 512 KB), 512 MB RAM.

Použité pc2: Intel Core 2 Duo E6300 (1,6 GHz, Conroe, L1 (2 x 32 KB / 2 x 32 KB), L2 2 MB, 2 GB RAM (PC5300, 333 MHz).

Měření: Zjistím, kolik procent času výpočtu zaberou jednotlivým procesorům operace skalárního a vektorového součinu, rozdílu vektorů, dělení a rozhodování podmínek. Pokud budou tyto hodnoty na pc AMD i Intel přibližně stejné, znamená to, že musíme příčinu tohoto rozdílu hledat jinde.

Hodnoty získané měřením jsou celkem nezajímavé pro tabulku, takže pouze ty zajímavé skutečnosti popíši ve výsledku.

Výsledek

Z mého měření vyšlo najevo, že procesory Intel zvládají rychleji rozhodování podmínek. Při zkoušení na algoritmu Möller byl tento rozdíl u poslední podmínky dokonce 6 % z celkové doby výpočtu. Protože metoda Chirkov má podmínky pouze dvě, zatímco Möller až čtyři, je tedy tato skutečnost patrnou příčinou tohoto rozdílu. V následující části se na jednotlivé algoritmy podíváme podrobněji a zjistíme, jestli je za tímto rozdílem i něco jiného, nebo pouze schopnost rychleji rozhodovat podmínky.

4.5 Analýza kódů

V této sekci se podíváme na všechny algoritmy jednotlivě a trochu je rozebereme. Kódy jsou zapsány v jazyce C.

Společné definice

Všechny funkce mají společné parametry `tri, ray, t`, které popisují trojúhelník, paprsek a parametr paprsku určující průsečík. Jednotlivé typy jsou definovány takto:

`TRIANGLE:` struktura, obsahuje pole `v0[3], v1[3], v2[3]`, která popisují jednotlivé vrcholy trojúhelníku (každé je tvořeno souřadnicí x, y a z). Další složkou je pole `normal[3]`, které popisuje normálu. `float d` je vzdálenost roviny trojúhelníku od počátku souřadného systému. Poslední tři složky `int i0, i1, i2` vyjadřují, která základní rovina je pro trojúhelník dominantní.

`RAY:` struktura, obsahuje tři pole, každé popisuje souřadnici x, y a z. `orig[3]` je počátek paprsku, `end[3]` konec a `dir[3]` je normalizovaný vektor určující směr paprsku.

`float t:` určuje parametr paprsku pro průsečík (pokud k němu dojde).

`EPSILON:` konstanta blízká nule, v našich algoritmech `EPSILON = 0.000001`.

Funkce vrací hodnotu 0, pokud paprsek trojúhelník protne, pokud ne, vrací se hodnota větší než nula. Standardně je tomu naopak, já to takto zavedl pouze, aby bylo možné demonstrovat některá měření.

4.5.1 Möller

Algoritmus

```
int moller (TRIANGLE *tri, RAY *ray, float *t) {
    float e1x = tri->v1[0] - tri->v0[0];
    float e1y = tri->v1[1] - tri->v0[1];
    float e1z = tri->v1[2] - tri->v0[2];
    float e2x = tri->v2[0] - tri->v0[0];
    float e2y = tri->v2[1] - tri->v0[1];
    float e2z = tri->v2[2] - tri->v0[2];

    float pvx = ray->dir[1] * e2z - ray->dir[2] * e2y;
    float pvy = ray->dir[2] * e2x - ray->dir[0] * e2z;
    float pvz = ray->dir[0] * e2y - ray->dir[1] * e2x;

    float det = e1x * pvx + e1y * pvy + e1z * pvz;

    float qvx, qvy, qvz;
    if (det > EPSILON) {
        float tvx = ray->orig[0] - tri->v0[0];
        float tvy = ray->orig[1] - tri->v0[1];
        float tvz = ray->orig[2] - tri->v0[2];
```

```
        float u = tvx * pvx + tvy * pvy + tvz * pvz;
        if (u < 0.0 || u > det)
            return 1;

        qvx = tvy * e1z - tvz * e1y;
        qvy = tvz * e1x - tvx * e1z;
        qvz = tvx * e1y - tvy * e1x;

        float v = ray->dir[0] * qvx + ray->dir[1] * qvy + ray->dir[2] * qvz;
        if (v < 0.0 || u + v > det)
            return 2;
    } else if (det < -EPSILON) {
        float tvx = ray->orig[0] - tri->v0[0];
        float tvy = ray->orig[1] - tri->v0[1];
        float tvz = ray->orig[2] - tri->v0[2];

        float u = tvx * pvx + tvy * pvy + tvz * pvz;
        if (u > 0.0 || u < det)
            return 3;

        qvx = tvy * e1z - tvz * e1y;
        qvy = tvz * e1x - tvx * e1z;
        qvz = tvx * e1y - tvy * e1x;

        float v = ray->dir[0] * qvx + ray->dir[1] * qvy + ray->dir[2] * qvz;
        if (v > 0.0 || u + v < det)
            return 4;
    } else
        return 5;

    float inv_det = 1.0f / det;
    *t = (e2x * qvx + e2y * qvy + e2z * qvz) *  inv_det;

    return 0;
}
```

Četnost použitých operací

V tabulce je uveden maximální počet kolikrát může být daná operace volána.

+	-	*	/	Podmínka
9	16	25	1	4

Tabulka 4.4: Četnost operací metody Möller

Z těchto operací jsou celkem 3 rozdíly vektorů, 2 vektorové a 4 skalární součiny vektorů.

Dosažení různých návratových hodnot

Pro zajímavost se podíváme, jak často se funkce vrací z jednotlivých možností. Pro tento test bylo vygenerováno 1 000 000 trojúhelníků a byl proveden na pc AMD Athlon 64 3000+. Tabulka ukazuje procentuální zastoupení dané návratové hodnoty (viz. kód na začátku sekce 4.5.1) a také čas potřebný k dosažení těchto hodnot (toto měření počítá 500 000 000 výpočtů pro skupinu trojúhelníků, které s příslušnými paprsky vrací danou hodnotu).

Návratová hodnota	Četnost / procent	Doba pro výpočet (sec.)
0	120 720 / 12,07 %	40,188
1	317 737 / 31,77 %	24,375
2	121 256 / 12,13 %	35,250
3	318 641 / 31,86 %	26,422
4	121 632 / 12,16 %	36,500
5	14 / 0,01 %	19,625

Tabulka 4.5: Návratové hodnoty metody Möller

4.5.2 Badouel

Algoritmus

```
int badouel (TRIANGLE *tri, RAY *ray, float *t) {
    float dot = ray->dir[0] * tri->normal[0] + ray->dir[1] * tri->normal[1]
        + ray->dir[2] * tri->normal[2];
    if (dot > -EPSILON && dot < EPSILON)
        return 1;

    float dot2 = ray->orig[0] * tri->normal[0] + ray->orig[1]
        * tri->normal[1] + ray->orig[2] * tri->normal[2];
    *t = -(tri->d + dot2) / dot;

    float pointa = ray->orig[tri->i1] + ray->dir[tri->i1] * *t;
    float pointb = ray->orig[tri->i2] + ray->dir[tri->i2] * *t;

    float uu0 = pointa - tri->v0[tri->i1];
    float uu1 = tri->v1[tri->i1] - tri->v0[tri->i1];
    float uu2 = tri->v2[tri->i1] - tri->v0[tri->i1];
    float vv0 = pointb - tri->v0[tri->i2];
    float vv1 = tri->v1[tri->i2] - tri->v0[tri->i2];
    float vv2 = tri->v2[tri->i2] - tri->v0[tri->i2];

    float alpha, beta;
    if (uu1 == 0.0) {
        beta = uu0 / uu2;
        if (beta < 0.0 || beta > 1.0)
            return 2;
        alpha = (vv0 - beta * vv2) / vv1;
```

```
    } else {
        beta = (vv0 * uu1 - uu0 * vv1) / (vv2 * uu1 - uu2 * vv1);
        if (beta < 0.0 || beta > 1.0)
            return 3;
        alpha = (uu0 - beta * uu2) / uu1;
    }

    if (alpha < 0 || (alpha + beta) > 1.0)
        return 4;

    return 0;
}
```

Četnost použitých operací

Tabulka uvádí maximální počet kolikrát může být daná operace volána.

+	-	*	/	Podmínka
8	11	13	3	4

Tabulka 4.6: Četnost operací metody Badouel

Z těchto operací jsou celkem 2 skalární součiny vektorů.

Dosažení různých návratových hodnot

Opět pro zajímavost uvedu, jak často se funkce vrací z jednotlivých možností. Pro tento test bylo vygenerováno 1 000 000 trojúhelníků a byl proveden na pc AMD Athlon 64 3000+. Tabulka ukazuje procentuální zastoupení dané návratové hodnoty (viz. kód na začátku sekce 4.5.2) a také čas potřebný k dosažení těchto hodnot (toto měření počítá 500 000 000 výpočtů pro skupinu trojúhelníků, které s příslušnými paprsky vrací danou hodnotu).

4.5.3 Chirkov

Algoritmus

```
int chirkov (TRIANGLE *tri, RAY *ray, float *t) {
    float signSrc = tri->normal[0] * ray->orig[0] + tri->normal[1]
        * ray->orig[1] + tri->normal[2] * ray->orig[2] + tri->d;
```

Návratová hodnota	Četnost / procent	Doba pro výpočet (sec.)
0	121 034 / 12,10 %	50,203
1	1 / 0,00 %	5,516
2	18 / 0,01 %	33,640
3	637 180 / 63,72 %	37,000
4	241 767 / 24,17 %	49,125

Tabulka 4.7: Návratové hodnoty metody Badouel

```
    float signDst = tri->normal[0] * ray->end[0] + tri->normal[1]
        * ray->end[1] + tri->normal[2] * ray->end[2] + tri->d;

    float dd = signSrc - signDst;

    float ay = tri->v1[tri->i1] - tri->v0[tri->i1];
    float az = tri->v1[tri->i2] - tri->v0[tri->i2];
    float by = tri->v2[tri->i1] - tri->v0[tri->i1];
    float bz = tri->v2[tri->i2] - tri->v0[tri->i2];

    float dely = ray->end[tri->i1] - ray->orig[tri->i1];
    float delz = ray->end[tri->i2] - ray->orig[tri->i2];

    float basey = ray->orig[tri->i1] - tri->v0[tri->i1];
    float basez = ray->orig[tri->i2] - tri->v0[tri->i2];

    float adelxbase = signSrc * (ay * delz - az * dely)
        + dd * (ay * basez - az * basey);

    if (adelxbase * (signSrc * (dely * bz - delz * by)
            + dd * (basey * bz - basez * by)) >= 0.0) {
        float cy = tri->v2[tri->i1] - tri->v1[tri->i1];
        float cz = tri->v2[tri->i2] - tri->v1[tri->i2];
        basey = ray->orig[tri->i1] - tri->v1[tri->i1];
        basez = ray->orig[tri->i2] - tri->v1[tri->i2];
        if (adelxbase * (signSrc * (dely * cz - delz * cy)
                + dd * (basey * cz - basez * cy)) < 0.0) {
            *t = - signSrc / (ray->dir[0] * tri->normal[0] + ray->dir[1]
                * tri->normal[1] + ray->dir[2] * tri->normal[2]);
            return 0;
        }
    }

    return 1;
}
```

Četnost použitých operací

Tabulka uvádí maximální počet kolikrát může být daná operace provedena.

+	-	*	/	Podmínka
11	20	29	1	2

Tabulka 4.8: Četnost operací metody Chirkov

Z těchto operací jsou celkem 3 skalární součiny vektorů.

Dosažení různých návratových hodnot

Pro tento algoritmus je zbytečné tyto údaje uvádět, protože algoritmus má pouze dvě možnosti návratu. Doba jejich dosažení se rovná době výpočtu z testu 4.2 s hustotou zasažení 0 nebo 1 (podle toho, který návrat chceme).

4.5.4 Závěr

Pro přehlednost uvedu tabulku obsahující informace z předešlého zkoumání.

Algoritmus	+	-	*	/	Podmínka
Möller	9	16	25	1	4
Badouel	8	11	13	3	4
Chirkov	11	20	29	1	2

Tabulka 4.9: Četnost operací

Z tabulky je vidět, že si celkově metoda Möller vystačí s méně operacemi, než metoda Chirkov. Potřebuje však více podmínek. Podmínky se ukázaly jako nejnáročnější část celého výpočtu. Procesory Intel je zvládají lépe než AMD, a proto je na nich metoda Möller rychlejší než Chirkov. Metoda Badouel má problém, že obsahuje také 4 podmínky, ale navíc 3 dělení, což je pro procesor také náročná operace, to metodu posouvá až na poslední místo. Pokud bychom dostali procesor, který bude velmi rychle dělit, získala by tato metoda mnohem lepší výsledky. Celkového zrychlení všech metod by se dalo dosáhnout zefektivněním rozhodování podmínek, čímž by ještě více získala metoda Möller.

4.6 Počítání barycentrických souřadnic průsečíku

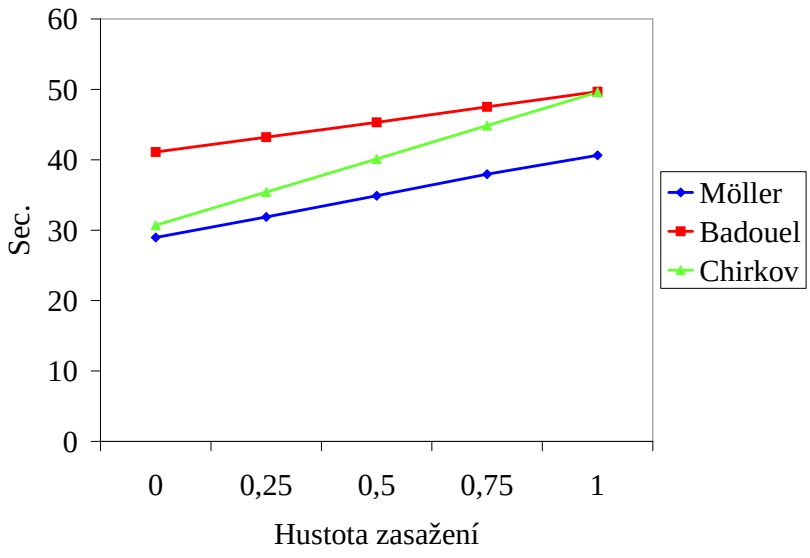
Motivace: Jak jsem na začátku zmiňoval, algoritmus Badouel se mi zdál v nevýhodě, protože jako jediný počítal úplně souřadnice průsečíku. Proto v následujícím testu změřím, jak si počínají i ostatní algoritmy, když musí dopočítat souřadnice úplně.

Použité pc: AMD Athlon 64 3000+ (1,8 GHz, Winchester, L1 (64 KB / 64KB), L2 512 KB), 1 GB RAM (PC3200, 200 MHz).

Měření: 25 000 trojúhelníků při 20 000 opakování. Sloupce tabulky udávají různé hustoty zasažení. Výsledné hodnoty jsou počet sekund potřebných k danému počtu výpočtů.

Algoritmus	0,0	0,25	0,5	0,75	1,0	Efek.
Möller	28,969	31,875	34,891	37,953	40,641	0 %
Badouel	41,109	43,203	45,321	47,516	49,671	30,1 %
Chirkov	30,703	35,406	40,109	44,859	49,594	15,1 %

Tabulka 4.10: Výpočet s vypočtením souřadnic (4.6)



Obrázek 4.4: Graf testu (4.6)

Výsledek

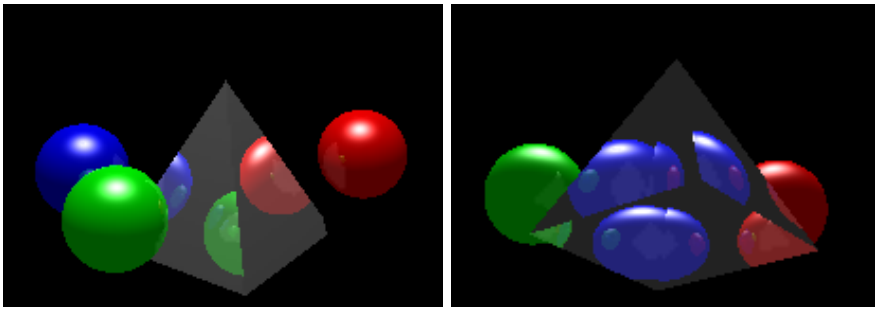
Můžeme vidět, že se nám efektivita algoritmů změnila. Přesto, že byl test prováděn na počítači AMD, nejrychlejším algoritmem se stal Möller. Přestože již není náskok oproti metodě Badouel tak velký jako předtím, pořád je dost znatelný.

Kapitola 5

Implementace ray traceru

Jako další část své práce jsem se rozhodl implementovat tyto algoritmy do existujícího ray traceru. Jako vhodná platforma mi posloužila diplomová práce [4]. Je to funkční ray tracer pracující v reálném čase, který zatím dokáže zobrazovat kouli, válec a rovinu. Využívá urychlující techniku adaptivního podvzorkování. Já k tomuto přidám schopnost zobrazovat trojúhelníky, čímž získáme ray tracer, který je schopný zobrazovat defakto veškerá důležitá primitiva.

Jako první jsem se s ray tracerem seznámil, zjistil jak je navržen a co bude potřeba udělat. Program je dobře navržen a pro moje účely je dobře modifikovatelný. Schopnost zobrazovat nový prvek je dosažena vytvořením nové třídy, která je děděna od třídy objektů. Rozhodl jsem se proto, že do ray traceru implementuji všechny tři algoritmy, které jsem testoval. Vytvořil jsem tedy nové tři druhy objektů, které mohou být zobrazeny (trojúhelník pomocí algoritmu Möllera, pomocí algoritmu Badouela a algoritmu Chirkova). Nejdůležitější u této nové třídy je metoda počítající průsečík paprsku s daným objektem. Algoritmy jsem musel trochu upravit, ale jejich implementace do ray traceru se mi povedla bez větších potíží. Máme tak k dispozici ray tracer, který je schopný zobrazovat trojúhelníky.



Obrázek 5.1: Ukázka scény z ray traceru

V praxi se ukázalo, že mezi jednotlivými metodami není zas tak velký rozdíl a všechny metody jsou vykreslovány jen s malým rozdílem v rychlosti (který se samozřejmě zvětšuje spolu s počtem trojúhelníků ve scéně). Přesto je sledování paprsků velmi náročné. Ray tracer zvládá interaktivně (přes 20 fps) lehčí scény, avšak na použití například v počítačových hrách, kde je scéna složena z mnoha miliónů trojúhelníků, není výpočet dostatečně rychlý.

Kapitola 6

Závěr

Ve své práci jsem zkoumal možnosti zobrazování trojúhelníků pomocí metody sledování paprsků. Zabýval jsem se tedy převážně počítáním průsečíku trojúhelníku s paprskem. Testoval jsem tři různé algoritmy na různých počítačích a v různých situacích. Při těchto testech jsem zjistil několik zajímavých věcí, například že na různých počítačových architekturách je nejrychlejší jiný algoritmus.

Kdybych měl nyní na základě získaných vědomostí implementovat vlastní ray tracer, jednoznačně si k tomu vyberu metodu Tomase Möllera a Bena Trumbore. Jednak je tato metoda nejméně náročná na paměť, jednak se ukázala nejrychlejší na počítačích Intel, které z testovací skupiny představují modernější část. Jako nejrychlejší se ukázala i na počítači AMD, když počítala i barycentrické souřadnice, které bych v ray traceru využil k mapování textur.

Závěrem své práce jsem si vyzkoušel i práci s programem zobrazujícím scény pomocí sledování paprsků a implementoval do něj algoritmy, které jsem zkoušel. Program však nedosahuje příliš dobrých výsledků, co se rychlosti výpočtu scény týče. Aby bylo možné v reálném čase zobrazovat složitější scény, bylo by třeba implementovat nějaké více urychlující metody jako například obalová tělesa, což by se mohlo stát předmětem mojí diplomové práce.

Práce se sledováním paprsků mě velice zaujala a v některých směrech i inspirovala. Začal jsem uvažovat o napsání vlastního ray traceru, který by nepracoval v reálném čase, ale sloužil by jako prostředek pro tvorbu vizuálně velmi zajímavých videosekvencí.

Literatura

[1] Didier Badouel. An efficient ray-polygon intersection. *Graphics gems*, pages 390–393, 1990.

[2] Nick Chirkov. Fast 3d line segment-triangle intersection test. *journal of graphics tools*, 10(3):13–18, 2005.

[3] Jim Hurley. Ray tracing goes mainstream. *Intel Technology Journal*, 9(2), 2005.

[4] Kamil Krupa. Interaktivní sledování paprsku. Master's thesis, FIT VUT v Brně, Brno, 2006.

[5] Marta Löfsted and Tomas Akenine-Möller. An evaluation framework for ray-triangle intersection algorithms. *journal of graphics tools*, 10(2):13–26, 2005.

[6] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics tools*, 2(1):21–28, 1997.

[7] Gilles Tran. Glasses. `http://www.povray.org/`.

Dodatek A

Ovládání programů

A.1 Spuštění ray_test

Program `ray_test.exe` může být spuštěn až se třemi volitelnými parametry. Prvním je hustota zasažení trojúhelníků, druhý počet trojúhelníků a poslední je počet opakování jednoho výpočtu.

`ray_test.exe 0.3 2000 10000`

Spustí program pro zasažení 30 % trojúhelníků z počtu 2 000 a každý výpočet zopakuje 10 000-krát.

Pokud není první parametr zadán, nebo je zadán chybně (i úmyslně), počítá program testy pro hustoty zasažení 0, 0,25, 0,5, 0,75 a 1.

Výchozí nastavení parametrů je 25 000 různých trojúhelníků a 20 000 opakování.

A.2 Ovládání raytracing

Ovládání kamery: přibližování, oddalování kamery: pravé tlačítko myši + pohyb; rotace kamery: prostřední tlačítko myši + pohyb.

Přepínání scén: klávesy 0–9

Inkrementace a dekrementace: klávesy + -, je však nejprve nutné zadat, který parametr se má měnit.

Parametry: mřížka adaptivního podvzorkování: L

mřížka adaptivního podvzorkování v ose x nebo y: X nebo Y

tolerance adaptivního podvzorkování: T

aproximační technika adaptivního podvzorkování: I

čítač objektů některých scén: C

Ostatní: zapnout / vypnout test vrhání paprsků: R

zapnout / vypnout zobrazení: D

zapnout / vypnout adaptivní podvzorkování: S