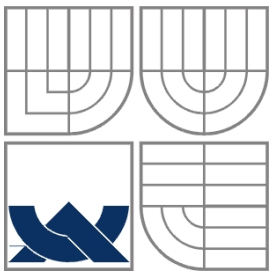
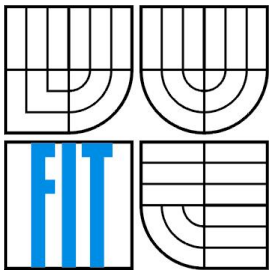


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RYCHLÝ VÝPOČET PRŮSEČÍKU PAPRSKU S TROJÚHELNÍKEM

FAST RAY-TRIANGLE INTERSECTION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Jiří Havel

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Adam Herout, Ph.D.

BRNO 2007

Abstrakt

Trojúhelník je nejpoužívanější primitivum v počítačové grafice. Výpočet jeho průsečíku s paprskem má mnoho využití a často bývá úzkým hrdlem programu. Tato práce se zaměřuje jeho využití a různé způsoby výpočtu. Tyto techniky se snaží kombinovat pro dosažení co nejvyššího výkonu na moderních procesorech.

Klíčová slova

paprsek, trojúhelník, výpočet průsečíku, raytracing, barycentrické koordináty, plückerovy koordináty, determinanty

Abstract

Triangle is the mostly used primitive in computer graphics. Calculation of its intersection with a ray has many applications and is often a bottleneck of a program. This work focuses on its usage and various methods of calculation. It tries to combine these techniques to achieve high performance on modern processors.

Keywords

ray, triangle, intersection calculation, raytracing, barycentric coordinates, plücker coordinates, determinants

Citace

Havel Jiří: Rychlý výpočet průsečíku paprsku s trojúhelníkem. Brno, 2008, diplomová práce, FIT VUT v Brně.

Rychlý výpočet průsečíku paprsku s trojúhelníkem

Prohlášení

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

……
Jiří Havel
19.5.2008

Poděkování

Chtěl bych poděkovat Adamu Heroutovi za vedení při této práci a mnoho užitečných rad.

Obsah

1 Úvod

Výpočet průsečíku paprsku s nejrůznějšími objekty je důležitou operací v počítačové grafice. Za paprsek můžeme považovat přímku, polopřímku, nebo úsečku a to podle toho, co se nejvíce hodí pro danou situaci. Nejčastěji se paprsek používá pro simulaci šíření světla. Je to totiž jeho nejjednodušší popis. Ačkoliv existují i přesnější popisy chování světla, v počítačové grafice se takřka nepoužívají. Jsou totiž mnohem náročnější na výpočet a jejich vlastnosti se dají emulovat jinými způsoby.

Vlnové vlastnosti světla popisují jevy jako interference a difrakce. Tyto jevy se projevují převážně jako vlastnosti materiálů, tudíž se obvykle emulují pomocí textury.

Kvantové vlastnosti světla vysvětlují jeho difůzní odraz. Ten se ovšem podstatně snáze modeluje pomocí Lambertova modelu a radiosity.

Jedním z nejčastěji používaných objektů v počítačové grafice je trojúhelník. Je jednoduchý, uniformní a složitější objekty se dají celkem snadno rozložit na síť trojúhelníků. Díky tomu mnohé programy pracují pouze s trojúhelníkovými sítěmi. Některé z těchto programu více či méně úspěšně předstírají, že zvládají práci i s jinými typy objektů. Ty jsou ovšem vnitřně opět reprezentované trojúhelníkovou sítí. Z tohoto důvodu se operace s trojúhelníky vyplatí silně optimalizovat.

Druhá kapitola popisuje různá využití výpočtu průsečíků. Popisuje také požadavky na vlastnosti používaných metod.

Ve třetí kapitole jsou popsány matematické postupy, které se pro tento výpočet obvykle používají. Dále jsou zde popsány jednotlivé používané metody a to na úrovni vzorců a rovnic.

Ve čtvrté kapitole se zaměřuje na implementaci vzorců z předchozí kapitoly. Jsou zde popsány nejdůležitější vlastnosti architektury x86. Dále se zde nachází jakýsi vzorník různých technik použitelných pro optimalizaci jednotlivých metod. Je zde popsána i implementace jednotlivých metod, ale ve většině případů se jedná o přepis vzorců ze třetí a použití vhodné techniky z této kapitoly.

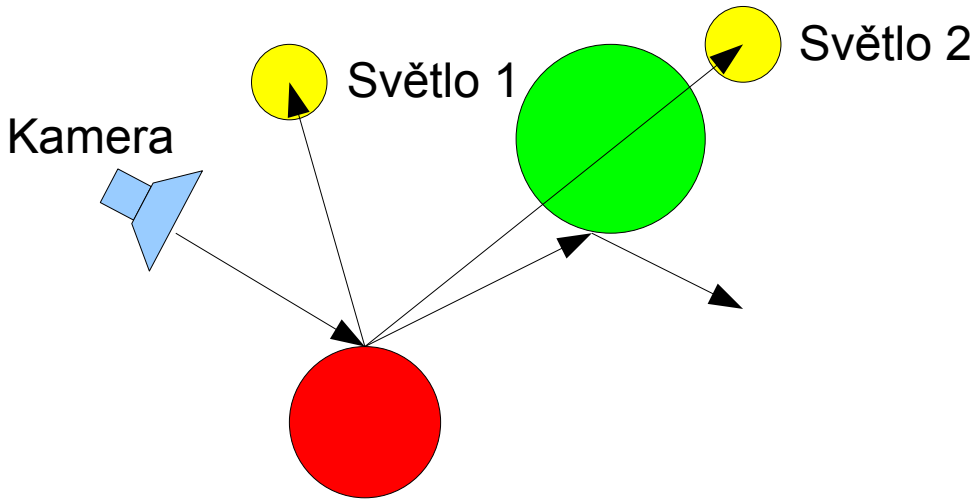
Pátá kapitola se zabývá testováním těchto metod. Popisuje jak způsob testování, tak výsledky. Metody jsou zde porovnávány podle různých vlastností.

2 Využití

2.1 Raytracing

Raytracing, neboli sledování paprsku je základní algoritmus realistické počítačové grafiky. Tato metoda je postavena na sledování světelných paprsků od cíle směrem ke zdroji.

Při vykreslování scény pomocí této metody je každým pixelem výsledného obrazu vržen paprsek směrem do scény. Pokud narazí na těleso ve scéně, je z tohoto bodu vrženo několik dalších paprsků. Ke každému relevantnímu světlu je vržen stínový paprsek. Jeho úkolem je zjistit, jestli se mezi daným bodem a světlem nachází nějaké objekty, které by konkrétnímu světlu stínily. Dále se v závislosti na vlastnostech materiálu vysílá ještě odražený paprsek a paprsek, který prochází tělesem, lomí se a na druhé straně pokračuje ven z tělesa.



Ilustrace 1: Princip raytracingu

Další možností je distribuovaný raytracing, kdy je z každého bodu vedeno více paprsků na rozdíl od jednoho u předchozí metody. Tento způsob poskytuje menší aliasing, lepší vlastnosti difúzních povrchů, měkké stíny, ale má mnohonásobně vyšší výpočetní nároky.

Z předchozích odstavců je zřejmé, že je zde výpočet průsečíku paprsku se scénou úzké místo výpočtu. Vzhledem k tomu, že je do scény vedeno velké množství paprsků, vyplatí se pro zrychlení tohoto výpočtu použít předpočítaná data.

2.2 Raycasting

Raycasting je zjednodušená metoda raytracingu. Na rozdíl od raytracingu se nesledují jednotlivé paprsky, ale jejich svazky. Paprsky se sledují pouze po první kolizi, ignoruje se lom, odrazy a nevrhají se ani stínové paprsky. Tato metoda poskytuje málo kvalitní výsledky, ale je extrémně rychlá. Proto byla využita například v prvních 3D enginech (Build Engine a starší), nebo při tvorbě nejstarších filmových efektů (Tron).

V této práci ji zmiňuji částečně z historických důvodů, ale hlavně proto, že její principy mohou být využity v distribuovaném raytracingu a to hlavně u stínových paprsků. Proto bude část práce zaměřena i na optimalizaci sledování svazků paprsků, ať už rovnoběžných, nebo vycházejících z jednoho výchozího bodu.

2.3 Detekce kolizí

Paprsek může být využit pro popis dostatečně malých a rychle se pohybujících objektů. Takové objekty se velmi často vyskytují například v počítačových hrách a vojenských simulacích. Počet vrhaných paprsků je o několik řádů nižší, než u raytracingu. Taktéž se netestují jednotlivé trojúhelníky, ale obalová tělesa. Trojúhelníky se testují až v rámci jednoho objektu pro přesné určení místa kolize, například pro přesné umístění grafického efektu.

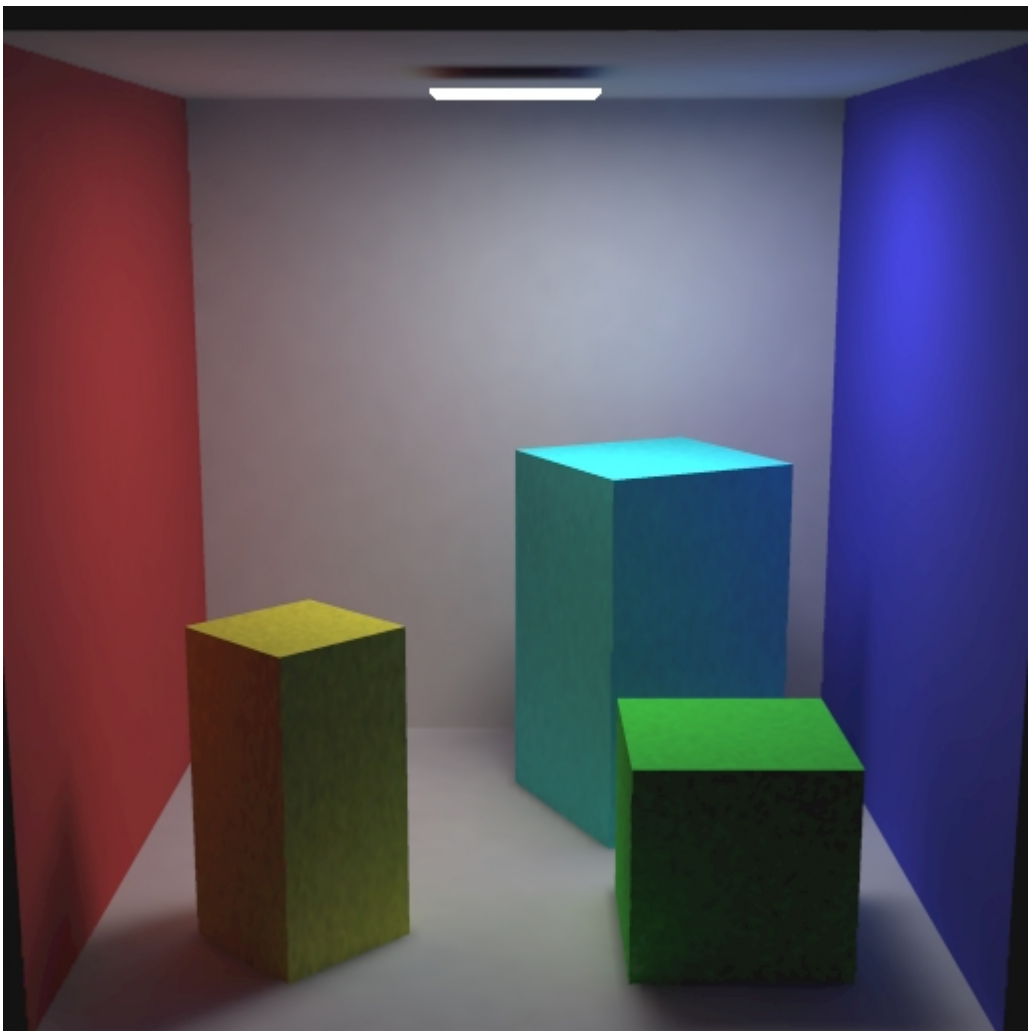
Z tohoto plyne, že tato detekce není úzkým místem programu. Naopak zde hraje roli množství předpočítaných dat. Uplatní se tedy metody, které využívají předpočítaných dat minimum, pokud možno žádná.

2.4 Detekce viditelnosti

Ve hrách a simulacích se paprsky využívají také pro detekci viditelnosti. Pokud paprsek vedený z pozice pozorovatele na cestě k pozorovanému neprotne žádný objekt scény, pak je mezi nimi přímá viditelnost. Stejně jako u detekce kolizí se testují spíše obalová tělesa. Test jednotlivých trojúhelníků pro zpřesnění se používá velmi zřídka. Požadavky na algoritmus jsou tedy obdobné jako u detekce kolizí.

2.5 Radiosita

Radiosita je iterační algoritmus, simulující šíření světla pomocí mnohonásobných odrazů od povrchů. Oproti raytracingu simuluje lépe difúzní povrchy, jelikož počítá s odrazem světla do všech směrů. Světlo odražené z barevných objektů tedy lehce obarví okolní objekty. Na obrázku je vidět vliv radiosity převážně na bílých stěnách.



Ilustrace 2: Cornell box s radiositou. Převzato z artoolkit.org.

Při výpočtu je scéna rozdělena na dostatečně malé plošky, ke kterým jsou spočteny form faktory. Form faktor pro každou dvojici plošek určuje, jak velká část světla, vyzářená jednou ploškou dopadne na tu druhou. Pokud je mezi nimi přímá viditelnost, je tato hodnota závislá na kosinu úhlu, který svírají, útlumu podle vzdálenosti mezi nimi a odrazivosti povrchu. Pokud mezi nimi přímá viditelnost není, je to 0.

Při výpočtu osvětlení scény se nastaví barva těm ploškám, které samy vyzařují světlo, ostatní začínají jako černé. Po každé iteraci je pak barva jednotlivých plošek rovna součtu barev ostatních plošek násobených jejich form faktory s touto ploškou. Po určitém počtu iterací se osvětlení ustálí. Takto osvětlená scéna se poté může vykreslit jednoduchým raytracerem, nebo pomocí rasterizace.

Pokud se nebude brát v potaz dělení prostoru, pak při počtu plošek N je počet form faktorů N^2, což je náročné jak na výpočet, tak na paměť. Pokud jsou plošky dostatečně malé, aby se dalo zanedbat, že můžou být zastíněné pouze částečně, pak se pro detekci viditelnosti při jejich výpočtu může použít právě sledování paprsku. Pokud bude sledování paprsku použito i pro vykreslování, může být s výhodné využít stejný algoritmus jak pro výpočet form faktorů, tak pro následné vykreslení. Zde je využití předpočítaných dat obzvlášť vhodné.

3 Algoritmy

3.1 Základní rozdělení

3.1.1 Podle vstupních dat

U vstupních dat rozlišujeme hlavně reprezentaci trojúhelníků. Ty mohou být zadané následovně:

- Trojicí bodů, tedy nijak neupraveným trojúhelníkem. Tyto algoritmy bývají nejpomalejší. Využívají se tam, kde se provádí málo testů.
- Předpočítanými daty, ke kterým již obvykle původní trojúhelníky nejsou potřeba. Tato data jsou optimalizovaná pro výpočet průsečíku a jen těžko se dají použít k něčemu jinému. Použití těchto metod znamená přípravu dat a tedy se vyplatí tam, kde je doba přípravy zanedbatelná proti vlastnímu výpočtu.
- Kombinací předchozích dvou možností, tedy trojúhelníkem s některým dalším parametrem. Ten bývá obvykle použit i k jinému účelu. Běžným příkladem může být trojúhelník spolu s jeho normálovým vektorem.

3.1.2 Podle výstupních dat

Potřebná výstupní data jsou důležitá pro výběr vhodného algoritmu, jelikož někdy mohou být získána v průběhu výpočtu a někdy je nutné je dopočítat zvlášť. Obvykle potřebujeme kombinaci několika z následujících možností.

- Příznak protíná/neprotíná. Je výstupem ve všech případech. Samostatně se užívá u detekcí viditelnosti.
- Parametr přímky/bod průsečíku. Tyto dvě hodnoty jsou mezi sebou snadno převeditelné. U raytracingu se používá pro výpočet odražených a stínových paprsku, někdy i pro texturování. Taktéž je výstupem detekce kolizí.
- Normálový vektor v bodě průsečíku. Používá se pro výpočet odražených paprsků a pro osvětlení. Někdy bývá modifikován vlastnostmi materiálu.
- Texturovací koordináty, nebo nějaké jiné určení pozice bodu v rámci trojúhelníku.

3.1.3 Podle způsobu výpočtu

Pro výpočet průsečíku můžeme použít větší počet matematických postupů. Nejběžnější z nich jsou popsány v další kapitole. Často se ovšem stává, že pomocí dvou i více různých metod získáme na konci stejné nebo velmi podobné vzorce.

Dále je možné výpočet průsečíku na metody, které nejprve počítají průsečík s rovinou a ten dále testují a ty, které počítají průsečík s trojúhelníkem přímo. Ani tady ovšem není ono rozdělení zcela striktní. Například výpočet parametru přímky bývá v obou skupinách takřka identický. Většinou se liší pouze test na příslušnost bodu do trojúhelníku.

3.2 Matematika

V této podkapitole popíšu některé matematické vztahy, na kterých jsou postaveny následující výpočty. Pokud nebude uvedeno jinak, budu využívat následující pojmenování.

- Trojúhelník je daný trojicí bodů A, B, C.
- Přímka je daná bodem O a vektorem D, což odpovídá parametrické rovnici ve tvaru $P = O + \vec{N} \cdot t$
- Rovina trojúhelníku je daná normálovým vektorem N a hodnotou d, což odpovídá obecné rovnici ve tvaru $\vec{N} \cdot P + d = 0, \vec{N} = \vec{AB} \times \vec{AC}, d = -(\vec{N} \cdot A)$.

3.2.1 Průsečík paprsku s rovinou

Získá se řešením soustavy rovnic pro přímku a rovinu

$$\begin{aligned} \vec{N} \cdot P + d &= 0 \\ P &= O + \vec{D} \cdot t \end{aligned} \quad .$$

Pro parametr t přímky získáme z této soustavy vztah

$$t = -\frac{\vec{N} \cdot S + d}{\vec{N} \cdot \vec{D}} \quad .$$

Průsečík samotný z tohoto parametru získáme dosazením do rovnice přímky.

Protože na délce normálového vektoru nezáleží, upravuje se někdy tak, aby měla jedna jeho složka délku 1. Označíme li tuto osu jako w a zbylé osy jako u a v, můžeme výpočet parametru t přepsat do tvaru

$$t = -\frac{N_u \cdot O_u + N_v \cdot O_v + O_w + d}{N_u \cdot D_u + N_v \cdot D_v + D_w} \quad .$$

3.2.2 Barycentrické koordináty

Využívají se pro reprezentaci bodů, ležících v trojúhelníku.

Je to trojice hodnot alfa, beta, gamma. Pro každý bod P trojúhelníku ABC platí

$$P = \alpha \cdot A + \beta \cdot B + \gamma \cdot C; \quad \alpha, \beta, \gamma \in \langle 0,1 \rangle; \quad \alpha + \beta + \gamma = 1 \quad .$$

Tento vztah můžeme upravit do častěji používaného tvaru

$$\vec{AP} = \beta \cdot \vec{AB} + \gamma \cdot \vec{AC}; \quad \beta, \gamma \geq 0; \quad \beta + \gamma \leq 1 \quad .$$

3.2.3 Determinanty se znaménkem

Absolutní hodnota determinantu 2x2 udává plochu rovnoběžníku, zadaného dvěma vektory. U determinantu 3x3 je to pak objem, zadaný třemi vektory. Znaménko determinantu udává orientaci těchto vektorů. Je kladné, pokud odpovídá pravotočivé soustavě. Toho je možné využít pro test, zda bod leží uvnitř trojúhelníku.

Výpočet determinantu je možno různými způsoby maskovat. Ve 2D se determinant často zapisuje jako vektorový součin. Také se používá Hillův 'perp dot' product. To je náhrada determinantu skalárním součinem s jedním vektorem kolmým na původní.

$$D = A^T \cdot B = (-A_y, A_x) \cdot (B_x, B_y) = A_x \cdot B_y - A_y \cdot B_x$$

Ve 3D se dá pro výpočet determinantu použít tripple product, což je kombinace skalárního a vektorového součinu.

$$D = \begin{vmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{vmatrix} = A \cdot (B \times C)$$

3.2.4 Plückerovy koordináty

Využívají se pro reprezentaci orientovaných přímek. Běžně se přímky reprezentují pomocí dvou bodů, nebo bodu a směrového vektoru. To znamená, že je možné každou přímku vyjádřit nekonečně mnoha možnostmi. Plückerovy koordináty popisují přímku pomocí normalizovaného směrového vektoru a vektoru od počátku souřadnic k nejbližšímu bodu na přímce.

Místo vektoru od počátku je možno použít vektorového součinu tohoto vektoru se směrovým vektorem přímky. Ten se dá snadno vypočítat z bodu na přímce a jejího směrového vektoru. Přímka procházející bodem O se směrovým vektorem D bude popsána jako dvojice

$$(\vec{D}; O \times \vec{D}) \quad .$$

Alternativně se dají tyto koordináty zapsat jako šestice, která se z bodů A a B vypočte jako

$$(A_x B_y - B_x A_y, A_x B_z - B_x A_z, A_x - B_x, A_y B_z - B_y A_z, A_z - B_z, A_y - B_y) \quad .$$

Důležitým pojmem je side operátor. Pro protínající se přímky má hodnotu 0. Pro přímky a a b se vypočte jako

$$side(a, b) = a_0 b_4 + a_1 b_5 + a_2 b_3 + a_3 b_2 + a_4 b_0 + a_5 b_1 \quad .$$

Pokud vyjádříme paprsek R a strany trojúhelníku AB, BC a CA pomocí plückerových koordinátů, pak pokud mají side(R, AB), side(R, BC), side(R, CA) stejné znaménko, pak paprsek protíná trojúhelník.

3.3 Přímý výpočet

Přímým výpočtem rozumím výpočet průsečíku přímo, bez mezistupně tvořeného výpočtem průsečíku paprsku s rovinou trojúhelníku.

3.3.1 Möller-Trumbore

Principem této metody je transformace paprsku do souřadné soustavy, ve které je jedna souřadná osa dána jeho směrovým vektorem a dvě další stranami trojúhelníku. Počátek této souřadné soustavy leží v jednom vrcholu(v tomto případě bod A) trojúhelníku. Po transformaci do této souřadné soustavy udávají souřadnice počátku paprsku parametr přímky a barycentrické koordináty průsečíku. Po úpravách je tento výpočet možno zapsat jako

$$\vec{P} = \vec{D} \times \vec{AC}$$
$$\vec{Q} = \vec{AO} \times \vec{AB}$$
$$\begin{bmatrix} t \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{\vec{AB} \cdot \vec{P}} \cdot \begin{bmatrix} \vec{AC} \cdot \vec{Q} \\ \vec{AO} \cdot \vec{P} \\ \vec{D} \cdot \vec{Q} \end{bmatrix} .$$

Pokud je jmenovatel prvního zlomku nulový, je paprsek rovnoběžný s rovinou trojúhelníku. Tento jmenovatel je totiž determinant vypočtený ze směrového vektoru paprsku a dvou hran trojúhelníku. Jedná se tedy pouze o jiný zápis skalárního součinu směrového vektoru paprsku a normály trojúhelníku.

3.3.2 Kensler-Shirley

Tento algoritmus sjednocuje výpočet pomocí Plückerových koordinátů a znaménkových determinantů. Algoritmus Möller-Trumbore je taktéž specifickým případem tohoto postupu, ačkoliv jej autoři odvodili ještě dalším možným způsobem. Tento algoritmus poskytuje velké množství způsobů výpočtu, ze kterých autoři vybrali pomocí genetické optimalizace jako nejrychlejší variantu

$$\vec{N} = \vec{AB} \times \vec{CA}$$
$$\vec{I} = \vec{OA} \times \vec{D}$$
$$\begin{bmatrix} t \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{\vec{D} \cdot \vec{N}} \cdot \begin{bmatrix} \vec{N} \cdot \vec{OA} \\ \vec{I} \cdot \vec{CA} \\ \vec{I} \cdot \vec{AB} \end{bmatrix}$$

Test jmenovatele zlomku a barycentrických souřadnic se provádí stejně, jako u algoritmu Möller-Trumbore.

3.3.3 Shevtsov-Soupikov-Kapustin

Tento algoritmus spojuje test pomocí Plückerových koordinátů s projekcí do jedné souřadné roviny. Na rozdíl od předchozích dvou metod již nemá jako vstup původní body trojúhelníku, ale

předpočítaná data tvořená rovino trojúhelníku, jedním bodem a dvěma hranami. Tyto hodnoty jsou dále promítnuté do jedné ze souřadných rovin. Vybrané osy jsou označeny jako u a v, vyřazená osa jako w.

$$d = N_u A_u + N_v A_v + A_w$$
$$E0_u = (-1)^w AB_u / N_w$$
$$E0_v = (-1)^w AB_v / N_w$$
$$E1_u = (-1)^w AC_u / N_w$$
$$E1_v = (-1)^w AC_v / N_w$$

S použitím těchto předpočítaných dat pak výpočet průsečíku vypadá následovně.

$$det = D_u N_u + D_v N_v + D_w$$
$$dett = d - (O_u N_u + O_v N_v + O_w)$$
$$T_u = dett\, D_u - det\,(A_u - O_u)$$
$$T_v = dett\, D_v - det\,(A_v - O_v)$$
$$detb = E1_v T_u - E1_u T_v$$
$$detc = E0_u T_v - E0_v T_u$$
$$\begin{bmatrix} t \\ \beta \\ gamma \end{bmatrix} = \frac{1}{det} \cdot \begin{bmatrix} dett \\ detb \\ detc \end{bmatrix}$$

Ačkoliv je tento algoritmus zařazen mezi přímé metody, tvoří spíše mezistupeň k příští kapitole. Hodnoty Tu a Tv je totiž vektor z bodu A do průsečíku paprsku a roviny trojúhelníku. Pouze je vynásobený hodnotou det a promítnutý do vybrané souřadné roviny.

3.4 Nepřímý výpočet

Nepřímým výpočtem myslím postup, kdy je nejprve vypočten průsečík paprsku s rovinou trojúhelníka a u něj je poté testováno, jestli leží uvnitř trojúhelníku. Protože je vypočtený parametr přímky potřeba pro další výpočet, hodí se tyto metody obzvlášť v případech, kdy je více trojúhelníků odmítnuto na základě testu vzdálenosti, než na základě testů na příslušnost bodu v trojúhelníku.

3.4.1 Badouel

Tento algoritmus je velmi jednoduchou implementací výpočtu barycentrických souřadnic, tedy řešení soustavy rovnic

$$\vec{AP} = \beta \cdot \vec{AB} + \gamma \cdot \vec{AC}\,; \quad \beta, \gamma \geq 0\,; \quad \beta + \gamma \leq 1 \quad .$$

Protože se jedná o soustavu tří rovnic pro dvě neznámé, je z nich nutné vybrat dvě tak, abychom nezatížili výpočet zbytečnou chybou. Promítneme proto trojúhelník do té souřadné roviny, do které má největší průmět. Jednodušší a ve výsledku stejné je ovšem vyřadit osu, do které má největší průmět normálový vektor trojúhelníku. Mohli bychom vyloučit kteroukoliv osu, do níž má

normálový vektor nenulový průmět, ale pokud by byl příliš malý, mělo by to vliv na přesnost výpočtu. Výsledkem je soustava rovnic

$$AP_u = \beta \cdot AB_u + \gamma \cdot AC_u$$
$$AP_v = \beta \cdot AB_v + \gamma \cdot AC_v$$

Ta je v původním algoritmu upravena dosazením do tvaru

$$AC_v \neq 0 \qquad AC_v = 0$$
$$\beta = \frac{AP_u \cdot AC_v - AC_u \cdot AP_v}{AB_u \cdot AC_v - AC_u \cdot AB_v} \qquad \beta = \frac{AP_v}{AB_v}$$
$$\gamma = \frac{AP_v - \beta \cdot AB_v}{AC_v} \qquad \gamma = \frac{AP_u - \beta \cdot AB_u}{AC_u}$$

Z implementačních důvodů je ovšem vhodnější ji pomocí Kramerova pravidla upravit do tvaru

$$\beta = \frac{\begin{vmatrix} AP_u & AC_u \\ AP_v & AC_v \end{vmatrix}}{\begin{vmatrix} AB_u & AC_u \\ AB_v & AC_v \end{vmatrix}} = \frac{AP_u \cdot AC_v - AC_u \cdot AP_v}{AB_u \cdot AC_v - AC_u \cdot AB_v}$$
$$\gamma = \frac{\begin{vmatrix} AB_u & AP_u \\ AB_v & AP_v \end{vmatrix}}{\begin{vmatrix} AB_u & AC_u \\ AB_v & AC_v \end{vmatrix}} = \frac{AB_u \cdot AP_v - AP_u \cdot AB_v}{AB_u \cdot AC_v - AC_u \cdot AB_v}$$

3.4.2 Wald

Tento algoritmus vychází z Badouelova, ale modifikuje jej tak, že část výpočtu přesouvá do přípravy scény. Pokud vezmeme předchozí soustavu rovnic, a vektor AP vyjádříme jako rozdíl bodů, můžeme tyto rovnice upravit do tvaru

$$\beta = P_u \cdot \frac{AC_v}{AB_u \cdot AC_v - AC_u \cdot AB_v} + P_v \cdot \frac{-AC_u}{AB_u \cdot AC_v - AC_u \cdot AB_v} + \frac{A_v \cdot AC_u - A_u \cdot AC_v}{AB_u \cdot AC_v - AC_u \cdot AB_v}$$
$$\gamma = P_u \cdot \frac{-AB_v}{AB_u \cdot AC_v - AC_u \cdot AB_v} + P_v \cdot \frac{AB_u}{AB_u \cdot AC_v - AC_u \cdot AB_v} + \frac{A_u \cdot AB_v - A_v \cdot AB_u}{AB_u \cdot AC_v - AC_u \cdot AB_v}$$

Z tohoto vzorce plyne, že již není potřeba původní trojúhelník, ale stačí nám předpočítaná data. Geometricky těchto šest konstant popisuje dvě přímky v souřadné rovině uv a výpočet barycentrických souřadnic odpovídá výpočtu vzdálenosti bodu od těchto přímek.

3.4.3 Keidy

Tento algoritmus se objevuje ve velkém množství zdrojových kódů. Obvykle je poblíž také komentář s poděkováním Keidymu z fóra Mr Gamemaker. Níže uvedený příklad je převzatý z [10], ale na jiných místech se od tohoto tvaru liší minimálně. Stejně jako jiné hodně optimalizované a málo pochopitelné fragmenty kódu je často přebírán bez znalosti toho, jak funguje.

```
typedef unsigned int uint32;
#define in(a)((uint32&)a)
bool checkPointInTriangle(const VECTOR &point, const VECTOR &pa,
const VECTOR &pb, const VECTOR &pc)
{
      VECTOR e10 = pb - pa;
      VECTOR e20 = pc - pa;
      float a = e10.dot(e10);
      float b = e10.dot(e20);
      float c = e20.dot(e20);
      float ac_bb = (a*c) - (b*b);
      VECTOR vp(point.x - pa.x, point.y - pa.y, point.z - pa.z);
      float d = vp.dot(e10);
      float e = vp.dot(e20);
      float x = (d*c) - (e*b);
      float y = (e*a) - (d*b);
      float z = x + y - ac_bb;
      return ((in(z)&~(in(x)|in(y)))&0x80000000);
}
```

Text 1: Test bodu v trojúhelníku

Vysvětlení a odvození tohoto výpočtu lze nalézt například v [9]. Tento algoritmus je založen na Hillově kolmém skalárním součinu, respektive jeho zobecnění do 3D. Pokud vyjdeme z rovnice

$$\vec{AP} = \beta \cdot \vec{AB} + \gamma \cdot \vec{AC}\,; \quad \beta, \gamma \geq 0; \quad \beta + \gamma \leq 1 \quad ,$$

vynásobíme obě strany NxAB resp. NxAC a upravíme abychom získali hodnotu gamma resp. beta, získáme pro ně vzorce

$$\beta = \frac{\vec{AP} \cdot (\vec{N} \times \vec{AC})}{\vec{AB} \cdot (\vec{N} \times \vec{AC})}$$
$$\gamma = \frac{\vec{AP} \cdot (\vec{N} \times \vec{AB})}{\vec{AC} \cdot (\vec{N} \times \vec{AB})} \quad .$$

Jelikož je Hillův 'perp dot' product jinak zapsaný determinant, což platí i pro jeho zobecnění do 3D, podobnost mezi tímto vztahem a předchozími metodami není překvapivá. Pokud nahradíme normálový vektor N vektorovým součinem AB a AC a výsledek upravíme podle vzorce

$$(\vec{A} \times \vec{B}) \times \vec{C} = \vec{B} \cdot (\vec{A} \cdot \vec{C}) - \vec{A} \cdot (\vec{B} \cdot \vec{C}) \quad ,$$

získáme výsledný vzorec

$$\beta = \frac{(\vec{AB} \cdot \vec{AC})(\vec{AP} \cdot \vec{AC}) - (\vec{AC} \cdot \vec{AC})(\vec{AP} \cdot \vec{AB})}{(\vec{AB} \cdot \vec{AC})^2 - (\vec{AB} \cdot \vec{AB})(\vec{AC} \cdot \vec{AC})}$$
$$\gamma = \frac{(\vec{AB} \cdot \vec{AC})(\vec{AP} \cdot \vec{AB}) - (\vec{AB} \cdot \vec{AB})(\vec{AP} \cdot \vec{AC})}{(\vec{AB} \cdot \vec{AC})^2 - (\vec{AB} \cdot \vec{AB})(\vec{AC} \cdot \vec{AC})} \quad .$$

3.4.4 Ostatní

V této podkapitole uvedu několik dalších metod, na které je možné někdy narazit. Některé jsou čistě ukázkové, některé se i používají. Bez výjimky jsou jednoduché a intuitivní.

3.4.4.1 Součet úhlů

Principem této metody je výpočet úhlů, které svírají vektory z bodu P do každých dvou vrcholů trojúhelníku, jak popisuje vzorec

$$a = \arccos\left(\frac{\vec{PB}\cdot\vec{PC}}{|\vec{PB}||\vec{PC}|}\right)$$
$$b = \arccos\left(\frac{\vec{PC}\cdot\vec{PA}}{|\vec{PC}||\vec{PA}|}\right) \quad .$$
$$c = \arccos\left(\frac{\vec{PA}\cdot\vec{PB}}{|\vec{PA}||\vec{PB}|}\right)$$

Pokud leží bod uvnitř trojúhelníku, bude součet úhlů 360°. V opačném případě bude menší. Metoda vyžaduje výpočet délek vektorů, goniometrické funkce a je náchylná na nepřesnosti při výpočtu.

3.4.4.2 Průnik poloprostorů

Tento test je založen na tom, že každou stranou trojúhelníka prochází rovina, která dělí prostor na dva poloprostory. Pokud se testovaný bod nachází ve stejném poloprostoru jako třetí bod trojúhelníku pro všechny roviny, pak bod leží uvnitř trojúhelníku.

Pro rozdělení prostoru je možné použít kteroukoliv rovinu, krom roviny trojúhelníku. Z tohoto nekonečného počtu jsou ovšem prakticky zajímavé pouze dvě možnosti a to rovina kolmá na rovinu trojúhelníku a rovina procházející počátkem paprsku.

3.4.4.3 Znaménkové plochy

Pokud se bod nachází uvnitř trojúhelníku, pak leží na stejné straně úseček AB BC i CA. Pokud budou mít hodnoty

$$a = \vec{N}\cdot(\vec{AB}\times\vec{AP})$$
$$b = \vec{N}\cdot(\vec{BC}\times\vec{BP})$$
$$c = \vec{N}\cdot(\vec{CA}\times\vec{CP})$$

stejné znaménko, pak bod leží uvnitř trojúhelníku. Za předpokladu, že je normálový vektor N normalizovaný, jsou absolutní hodnoty a, b, c rovny plochám rovnoběžníků, daných dvojicemi vektorů (AB, AP), (BC, BP), (CA, CP).

3.5 Kolmé roviny

Tato metoda spadá mezi nepřímé metody. V samostatné kapitole je z toho důvodu, že byla vytvořena v rámci této práce. Vychází z Badouelovy metody a v principu se podobá Waldově. Na rozdíl od předchozích dvou nevyžaduje projekci trojúhelníku do jedné souřadné roviny. Platí se za to větším počtem matematických operací.

Bez projekce mají vzorce ve Waldově metodě tvar

$$\begin{aligned}\beta &= B_x \cdot P_x + B_y \cdot P_y + B_z \cdot P_z + B_d = \vec{N}_b \cdot P + B_d \\ \gamma &= C_x \cdot P_x + C_y \cdot P_y + C_z \cdot P_z + C_d = \vec{N}_c \cdot P + C_d\end{aligned} .$$

Je to výpočet vzdálenosti bodu od roviny, ovšem bez dělení délkou normálového vektoru. Barycentrická souřadnice bodu je tedy dána jeho vzdáleností od určité roviny. Pro souřadnici beta tato rovina prochází hranou AC a délka jejího normálového vektoru je taková, aby bod B ležel ve vzdálenosti 1. Těchto rovin může být pro každý trojúhelník nekonečné množství. Krom rovin odpovídajícím Waldovu algoritmu jsou prakticky zajímavé také roviny kolmé na rovinu trojúhelníku. Ty získáme podle vzorce

$$\begin{aligned}\vec{N}_b &= \frac{\vec{AC} \times \vec{N}}{|\vec{N}|^2} \\ B_d &= -\vec{N}_b \cdot A \\ \vec{N}_c &= \frac{\vec{N} \times \vec{AB}}{|\vec{N}|^2} \\ C_d &= -\vec{N}_c \cdot A\end{aligned} .$$

Délka normálového vektoru trojúhelníku na druhou ve jmenovatelích upravuje roviny tak, aby byly protilehlé body v požadované vzdálenosti 1. Tento výpočet vznikl úpravou prostého dosazení protějšího bodu.

$$\begin{aligned} D_b &= \vec{N}_b \cdot B - \vec{N}_b \cdot A & D_c &= \vec{N}_c \cdot C - \vec{N}_c \cdot A \\ D_b &= \vec{N}_b \cdot \vec{AB} & D_c &= \vec{N}_c \cdot \vec{AC} \\ D_b &= \vec{AB} \cdot (\vec{AC} \times \vec{N}) & D_c &= \vec{AC} \cdot (\vec{N} \times \vec{AB}) \\ D_b &= D_c = \vec{N} \cdot (\vec{AB} \times \vec{AC}) = \vec{N} \cdot \vec{N} \end{aligned}$$

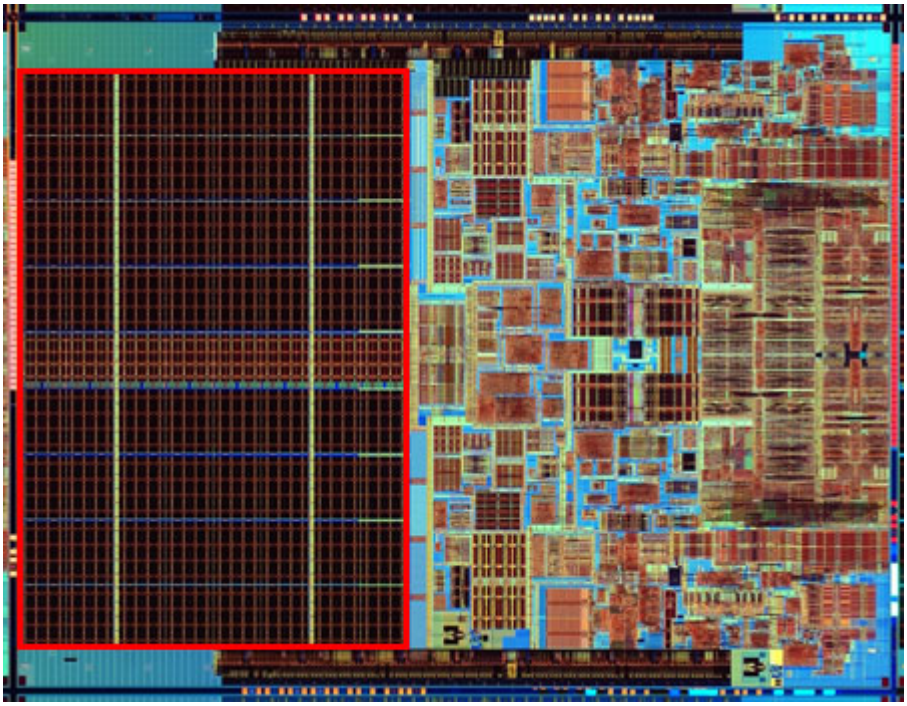
4 Implementace

V této kapitole popíšu implementaci metod, probraných v předchozí kapitole. Jako cílovou jsem zvolil architekturu x86, jelikož je nejpoužívanější u běžných procesorů.

4.1 Vlastnosti architektury x86

Vývoj této architektury je silně ovlivněn zpětnou kompatibilitou mezi jednotlivými generacemi. Ta byla sice klíčová pro komerční úspěch této architektury, ovšem na druhé straně znamenala zakonzervování některých vlastností, pro moderní procesory nepříliš vhodných.

Ačkoliv byla tato architektura původně CISC, současné procesory jsou hybridy, které si berou to lepší z obou skupin. Jako CISC mají komplexnější instrukční sadu a kompaktnější kód, který zabírá méně místa. To je výhodné, jelikož kapacita CACHE procesoru je stále silně omezená. I přesto ovšem tato paměť zabírá významnou část plochy procesoru, jak je vidět například na následujícím snímku procesoru Core 2, kde zarámovaná CACHE zabírá okolo 40% jeho plochy.



Ilustrace 3: Procesor Core 2 Duo

Moderní procesory kód spíše interpretují, než přímo vykonávají. Od procesoru Pentium mají tyto procesory jádro RISC, na kterém provádějí přeložený kód. Tato jádra mají mnoho pomocných registrů, několik paralelních jednotek a dlouhou pipeline. V současné době zabírají tyto jednotky menší plochu čipu než logika, která pro ně překládá původní kód.

4.1.1 Spekulativní vyhodnocení instrukcí

Současné procesory dokáží vykonávat instrukce v jiném pořadí, než jsou zapsány v programu. Pokud některá instrukce ještě nemá připravená data, tak se procesor pokusí najít další instrukci, kterou bude moci provést, zatímco první bude čekat na data. Je zřejmé, že tyto instrukce na sobě nesmí být závislé. Zajímavé je, že tyto instrukce nemusí využívat různé registry. Procesory mají velké množství pomocných registrů, které můžou instrukcím přiřazovat podle aktuální potřeby.

Pro co nejlepší využití procesoru je tedy vhodné navrhnout výpočty tak, aby na sobě co nejméně závisely. Někdy je vhodnější použít dva složitější výpočty, které na sobě nezávisí, než jednodušší, které jsou na sobě závislé.

4.1.2 Predikce větvení

Dlouhá pipeline procesorů způsobuje problémy při větvení, protože je nutné ji vyprázdnit a počkat na její naplnění instrukcemi z cíle skoku. To znamená velmi výrazné zpomalení. U nepodmíněných skoků tento problém nenastává, jelikož je možné včas určit cíl skoku a začít načítat instrukce ze správného místa. Jiná situace je u podmíněných skoků, jelikož to, jestli se skok provede, nebo není v době načítání instrukce známé.

Předvídání podmíněných skoků funguje obvykle tak, že se procesor snaží odhadnout na základě předchozích vyhodnocení, jestli se skok provede nebo ne. Pokud se chování některého skoku opakuje v čase s jednoduchým vzorem, pak funguje predikce takřka spolehlivě. Taktéž kratší smyčky s konstantním počtem opakování dokáží procesory odhalit a správně zpracovat. Pokud se nejedná o takovou situaci, je vhodnější se přílišnému větvení v časově kritických úsecích kódu raději vyhnout.

4.1.3 Vektorové instrukce

Vektorové, nebo také SIMD instrukce jsou instrukce, které dokáží zpracovat jednou instrukcí více dat najednou. Odtud také plyne zkratka SIMD, neboli Single Instruction, Multiple Data. Na architektuře x86 se vyskytují tři skupiny těchto instrukcí. Jsou to MMX, 3DNow! a SSE.

MMX je z těchto tří nejstarší. Využívá registry pro výpočty s pohyblivou řádovou čákou ke zpracování celočíselných hodnot. Z původních 80 bitů jich využívá 64, takže dokáže najednou provést 2, 4, nebo 8 operací, podle šířky operandů. Přepínání mezi MMX a původním FPU je ovšem pomalé. Jakožto celočíselné se tyto instrukce pro zpracování geometrických dat příliš nehodí.

3DNow! je rozšíření MMX pro zpracování čísel s pohyblivou řádovou čárkou. Najednou může zpracovávat dvě čísla s jednoduchou přesností. Bohužel je tato sada specifická pro procesory AMD.

SSE bylo vyvinuto Intelem v reakci na 3DNow! a chyby MMX. Na rozdíl od 3DNow! je podporují oba výrobci. SSE existuje v několika verzích. Původní SSE umožňovalo pracovat pouze se čtyřmi čísly s jednoduchou přesností. SSE2 přidává rozšířenou přesnost a rozšiřuje instrukce MMX pro práci nad registry SSE. SSE3 přidává hlavně některé horizontální instrukce, které na rozdíl od

3DNow! v SSE chyběly. Nejnovější SSE4 přidává velmi důležitou instrukci pro skalární součin. Kromě vektorových instrukcí obsahuje SSE i instrukce skalární pro zpracování jednotlivých hodnot.

4.1.4 Rozdíly v 64 bitové verzi

64 bitová verze, navržená firmou AMD má oproti 32 bitové více rozdílů, než jen 64 bitové celočíselné registry a větší adresový prostor.

Instrukční sady MMX, 3DNow! i původní instrukce pro zpracování čísel s pohyblivou desetinnou čárkou zůstávají pouze ve 32bitovém režimu z důvodu zpětné kompatibility. Pro zpracování desetinných čísel již slouží pouze instrukce SSE.

Jak celočíselných, tak SSE registrů je v 64bitové verzi dvojnásobný počet tedy 16. Tyto nové registry je možné používat i v 32bitovém režimu. Toto vylepšení je velmi důležité, neboť nedostatek registrů byl jeden z velkých problémů u této architektury. Bohužel ani současných 16 registrů není nijak závratný počet.

Nová verze také odstraňuje nepoužívané vlastnosti jako segmentové adresování. Na druhou stranu umožňuje zakázat provádění částí paměti, což zlepšuje zabezpečení. Také zjednodušuje tvorbu dynamických knihoven.

4.2 Společné techniky

Protože se různé způsoby výpočtu často podobají, je možné stejné techniky použít pro optimalizaci různých metod. Tyto techniky tedy proberu samostatně, pouze se zmíním, pro které metody je možno je použít.

4.2.1 Využití SIMD instrukcí

Ačkoliv se některé překladače snaží vektorové instrukce využívat automaticky, nejsou při tom vždy úspěšné. Vektorové instrukce je totiž třeba zohlednit již při návrhu programu a datových struktur. Pro spolehlivé využití těchto instrukcí je nutno použít buď vkládaný assembler, nebo vestavěné(intrinsic) funkce překladače. Ačkoliv je možné v assembleru napsat celou aplikaci, je to silně nevhodné.

Vkládaný assembler se liší překladač od překladače. Obzvlášť výrazně se liší překladač GCC. Naproti tomu vkládané funkce se mezi různými překladači takřka neliší. Všichni výrobci vycházejí z vkládaných funkcí překladače firmy Intel.

Tvorba kódu, využívajícího SSE je poměrně přímočará. Jedná se o skládání několika málo obratů, odpovídajícím běžným konstrukcím v programovacích jazycích. Pro ilustraci přidám několik příkladů. Ačkoliv ani zdaleka nepokrývají možnosti SSE, pro implementaci probíraných algoritmů by měly být dostatečné.

4.2.1.1 Skalární součin

Tato funkce vypočítá čtyři skalární součiny tříprvkových vektorů. Krom vkládaných funkcí je zde vidět také uspořádání dat v paměti pro efektivní přístup.

```
void DotProduct(float y[4], const float a[3][4], const float b[3][4])
{
      _mm_store_ps(y, _mm_add_ps(_mm_add_ps(
            _mm_mul_ps(_mm_load_ps(a[0]), _mm_load_ps(b[0])),
            _mm_mul_ps(_mm_load_ps(a[1]), _mm_load_ps(b[1]))),
            _mm_mul_ps(_mm_load_ps(a[3]), _mm_load_ps(b[3]))));
}
```

Text 2: SSE skalární součin

4.2.1.2 Podmíněné přiřazení

Větvení při výpočtu pomocí SSE je obtížnější, než při běžném výpočtu. Problém nastává v případě, že pro některé hodnoty v registru vychází test jinak, než pro jiné. V této situaci je nutné spočítat pro všechny hodnoty obě varianty a pomocí podmíněného přiřazení z nich vybrat ty správné. Následující kus kódu demonstruje jednoduchou podmínku if(a>0) b += a; Pro zjednodušení jsou již hodnoty a a b v proměnných typu __m128.

```
__m128 mask = _mm_cmpgt_ps(a, _mm_setzero_ps());
if(_mm_movemask_ps(mask))
{
      b = _mm_or_ps(
            _mm_and_ps(mask, _mm_add_ps(a, b)),
            _mm_andnot_ps(mask, b));
}
```

Text 3: SSE podmíněné přiřazení

Porovnávací funkce v SSE nenastavují jeden bit, ale celou šířku operandu. Díky tomu funkce and nechá v registru hodnoty pouze tam, kde byla podmínka splněna a andnot tam, kde nebyla. Jejich sečtením získáme na všech místech správné hodnoty. Funkce movemask vybere z registru čtyři znaménkové bity. Přiřazení se v tomto případě neprovede jen v případě, že porovnání nebude platit ve všech čtyřech případech. Z tohoto důvodu je vhodné, aby byla data co nejpodobnější. V opačném případě je mnoho operací provedeno naprosto zbytečně.

4.2.1.3 Horizontální součet

Až do verze 3 nemělo SSE pro horizontální operace takřka žádnou podporu. Proto bylo nutné horizontální operace implementovat pomocí přesunů prvků.

```
const __m128 temp=_mm_add_ps(x,_mm_movehl_ps(x,x));
y = _mm_add_ss(temp,_mm_shuffle_ps(temp,temp,1));
```

Text 4: SSE horizontální součet

Je zřejmé, že horizontálním operacím je vhodné se vyhýbat. Naštěstí nejsou potřeba pro paralelní výpočet několika hodnot, ale spíše pro operace s jedním vektorem.

4.2.1.4 Inverzní hodnota

Ačkoliv je v SSE operace dělení, obrácená hodnota se obvykle počítá jinak.

```
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 5: SSE obrácená hodnota

Jedná se o odhad pomocí instrukce rcp, který je dále zpřesněn jedním krokem Newtonovy metody, tedy podle vzorce

$$y_{i+1} = 2 \cdot y_i - x \cdot y_i^2 \quad .$$

Tento postup na rozdíl od operace dělení nepotřebuje načítat jedničku z paměti. Také bývá rychlejší, protože je možné jej rozložit mezi okolní instrukce.

4.2.2 Seskupování primitiv

Tato optimalizace úzce souvisí s využitím vektorových instrukcí, ačkoliv některé techniky je možné využít i bez nich. Vektorové instrukce je sice možné využít i u testů s jednotlivými primitivy pro práci s vektory, ale častěji se používají pro skupiny primitiv.

4.2.2.1 Skupiny trojúhelníků

Skupiny trojúhelníků se nepoužívají příliš často. Při jejich použití je nutné buď dělit scénu dostatečně hrubě, nejlépe na čtveřice trojúhelníků, nebo tyto skupiny za běhu skládat. Ačkoliv je možné využít souvislosti mezi trojúhelníky jako je společná rovina, u paprsků jsou společné vlastnosti častější. Výpočet průsečíku se skupinou trojúhelníků také vyžaduje horizontální porovnávání.

4.2.2.2 Svazky paprsků

Svazky paprsků se často využívají u raytracingu, kdy je do scény vrháno velké množství paprsků s podobnými vlastnostmi. Tyto svazky mívají často společný počátek a svírají mezi sebou jen velmi malý úhel. To je výhodné jak pro průchod scénou, tak pro větvení výpočtu.

I bez použití SSE je možné použít některé optimalizace. U metody Kensler-Shirley je možné spočítat normálový vektor trojúhelníku pouze jednou pro celý svazek. Společný počátek paprsků umožňuje spočítat polovinu průsečíku s rovinou jednou pro celý svazek. Svazek rovnoběžných paprsků by byl ještě výhodnější, jelikož ve většině metod tvoří jmenovatele skalární součin směrového vektoru paprsku a normálového vektoru trojúhelníku. Naneštěstí jsou rovnoběžné paprsky podstatně méně časté než paprsky se společným počátkem, nebo obecné svazky.

Využití SSE pro svazky paprsků je poměrně jednoduché. Krom kombinace technik popsaných v minulé kapitole je nutné ošetřit rozpad svazků na jednotlivé paprsky.

4.2.3 Organizace paměti

Vhodné uspořádání struktur v paměti má vliv na využití cache. Tím má na rychlost zpracování někdy větší vliv, než samotný výpočet. Data je možno uložit do paměti jako pole struktur, nebo strukturu polí.

```
struct S1 {
      float p1;
      float p2;
      //...
} pole_struktur[pocet];

struct S2 {
      float p1[pocet];
      float p2[pocet];
      //...
} struktura_poli;
```

Text 6: Pole struktur a struktura polí

Pole struktur je pro využití CACHE výhodnější, než struktura polí. CACHE funguje dobře, pokud se data čtou z míst blízko sobě. Struktura polí způsobuje, že se v jednom kroku čtou data z různých míst v paměti. Tento způsob je také poměrně nepřehledný a hůře spravovatelný. Obzvlášť neintuitivní je pro programátory zvyklé na objektový přístup. Struktura polí je ovšem potřena pro efektivní využití SSE. Tento přístup je využívají Kensler a Shirley v [3]. Tyto dva přístupy je možné zkombinovat. Ačkoliv se stále jedná o strukturu polí, jsou související data umístěná na jednom místě v paměti. Také odpadají problémy se správou velkého množství polí. Bohužel je to výměnou za nepraktické adresování. Výsledek je také ještě méně přehledný a intuitivní než předtím.

```
struct S {
      float p1[4];
      float p2[4];
      //...
} kombinace[pocet%4 ? pocet/4 + 1 : pocet/4];
```

Text 7: Kombinace obou přístupů

Pro rychlý přístup do paměti je nutné její správné zarovnání. Procesory architektury x86 sice nezarovnaný přístup do paměti zvládají, ale má dopad na rychlost. Obzvlášť patrné je to u instrukcí SSE, které vyžadují zarovnání na 16 bytů. Příkazy pro zarovnání struktur se bohužel liší překladač od překladače. C++ používá __declspec(align(16)), GCC __attribute__((aligned(16)). Dále je možné použít #pragma direktivu, nebo zadat zarovnání struktur jako parametr překladače.

4.2.4 Přesun dělení

Dělení je velmi náročná instrukce a proto je snaha dělení úplně odstranit, nebo omezit jeho použití. Pro kompletní výpočet průsečíku je bohužel alespoň jedno dělení potřeba. Pro minimalizaci jeho vlivu na výkon je možno použít dvou naprosto rozdílných přístupů.

První možnost je začít dělit co možná nejdříve. V případě výpočtu zlomku tedy nejprve vypočítat jmenovatel, ten invertovat a poté pokračovat výpočtem čitatele. Výpočet inverze by měl proběhnout alespoň částečně paralelně s výpočtem čitatele, tedy s menším dopadem na výkon. Tento postup využil Wald ve své implementaci. Vhodný je v situacích, kdy se dá dělení přeskočit jen velmi zřídka, například při použití jemného dělení prostoru, kdy je počet odmítnutých testů velmi malý.

Druhou možností je odložit dělení na co nejpozději. Výhodné je to v případě, že se odloží za testy vypočtených hodnot, díky čemuž se často přeskočí. V případě, že je znaménko jmenovatele dopředu známé je takovýto přesun obvykle velmi jednoduchý. Pokud ovšem znaménko dopředu neznáme, situace se komplikuje. U výpočtu průsečíků se dá vyhnout komplikovaným testům pomocí testů znamének.

Tuto optimalizaci znázorním na pseudokódu popisujícím například algoritmus Kensler-Shirley.

```
float jmenovatel = VypoctiJmenovatel();
float inv_jm = 1.0f/jmenovatel;

float t = VypoctiT()*inv_jm;
if((t > 0) & (t < maxt))
{
      b = VypoctiB()*inv_jm;
      c = VypoctiC()*inv_jm;
      if((b >= 0) & (c >= 0) & (b+c <= 1))
      {
            UlozVysledek(t, b, c);
      }
}
```

Text 8: Pseudokód metody Kensler-Shirley

V tomto kódu je možné přesunout inverzi jmenovatele až před volání UlozVysledek. Vzhledem k reprezentaci čísel s pohyblivou řádovou čárkou je možné zjednodušit testy pomocí bitových operací. Výsledný kód vypadá následovně.

```
float jm = VypoctiJmenovatel();
float t = VypoctiT();
if(Sign(Int(t) ^ Int(jm*maxt - t)) == 0)
{
      float b = VypoctiB();
      float c = VypoctiC();
      if(Sign((Int(b) ^ Int(c)) | (Int(b) ^ Int(jm-b-c))) == 0)
      {
            float inv_jm = 1.0f/jm;
            UlozVysledek(t*inv_jm, b*inv_jm, c*inv_jm);
      }
}
```

Text 9: Pseudokód po úpravě

Funkce Int převede proměnnou typu float na celočíselný typ, funkce Sign vymaskuje znaménkový bit. Operace xor je zde použita pro test shodnosti znaménkových bitů. Určitým nedostatkem je, že se tyto dva kódy nechovají úplně stejně. Problém vzniká naštěstí pouze na hraničních hodnotách, které se vyskytují velmi zřídka.

Nevýhodou tohoto postupu je, že míchá dohromady celočíselné typy a typy s pohyblivou desetinnou čárkou. To může znamenat potřebu přenášet tyto hodnoty mezi různými registry a někdy i přes pomocnou paměť. U SSE by tento problém neměl nastat, protože pro oba typy operací používají stejnou sadu registrů. Bohužel firma AMD toto nedoporučuje, protože jejich procesory používají skryté pomocné hodnoty, které je potřeba přepočítávat, což výpočet brzdí.

Tento postup použil původně Keidy a Shevtson. Dá se ovšem použít a také jsem jej použil u všech algoritmů.

4.2.5 Projekce do souřadné roviny

Výpočet v jedné ze souřadných rovin využívá Badouel, Wald a Shevtson. Tato projekce bývá řešena pomocí výběru souřadnic. Pro tento výběr je potřeba předávat indexy vybraných os. To je možné dvěma způsoby.

Pokud vybereme osy ve správném pořadí, stačí předávat index pouze jedné z nich, například osy w. Další dvě získáme například pomocí vzorce

$$u = (w+1) \bmod 3$$
$$v = (w+2) \bmod 3$$

Operaci modulo je nejlepší implementovat tabulkou.

Druhý způsob je předat všechny tři indexy ve struktuře trojúhelníku. Tato možnost je vhodná v případě, kdy je ve struktuře trojúhelníku volné místo kvůli zarovnání. Jako příklad může sloužit třeba struktura, použitá u Waldova algoritmu.

```
struct Triangle
{
      float nu, nv, nd;
      unsigned w;
      float bu, bv, bd;
      unsigned unused1;
      float cu, cv, cd;
      unsigned unused2;
};
```

Rovinu pro promítání je také možno vybrat při implementaci a tedy odpadá potřeba ji přidávat k trojúhelníku. Pokud ovšem není možné zaručit, že ve vstupních datech nebudou trojúhelníky kolmé k této rovině, jedná se o riskantní optimalizaci s nejistým výsledkem.

4.3 Implementace jednotlivých metod

V této podkapitole proberu techniky použité pro jednotlivé algoritmy, ať už to budou optimalizace specifické pro jednotlivé algoritmy, nebo konkrétní využití technik probraných v předchozí kapitole.

4.3.1 Kensler-Shirley

Tento algoritmus má oproti Möller-Trumbore výhodu, že využívá normálový vektor trojúhelníku místo jiného vektorového součinu. Tento vektor je totiž možné spočítat jednou pro celý svazek paprsků, čehož je využito i v původních testech. Mimo to bývá normálový vektor jeden z častých dodatečných parametrů trojúhelníku.

Tento algoritmus je možné jednoduše upravit tak, aby používal předpočítaná data. Vhodná jsou například kombinace normálového vektoru, bodu A a hran AB a AC. Další vhodnou úpravou oproti původní implementaci je přesun dělení za všechny testy. Ten je navíc oproti jiným algoritmům velmi jednoduchý.

4.3.2 Shevtsov-Soupikov-Kapustin

Tento algoritmus jsem implementoval podle původní práce v takřka nezměněné verzi. Kombinuje všechny techniky zmíněné v předchozí kapitole. Jediný rozdíl je možné udělat v předávání vybraných os, kdy je se dá využít prostor pro zarovnání.

4.3.3 Badouel

V teoretické části jsem se zmínil o tom, že vzorce získané pomocí Kramerova pravidla jsou pro implementaci vhodnější, než předchozí. Důvodů je několik.

- Větvení – To je nutné pro zamezení dělení nulou. To sice nastává velmi zřídka a predikce tedy bude fungovat dobře, ale zjednodušení výpočtu není tak velké aby se to vyplatilo.
- Závislosti – To, že výpočet druhé barycentrické souřadnice závisí na první sice zjednodušuje výpočet, ale zároveň brání procesoru jej provést paralelně.
- Různé jmenovatele – Dva různé jmenovatele znamenají dvě operace dělení namísto jedné a dvou násobení. Protože dělení může být i desetkrát náročnější operace, jedná se o velmi nepříjemné zpomalení.

I při použití vzorců, získaných pomocí Kramerova pravidla má tento algoritmus nedostatky. Stále vyžaduje dvě dělení. Jedno pro výpočet průsečíku s rovinou a druhé pro výpočet barycentrických koordinátů. Taktéž vyžaduje projekci do souřadné roviny, k čemuž jsou potřeba předpočítaná data.

4.3.4 Wald

Tento algoritmus je velmi dobře implementován již v původní práci [5]. Pro porovnání jsem tento algoritmus implementoval s dělením odloženým na co nejpozději. V tomto případě byla tato úprava poměrně obtížná a měla za následek znatelný nárůst počtu násobení. Jmenovatelem je totiž potřeba vynásobit výchozí bod paprsku u výpočtu testovaného bodu a dále se jím musí vynásobit hodnoty bd a cd u výpočtu barycentrických souřadnic.

4.3.5 Keidy

Implementace tohoto algoritmu je prostá kombinace technik, popsaných v předchozí kapitole. Tento algoritmus trpí stejným nedostatkem, jako Badouelův a to potřebou dvou dělení. Ačkoliv na první pohled nevyužívá normálový vektor trojúhelníku, je jeho výpočet uvnitř skryt. Pro srovnání s ostatními algoritmy jsem jej doplnil o zbytek výpočtu barycentrických souřadnic.

4.3.6 Kolmé roviny

Tento algoritmus sice využívá více předpočítaných dat než Waldův, ale jeho nároky na paměť jsou stejné. Je to způsobeno zarovnáním na 16 bytů. U Waldova algoritmu zůstává 8 bytů nevyužito a 4B se používají pro informaci o vybrané souřadné rovině. Rozdíl mezi velikostí předpočítaných dat u Waldova a tohoto algoritmu je právě 12 bytů.

5 Testy

Pro testování jsem využil náhodně vygenerovaných paprsků a trojúhelníků. Inspiroval jsem se postupem, použitým v [3]. Pro implementaci jsem použil jazyk C++. Jako překladače jsem použil GCC ve verzích 3.4.5 a 4.3.0 a Visual C++ 9.0, které je součástí Visual Studio 2008. U všech překladačů byla nastavena maximální optimalizace pro rychlost. Obě verze GCC vytvářely 32bitový kód, MSVC vytvářelo kód 32 i 64 bitový.

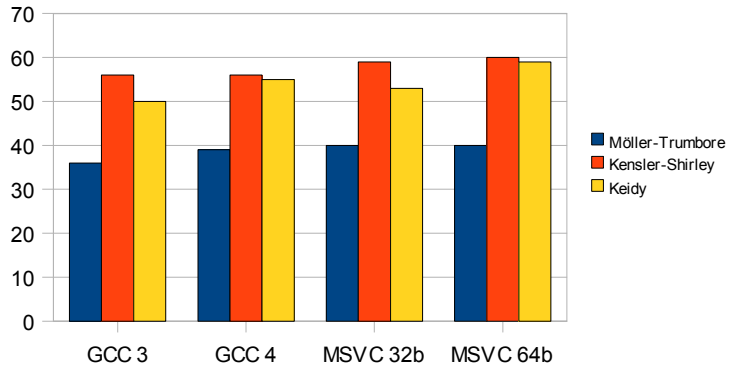
Testovací data v tomto tvaru odpovídají modelům bez hierarchického dělení. Výpočet je velmi často ukončen předčasně. Pro otestování nejhoršího možného případu, tedy že je výpočet vždy proveden celý jsem nevytvářel jiná testovací data, ale zablokoval předčasné ukončení výpočtu. Hodnoty jsou v milionech testů za sekundu a získaná data mají chybu okolo 3%.

Protože je díky kombinaci různých překladačů, algoritmů a jejich různých implementací naměřených hodnot velké množství, budou v následujících srovnáních vybrány z důvodu přehlednosti jen vhodné podmnožiny údajů.

5.1 Rychlosti metod podle typů

Zde se metody nedělí do skupin podle způsobu výpočtu jako v teoretické části, ale podle typu vstupních dat. Ten má totiž podstatnější vliv na rychlost. Podle rychlosti jsou metody rozděleny z důvodu přehlednosti. Mezi nejrychlejšími a nejpomalejšími je rozdíl skoro jednoho řádu.

5.1.1 Bez předpočítaných dat

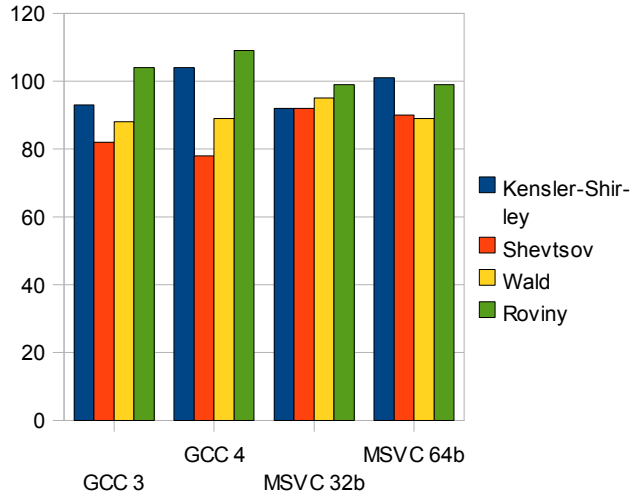


Ilustrace 4: Rychlost v milionech testů za sekundu - bez předpočítaných dat

Obě rychlejší metody více vyhovují testovacím datům, jelikož nejprve testují vzdálenost průsečíku. To je také jediný důvod pro tak velký rozdíl mezi Möller-Trumbore a Kensler-Shirley. Složitost jejich výpočtu je stejná a srovnání u nejhoršího případu to ukáže celkem přesně.

5.1.2 S předpočítanými daty

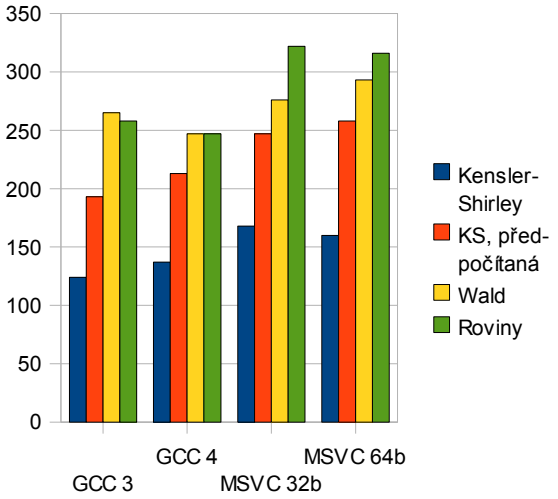
Z tohoto grafu je vidět, že metody s méně operacemi nemusí být ve výsledku rychlejší.



Ilustrace 5: Rychlost v milionech testů za sekundu - předpočítaná data

Projekce do roviny je v případě jednotlivých paprsků tak náročná, že ji zjednodušení vzorců nevyváží. V této skupině jsou také vidět výraznější rozdíly mezi překladači. Dá se ale těžko určit, který optimalizuje lépe.

5.1.3 Svazky paprsků

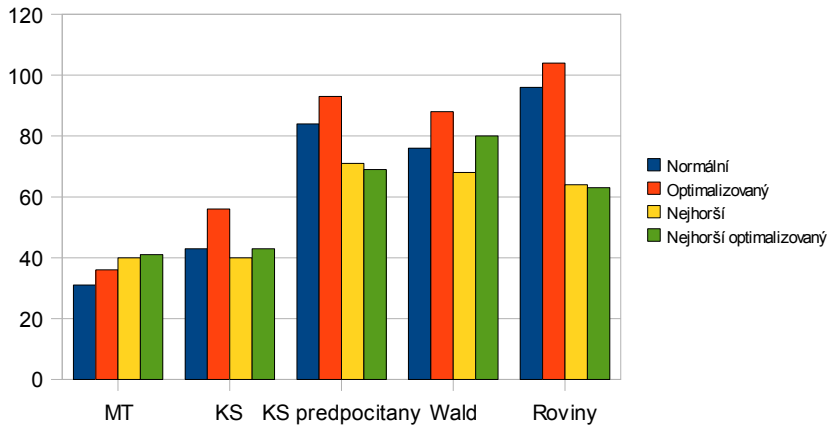


Ilustrace 6: Rychlost v milionech testů za sekundu - svazky paprsků

Pro svazky paprsků již projekce není takový problém, jak je vidět na Waldově algoritmu. To se projevuje hlavně na rozdílu mezi ním a Kensler-Shirley. Oproti Waldově výpočtu a výpočtu s kolmými rovinami je Kensler-Shirley podstatně komplikovanější.

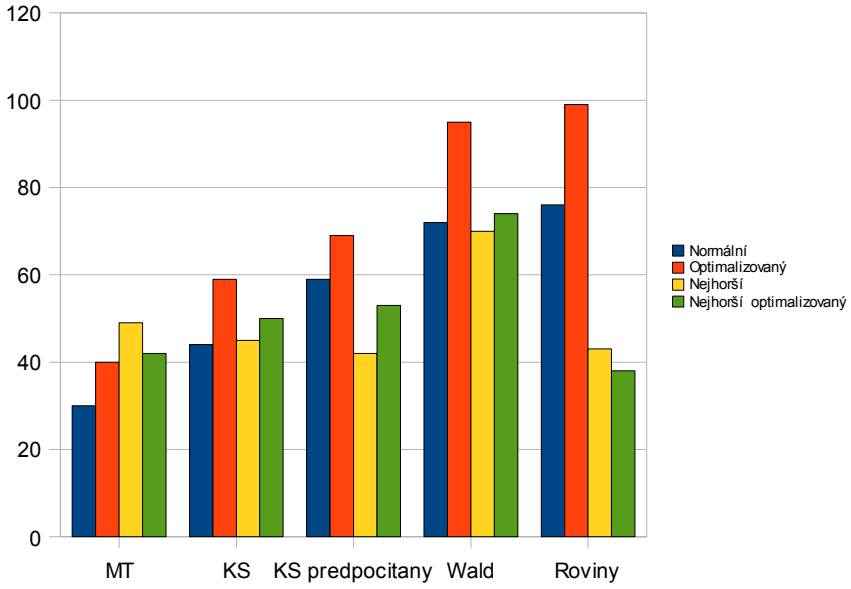
5.2 Vliv přesunu dělení

V předchozích srovnáních měly všechny metody dělení přesunuté až za testy. Vlastní vliv této optimalizace je ale silně závislý na datech, metodě a částečně i překladači. Proto vliv tohoto přesunu porovnávám jak na původních datech, tak na nejhorším případě, kdy nedochází k předčasným ukončením výpočtu.



Ilustrace 7: Výsledky přesunu dělení v milionech testů za sekundu – GCC 3

Na překladači GCC jsou rozdíly méně výrazné, než v případě MSVC.



Ilustrace 8: Výsledky přesunu dělení v milionech testů za sekundu - MSVC 32b

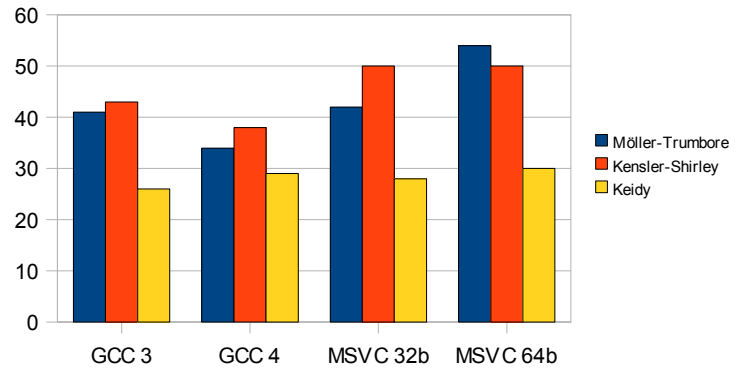
Že má přesun dělení velký vliv na rychlost v případě, že je mnoho testů ukončeno předčasně není příliš překvapivé. Zajímavé je, že ani u nejhoršího případu nemá tento přesun výrazné dopady na

výkon. Největší dopad na výkon by měl být u Waldova výpočtu a kolmých rovin. Tam přesun dělení znamenal přidání pěti, respektive šesti násobení.

V grafu je také jasně vidět, že v nejhorším případě není mezi algoritmy Möller-Trumbore a Kensler-Shirley velký rozdíl. To je způsobeno tím, že ačkoliv jsou z složitostí přibližně stejné, liší se v pořadí, v jakém testují jednotlivé vypočtené hodnoty. Kensler-Shirley nejprve testuje vzdálenost průsečíku, což více vyhovuje původním datům.

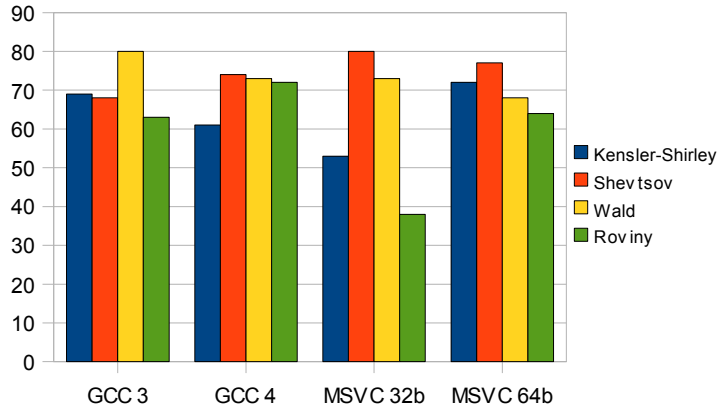
5.3 Nejhorší případy

Pro nejhorší případy má cenu srovnávat i metody mezi sebou, nejen vliv optimalizací. Rychlost pro nejhorší případ totiž odpovídá použití hierarchického dělení prostoru.



Ilustrace 9: Rychlost v milionech testů za sekundu - bez předpočítaných dat

To, že se vyrovnal výkon Möller-Trumbore a Kensler-Shirley odpovídá jejich velmi podobným vzorcům. Keidyho výpočet má proti nim komplikovanější výpočet a také o jedno dělení navíc, což limituje jeho výkon v tomto případě.



Ilustrace 10: Rychlost v milionech testů za sekundu - předpočítaná data

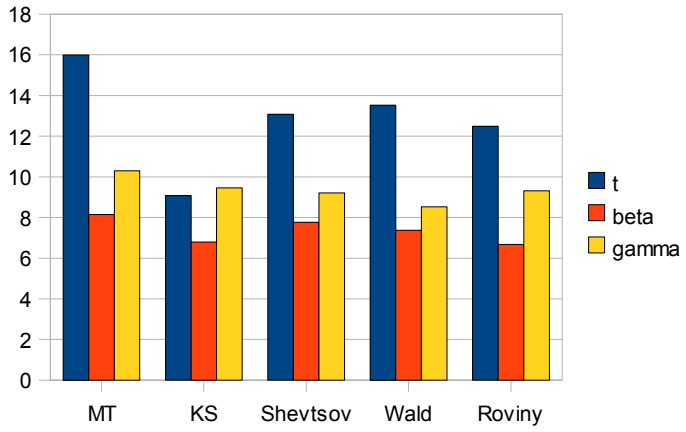
V tomto případě mají metody s jednodušším výpočtem jednoznačně navrch. Překvapivě špatný výsledek kolmých rovin u MSVC není způsoben ani špatným přepisem, ani špatně proběhlým testem.

5.4 Výpočet ve dvojité přesnosti

Původně měl být výpočet ve dvojité přesnosti použit pouze pro srovnání přesnosti jednotlivých metod.

5.5 Přesnost jednotlivých metod

Na testovacích datech nedošlo k situaci, že by dvě metody vybraly pro jeden paprsek dva různé trojúhelníky. Detailnější srovnání chyb je podle průměrné relativní chyby parametru paprsku a barycentrických koordinátů. Jako referenční výpočet byla použita metoda Möller-Trumbore ve dvojité přesnosti. Na chybu neměl takřka žádný vliv ani výběr překladače, ani přesun dělení.



Ilustrace 11: Průměrná relativní chyba v miliontinách

Nejdůležitější informací je, že chyba se pohybuje v řádu miliontin. Ačkoliv se u různých metod liší, nejsou rozdíly tak velké aby je bylo třeba brát v úvahu. Za zmínku stojí, že chyba u Möller-Trumbore je ze všech nejvyšší, i když byla v dvojité přesnosti braná jako reference. Výpočty s dvojitou přesností mají chybu v řádu 10^-14.

6 Závěr

Prostudoval jsem různá využití výpočtu průsečíku paprsku s trojúhelníkem. Algoritmů k jeho výpočtu existuje celá řada. Tyto algoritmy mají různé vlastnosti a hodí se pro různé aplikace. Při detailním studiu ovšem mají mnoho společných prvků. Pro jejich odvození se dá použít mnoho různých matematických postupů, které ale ve výsledku poskytují velmi podobné výsledky. Taktéž při implementaci existuje řada technik použitelných napříč spektrem těchto metod.

Z Waldovy metody jsem odstraněním projekce do roviny získal metodu v některých případech i o 20% rychlejší.

Metody jsem implementoval jak v původním tvaru, tak doplněné o některé optimalizační techniky přejaté z jiných metod. Po jejich otestování můžu pro tyto metody najít vhodné aplikace.

6.1 Vhodné aplikace

6.1.1 Kensler-Shirley

Tato metoda je ze všech testovaných nejuniverzálnější. Dá se velmi jednoduše upravit pro různá vstupní data a také produkuje všechny běžně požadované výstupy. Pro neupravené trojúhelníky na vstupu je nejrychlejší ze všech testovaných. Pro předpočítaná data již není nejrychlejší, ale stále je velmi výkonná.

6.1.2 Keidy

Pro průsečík paprsku s trojúhelníkem nemá tato metoda žádné výhody proti ostatním. Užitečná je v případě kdy se netestuje průsečík paprsku, ale pouze bod na rovině trojúhelníku. Také se hodí v případech kdy nemáme jednoduše k dispozici normálový vektor trojúhelníku.

6.1.3 Wald a Shevtsov-Soupikov-Kapustin

Vlastnosti těchto dvou metod jsou velmi podobné. Obě využívají předpočítaná data a výpočet v jedné souřadné rovině. Obě se hodí pro hierarchicky dělené scény s malým množstvím testů ukončených předčasně. Shevtsov je rychlejší, z Waldova výpočtu se snáze získá bod průsečíku.

6.1.4 Kolmé roviny

Tato metoda se hodí v situacích, kdy se trojúhelníky vyplatí přepočítat na vnitřní reprezentaci, ale už se nevyplatí nad nimi budovat komplikované hierarchické struktury a tím minimalizovat počet testů.

Díky tomu se hodí pro dynamicky se měnící scény. Taktéž je vhodná pro architektury s obtížnou implementaci výběru os.

6.2 Zhodnocení a pokračování

Optimalizace tohoto výpočtu je sice důležitá, sama o sobě ale nestačí. Pro dostatečně rychlé výpočty průsečíku se scénou je nutné využití datových struktur, které změní časovou náročnost tohoto výpočtu z lineární minimálně na logaritmickou. Bez alespoň hrubého omezení testovaných dat je jakákoliv optimalizace samotného průsečíku paprsku s trojúhelníkem nedostatečná. S rostoucím výkonem počítačů bude rozdíl mezi množstvím dat zpracovatelným logaritmickými a lineárními algoritmy stále větší.

Pokračováním této práce bude v prvé řadě doplnění výpočtu průsečíků o nějaký druh dělení scény. Nejlepším způsobem bude zabudování implementovaných metod do některého již existujícího projektu.

Literatura

[1] Badouel D., An Efficient Ray-Polygon Intersection, Graphics Gems, 1990, s. 390-393

[2] Möller T., Trumbore B., Fast, Minimum Storage Ray-Triangle Intersection, Journal On Graphics Tools 1997,

[3] Kensler A., Shirley P., Optimizing Ray-Triangle Intersetion via Automated Search, 2006

[4] Benthin C., Realtime Raytracing on Current CPU Architectures [Ph.D. Thesis], Saarland University, leden 2006

[5] Wald I., Realtime Ray-Tracing and Interactive Global Illumination, IT – Information Technology, 2006

[6] Agner F., Optimizing Software in C++, An optimization guide for Windows, Linux and Mac plattforms, <http://www.agner.org/optimize/>, (únor 2008)

[7] http://www.devmaster.net/wiki/Ray-triangle_intersection

[8] Hill F., The Pleasures of 'Perp Dot' Products, Graphics Gems IV, 1994

[9] Sunday D., Intersection of Rays, Segments, Planes and Triangles in 3D, <http://geometryalgorithms.com/Archive/algorithm_0105/algorithm_0105.htm>, (květen 2008)

[10] Fauerby K., Improved Collision Detection and Response, 2003

[11] Shevtsov M., Soupikov A., Kapustin A., Ray-Triangle Intersection Algorithm for Modern CPU Architectures, GraphiCon 2007

Seznam příloh

Příloha 1. CD se zdrojovými kódy