VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra Informatiky

Raytracing pomocí technologie CUDA

Raytracing with CUDA technology

Rok: 2011

Jan Křístek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Křístek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Raytracing pomocí technologie CUDA
Raytracing with CUDA Technology

Zásady pro vypracování:

Nastudujte základní techniky používané v oblasti raytracingu a seznamte se s technologií CUDA. Realizujte jednoduchý zpětný raytracer, který bude akceptovat trojúhelníkové sítě a základní materiálové vlastnosti. Zaměřte se zejména na takový typ implementace, která bude efektivním způsobem využívat masivního paralelismu GPU. Závěrem by mělo být zhodnocení přínosu použití technologie CUDA.

1. Seznamte se s technologií CUDA.
2. Zvolte vhodné struktury a algoritmy pro paralelní implementaci zpětného raytracingu.
3. Realizujte navržený raytracer pomocí CUDA.
4. Zzhodnoťte přínos technologie CUDA pro zadaný problém.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 20.11.2009
Datum odevzdání: 06.05.2011

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení:

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Poděkování:

Chtěl bych poděkovat Ing. Tomáši Fabiánovi , za jeho cenné rady a připomínky k mé bakalářské práci.

Abstract

Ray tracing is considered to be the future standard for real-time image processing.The aim of this thesis was to prove that at the correct implementation can achieveinteractive ray tracing. This was made accelerating the structure of the division of space Octree.

Abstrakt

Metoda sledování paprsků je považována za budoucí standart, pro real-time zpracovávání obrazu. Cílem bakalářské práce bylo dokázat, že při správné implementaci lze docílit interaktivního trasování paprsků. Bylo toho dosaženo akcelerační strukturou pro dělení prostoru Octree.

Keywords

ray, ray tracing, Octree, Octree traversal, phong shading, CUDA, intersection with the triangle

Klíčová slova

paprsek, trasování paprsku, octree, octree traversace, phong shading, CUDA, průsečík s trojúhelníkem

Seznam použitých symbolů a zkratek

CPU	(Central Processing Unit)- procesor, základní součást počítače
CUDA	(Compute Unified Device Architecture) -technologie paralelního zpracování
GPGPU	(General Purpose Computing on GPU) – matice paralelního zpracování
GPU	(Graphic processing unit) – grafický akcelerátor
API	(Aplication Programing Interface)

Obsah

1 Úvod

S počítačovou grafikou se lidé setkávají už řadu let, skoro v každém profesním odvětví se tato technologie považuje za důležitou pro každodenní práci. V zobrazovací technologii existuje spousta rozdílných renderovacích metod od nerealistických Wire-frame (Drátový model) až po pokročilejší techniky Ray tracing.

1.1 Ray tracing

Cílem počítačové grafiky bylo po mnoho let nalézt způsob, aby výsledný 2D obraz 3D scény vypadal co nejvěrohodněji. Aby dokázal respektovat fyzikální vlastnosti těles v renderovací scéně jako jsou průhlednost materiálu, difusní (matný) nebo spektakulární (lesklý) povrch.

Arthur Appel v roce 1968 vynalezl Ray casting algoritmus. Myšlenka této metody byla v tom, že vrhané paprsky vycházely z 1 oka, paprsek pro každý z pixelů (pixel myšlen jako 1 bod na obrazovce) a najít nejbližší objekt k tomuto paprsku, který mu blokuje cestu. Tento objekt by bylo možné vidět lidským okem i v reálném světě. Po použití parametrů vlastnosti materiálu, vlivu světla ve scéně, dokáže určit zastínění tohoto objektu. Vychází z předpokladu, že objekt není stíněn jiným objektem. Stínění je tedy vypočítáváno jen běžným grafickým stínovým modelem. Ale dokázal ukázat, že když je objekt vyjádřen matematicky, tak jde pomocí vrhání paprsků z oka vypočítat jeho zobrazovací vlastnosti ve scéně.

V roce 1979 přišel *Turner Whited ,* že tento paprsek při dotyku s povrchem tělesa dokáže vytvořit 3 různé paprsky - reflexi, lom světla a stín. Reflexe je paprsek, který pokračuje ve směru odraženého paprsku v zrcadlovém odrazu od lesklých povrchů. Ten se pak protne s jinými objekty a vzniká tzv. odraz. Průhledné materiály fungují tak, že přídává k reflexi to, zda paprsek vstoupil nebo opustil objekt. Ukončení trasování paprsků lze zabránit testem, zda na tento objekt dopadá světlo anebo je stíněn. Tato nová funkcionalita přidala ray tracingu větší realismus.

2 Ray tracing

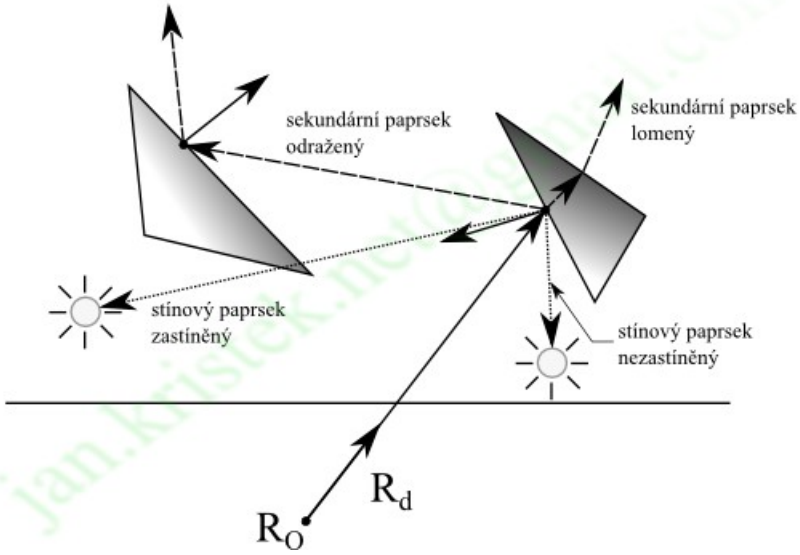
Hlavní myšlenkou ray-tracingu byla už nastíněná dříve v algoritmu ray-casting a to, že vrhané trasované paprsky vycházely z oka pozorovatele. Ray tracing k tomuto algoritmu přidává to, že paprsek při dotyku s tělesem dokáže vygenerovat další 3 odlišné paprsky reflexi, lom světla a stín. Z fyzikální podstaty světla vyplývá, že se paprsky šíří po přímkách různými směry od světelných zdrojů. Když světelný paprsek zasáhne objekt, část světla je odražena, část propuštěna a část pohlcena a s tím ray-tracing počítá a díky tomu dokáže produkovat fotorealistickou grafiku. Některé paprsky opustí prostor scény, jiné zase naopak zasáhnou povrch tělesa. Podle námi definovaných optických vlastností těchto objektů se paprsky odrážejí dál v prostoru scény a také se mohou lomit a ovlivňovat osvětlení v dalších místech scény. Základem těchto metod je trasování paprsku vloženého do scény buď to od světelného zdroje nebo oka pozorovatele, popřípadně obojí. Metody, které sledují paprsek od světelného zdroje, se nazývají shooting methods - metody vystřelující energii. K nim se řadí metody photon tracing a ligth tracing. Metody vycházející od oka pozorovatele se nazývají gathering methods – metody, kdy paprsek na své pomyslné dráze sbírá energii. Mezi tyto metody patří ray-tracing (trasování paprsku) anebo path-tracing (trasování cesty). Metody, které sledují paprsek od světelného zdroje, se však příliš často nepoužívají. Z důvodu počtů paprsků od světelného zdroje, který je velmi vysoký a ne každý z nich se dostane k pozorovateli. K tomu jsou vygenerované obrázky zatížené silnějším šumem. Ale mají i výhodu, když pracují s malými nebo skrytými zdroji světla. Proto v ray-tracingu nesledujeme paprsky vržené světlem, ale paprsek vyšleme od oka pozorovatele přes pixel na obrazovce do prostoru scény. Výhoda této metody spočívá v snadném řízení generování požadovaného počtu paprsku procházejícím pixelem a tudíž výsledný obraz je méně zatížen šumem. Paprsky vržené do scény nám zajišťují to, co je vidět v daném pixelu. Pokud se trasovaný paprsek nedotkne žádného tělesa, poté je pixel obarven barvou pozadí. V opačném případě určíme nejbližší průsečík a zjistíme barvu objektu v námi nalezeném bodě. Z tohoto bodu vyšleme paprsek směrem ke zdroji světla, pro zjištění zda-li je tento bod osvětlen anebo zastíněn jiným objektem a tudíž leží ve stínu. Pokud je materiál tělesa lesklý nebo průhledný, vysílají se další paprsky.

Paprsek je definován:

$$R(t) = R_0 + t * R_d \quad \text{kde} \quad t > 0 \tag{1}$$

Pro jednotlivé typy paprsků ve scéně byly zavedeny tyto názvy:

1) Primární paprsky - Jsou vystřelovány z oka pozorovatele přes pixel obrazovky. Počet primárních paprsků je dán počtem pixelů zobrazovaného okna.
2) Sekundární paprsky - Jsou vytvořené po styku primárního nebo sekundárního paprsku na některý objekt ve scéně. Když se paprsek dotkne objektu, který má nastaven lesklý materiál, na kterém se mohou odrážet další objekty ve scéně nebo po dotyku objektu paprskem, který má nastaven průhledný materiál, vznikne lomený paprsek.
3) Stínové paprsky - Jsou vysílány z místa styku primárního nebo sekundárního paprsku na některý objekt ve scéně ke každému zdroji světla. Jejich cílem je určit, zda je bod styku osvětlen některým z daných zdrojů světla, nebo jestli bod leží ve stínu jiného objektu. U stínových paprsků nepotřebujeme najít nejbližší průsečík, ale nalézt libovolný objekt mezi bodem styku a světelným zdrojem.



Obrázek č. (1) - Cesta trasovaného paprsku (Cuda: 1 kernel)

Důležitou složkou pro renderování výstupního obrazu pomocí trasování paprsku je výpočet osvětlení. Nejvíce používaný je Phongův osvětlovací model rozšířený o složku odraženého a lomeného paprsku.

$$I = I_a + I_d + I_s + I_r + I_t \tag{2}$$

Kde jednotlivé složky jsou:

4) I_a Ambientní složka – vyjadřuje množství okolního světla
5) I_d Difuzní složka – vyjadřuje množství rozptýleného světla na povrchu tělesa. Je založena na Lambertově zákonu difuzního odrazu.
6) I_s Zrcadlová složka – vyjadřuje množství zrcadlově odraženého světla.
7) I_r Odrazová složka – vyjadřuje množství světla ze směru odrazu od jiného tělesa.
8) I_t Lomová složka – vyjadřuje množství světla ze směru lomu. V častých případech je sníženo koeficientem útlumu závislém na vzdálenosti, kterou trasovaný paprsek urazí průchodem tělesa.

Algoritmus.

Když byl nalezen průsečík, pak

$$P = R_0 + R_d * t \tag{3}$$

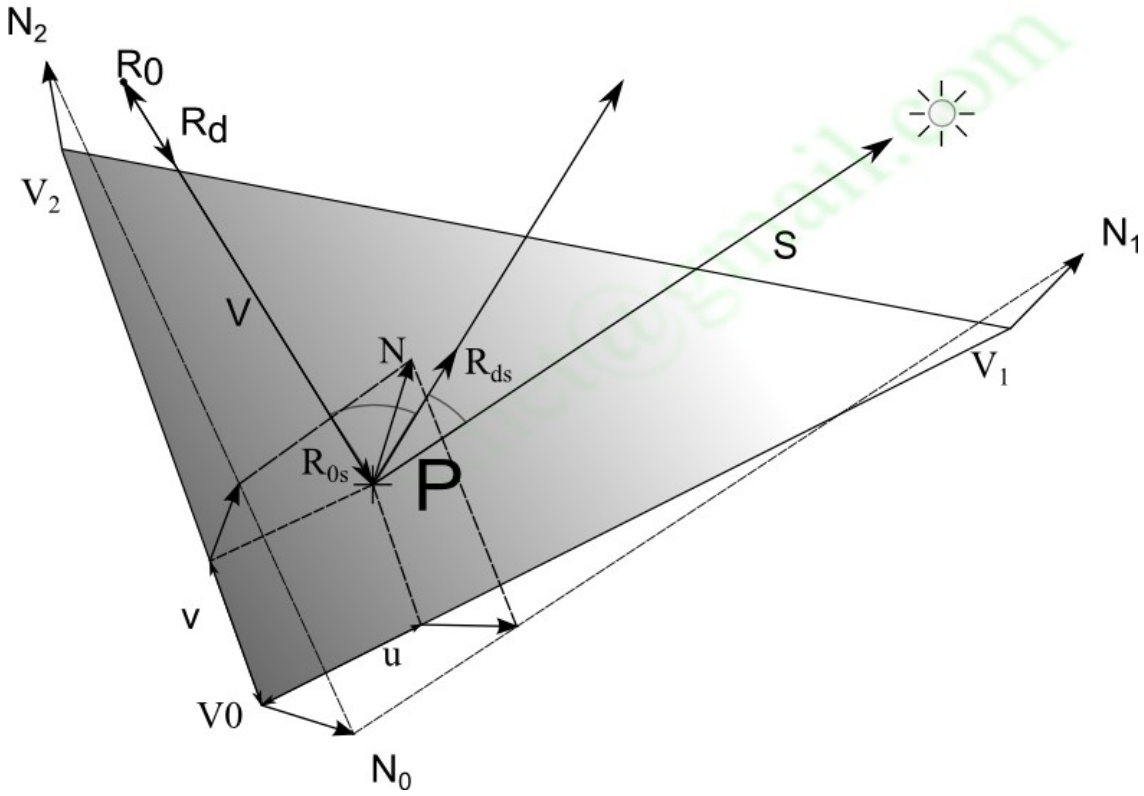
$$L = Ligth_{position} - P \tag{4}$$

$$I_a = konstanta * c_{barvaObjektu} \tag{5}$$

když je průsečík nezastíněn, pak můžeme normálu počítat dvěma způsoby

1) $N_p = (v_1 - v_0) \times (v_2 - v_0)$
2) Nebo můžeme použít počítání normály ze zadaných normál k daným vrcholům

$$N_p = (1 - (u + v))\, A_{normála} + u * B_{normála} + v * C_{normála}$$



Obrázek č. (2) - Počítání normály k danému průsečíku

$$V = R_0 - P \tag{6}$$

$$R_i = normalize(L + V/2)$$

$$I_d = saturatef(N_p \cdot R_i) * c_{barvaObjektu} * síla_{difuzníSložky}$$

$$I_s = saturatef(N_p \cdot R_i)^{(1,50)} * c_{světla} * síla_{zrcadlovéSložky}$$

Základní algoritmus pro rekurzivní trasování paprsků vypadá takto:

ray_tracing(ray R , depth_of_recursion D)

1) Hledání průsečíku P trasovaného paprsku R s nejbližším objektem v prostoru scény.
2) Když nebyl nalezen průsečík P to znamená, že trasovaný paprsek R opustil prostor scény, barva tohoto paprsku je nastavena na barvu pozadí.
3) Když tento průsečík P existuje, vyšli z tohoto bodu stínový paprsek ke všem zdrojům světla. Pokud tento paprsek dorazí ke světelnému zdroji bez toho, aby narazil na jiný objekt, označ tento zdroj světla jako nezakrytý.
4) Vyhodnoť všechny nezakryté světelné zdroje v bodě P .
5) Když hloubka D nedosáhla maximální hloubky trasování paprsků, pak:
   5.1) Pokud je materiál objektu lesklý, pak trasuj nový paprsek (odražený) R_r a volej rekurzivní metodu ray_tracing(R_r , $D+1$)
   5.2) Pokud je materiál objektu průhledný, pak trasuj nový paprsek (lomený) R_t a volej rekurzivní metodu ray_tracing(R_t , $D+1$)
6) Paprsku R přiřaď výslednou barvu jako součet příspěvků osvětlení, barvy odraženého paprsku R_r a barvy lomeného paprsku R_t .

2.1 Průsečík paprsků s trojúhelníkem

Hledání průsečíků s trojúhelníkem je v počítačové grafice jeden ze zásadních problémů. Libovolný objekt můžeme zobrazit ve formě sítě trojúhelníků, tento složený objekt se stane vhodný pro testování dotyčného trojúhelníku s paprskem, který lze efektivně řešit.

Každý algoritmus pro výpočet potřebuje nejméně dva vstupy:

1) Paprsek popsán parametrickou rovnicí
2) 3 vrcholy trojúhelníku($V_{0,}V_{1,}V_2$) s orientací proti směru hodinových ručiček

Metoda počítá průsečík paprsku s trojúhelníkem pomocí barycentrických souřadnic u a v

$$P(u,v)=(1-u-v)*V_1+u*V_1+v*V_2=V_0+u*(V_1-V_0)+v*(V_2-V_O) \tag{7}$$

Platí-li $u+v\leq 1$ a $u+v\geq 0$, pak můžeme říct, že bod P je uvnitř trojúhelníku (speciálním případem je, pokud $u=0$, $v=0$ nebo $u+v=1$, pak průsečík leží na jedné z hran trojúhelníku). Toto je velmi praktické, můžeme použít koeficienty u a v pro výpočet texturového mapovaní, interpolace barvy, interpolace normály v bodě P.

Parametr t je vzdálenost od počátku paprsku po průsečík P trojúhelníku a paprsku, R_0 počátek paprsku a R_d směr paprsku.

Po dosazení do (1) platí:

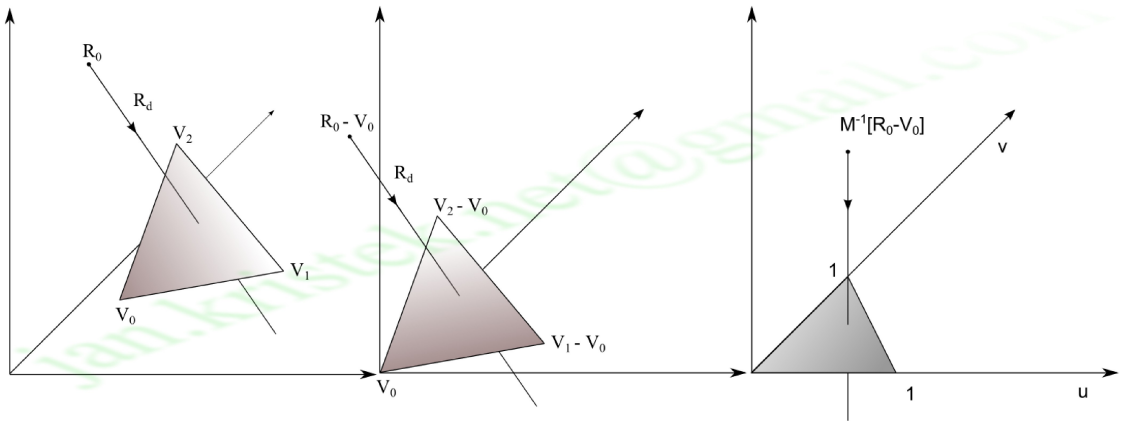
$$R_0+t*R_d=(1-u-v)*V_0+u*V_1+v*V_2 \tag{8}$$

2.1.1 Möller-Thrumborne metoda

Tato metoda byla poprvé popsána Tomasem Möllerem roku 1997 , řadí se k nejvíce používanému algoritmu pro test průsečíku trasovaného paprsku a trojúhelníku. Myšlenka této metody spočívá v nalezení transformací, které nám umožní mapovat vektor z jednoho souřadného systému do druhé báze, která používá 2 hrany trojúhelníku $E_1 = V_1 - V_0$, $E_2 = V_2 - V_0$ a dobře zvolený vektor paprsku $T = R_0 - V_0$. To znamená, když máme n – dimenzionální vektor v n – dimenzionálním vektorovém prostoru, pak lze vyjádřit jako lineární kombinací bázových vektorů prostoru. Existuje vždy nejméně 1 báze pro každý vektorový prostor. Množina n-dimenzionálních vektorů, které jsou lineárně nezávislé a popisují celý n-dimenzionální prostor je báze.Většinou používáme trojrozměrný prostor a tudiž bázové vektory $(1,0,0)(0,1,0)(0,0,1)$. Takže dokážeme popsat libovolný vektor lineární kombinací bázových vektorů $x*(1,0,0)+y*(0,1,0)+z*(0,0,1)$. Můžeme použít u libovolného vektoru jen x, y, z a říct o vektoru v , že $v=(x, y, z)$. Každá matice o rozměrech nxn vyjadřuje transformaci z jednoho libovolného vektorového prostoru do jiného. Musíme najít způsob, jak transformovat vektor $v=(x, y, z)$ na vektor $v=(u, v, t)$ použitím bázového vektoru $V_1 - V_0$, $V_2 - V_0$ a R_d , potom $v = u*(V_1 - V_0) + v*(V_2 - V_O) + \ldots$. Z toho vyplývá , že koeficienty u, v jsou barycentrickými souřadnicemi trojúhelníku.

Algoritmus:

1) Posunutí (R_0, V_0, V_1, V_0) tak, aby V_0 bylo v počátku ($V_0 = (0.0, 0.0, 0.0)$)
2) Transformace trojúhelníku na jednotkový trojúhelník (velikosti hran rovna 1)
3) Směr paprsku je rovnoběžný s osou x



Obrázek č. (3) - Postup algoritmu zobrazený ve 2D prostoru

Po upravení rovnice (1.1.1) dostaneme:

$$[-R_d, V_1 - V_0, V_2 - V_0] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = R_0 - V_0 \tag{9}$$

Po nalezení transformační matice aplikujeme na po R_0 . Z lineární algebry víme

$M * M^I = I$, kde bázovými vektory Matice M jsou $V_1 - V_0$, $V_2 - V_0$ a $-R_d$, Z toho vyplývá, že transformační matice převádí na bázi $(1,0,0)(0,1,0)(0,0,1)$. Po aplikaci inverzní matice M^I na R_0 dostaneme koeficienty u, v, t . Čtvercová matice může být invertovaná, pokud jsou její řádky, v našem případě vektory lineárně nezávislé. (Nastává, pokud směrový vektor R_d leží ve stejné rovině jako samotný trojúhelník, a tudíž neexistuje průsečík)

$$E_1 = V_1 - V_0 \quad , \quad E_2 = V_2 - V_0 \quad \text{a} \quad T = R_0 - V_0$$
$$[-R_d, E_1, E_2] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T \tag{10}$$

Po použití Cramerova pravidla na rovnici (6) dostáváme:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-R_d, E_1, E_2|} \begin{bmatrix} |-T, E_1, E_2| \\ |-R_d, T, E_2| \\ |-R_d, E_1, T| \end{bmatrix} \tag{11}$$

Z lineární algebry víme, že $|A, B, C| = -(A \times C) \cdot B = -(C \times B) \cdot A$

Proto můžeme tuto rovnici (6) rozepsat takto:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(R_d \times E_2) \cdot E_1} \begin{bmatrix} (T \times E_1) \cdot E_2 \\ (R_d \times E_2) \cdot T \\ (T \times E_1) \cdot R_d \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot R_d \end{bmatrix} \tag{12}$$

Kde $R_d \times E_2 = P$ a $T \times E_1 = Q$ přepočítávání pro rychlejší výpočetní výkon

Výhody:

1) Hodně early rejection testů
2) Zabírá minimum paměti (např. nepotřebujeme počítat normálový vektor roviny trojúhelníka)
3) Možná implementace v reálných a celých čísel

Nevýhody:

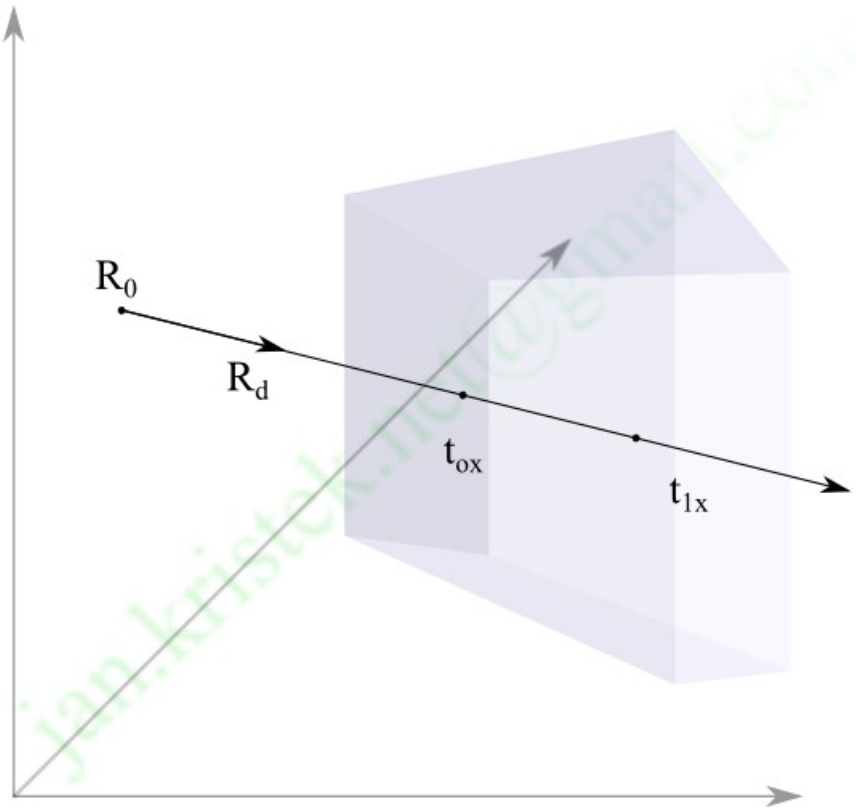
1) Nelze předpočítat moc hodnot

2.2 Průsečík paprsku s kvádrem

Objekt typu kvádr se v ray-tracingu používá velmi často, jako tzv. ohraničující obálka (bounding box) složitějších objektů. Řešení tohoto problémů je založené v nalezení 6 rovin stěn kvádru a zjištěním průsečíku s těmito rovinami s testem průsečíku, zda leží uvnitř obdelniku. Když je paprsek uvnitř, můžeme říct, že paprsek protíná kvádr.

2.2.1 Metoda L.Kay a T.Kajiya

Tato metoda je založena na tzv. Plátech (slabs). Plát je prostor mezi 2 rovnoběžnými rovinami daného kvádru. Metoda počítá s průsečíkem každého z páru plátů trasovaným paprskem blízkého k počátku paprsku (near) a vzdálenějšího (far) průsečíku. Když je nejmenší vzdálenější hodnota menší než největší blízká hodnota , potom můžeme říct, že trasovaný paprsek mine kvádr. V opačném případě ho zasáhne. Pro ohraničující obálky se používají 2 rovnoběžné roviny (plát), kde jejich normálové vektory mají stejný směr jako osy X, Y, Z . Takové uspořádání má výhody, které poté ulehčuje hledání daného průsečíku.



Obrázek č. (4) - Zobrazené průsečíky s kvádrem

Dále musíme definovat kvádr 2 body:

$O_{min} = [X_{min}, Y_{min}, Z_{min}]$

$O_{max} = [X_{max}, Y_{max}, Z_{max}]$

Pokud trasovaný paprsek zasáhne krychli, algoritmus vrátí hodnotu TRUE.

Nastavení hodnot $t_{near} = -\infty$ a $t_{far} = \infty$ na maximum a minimum

Pro každý plát spojený s X, Y, Z platí: jestliže směr trasovaného paprsku se blíží k 0, potom je tento paprsek rovnoběžný s rovinou plátu, takže pokud počátek R_0 není mezí páry rovin plátu $R_0 < O_{min} \vee R_0 > O_{max}$, potom algoritmus vrátí hodnotu FALSE.

1) Když není trasovaný paprsek rovnoběžný s rovinami, potom začne algoritmus vypočítávat vzdálenost průsečíku s rovinami

$$t_{0X} = (O_{minX} - R_{0X}) / R_{dX} \qquad (13)$$
$$t_{1X} = (O_{maxX} - R_{0X}) / R_{dX}$$

2) Když $t_{0X} > t_{1X}$, potom prohoď tyto hodnoty mezi sebou t_{0X} a t_{1X} .
3) Když $t_{0X} > t_{near}$, potom nastav hodnotu t_{near} na $t_{near} = t_{0X}$.
4) Když $t_{1X} < t_{far}$, potom nastav hodnotu t_{far} na $t_{far} = t_{1X}$.
5) Když $t_{near} > t_{far}$, potom vrať hodnotu FALSE.
6) Když $0 > t_{far}$, potom je naše krychle v pozici za paprskem, proto vrať hodnotu FALSE.
7) END.

Pokud projde všemi testy, potom vrať hodnotu TRUE.

Když trasovaný paprsek protne kvádr, pak vzdálenost počátku trasovaného paprsku R_0 od průsečíku je rovna $t = t_{near}$ a vzdálenost R_0 od bodu, z něhož paprsek vychází, rovna t_{far} . Dosazením t do rovnice $R(t) = R_0 + t * R_d$ kde $t > 0$ dostáváme průsečík P .

Z důvodu efektivnosti je vhodné si přepočítat $1/R_{dX}$, $1/R_{dY}$ a $1/R_{dZ}$ a využít toto předpočítávání k výpočtu vzdálenosti paprsku od roviny.

2.2.2 Metoda A.Williams

Amy Williams se svými společníky tuto metodu poupravil pro lepší efektivnost.
1) Na počátku algoritmu se nastaví t_{min} (t_{0X}) na $t_{min} = (O_{minX} - R_{0X}) / R_{dX}$ a t_{max} (t_{1X}) $t_{max} = (O_{maxX} - R_{0X}) / R_{dX}$ (když je směrový vektor R_{dx} záporný, pak ho obrať) a stejným způsobem nastav t_{0Y} a t_{1Y} .
2) Když $t_{min} > t_{1Y} \vee t_{max} < t_{0Y}$, potom náš trasovaný paprsek neprotne kvádr (testuje se průběžně, aby se test mohl zastavit co nejdřív, pokud je jasné, že paprsek neprotne kvádr).
3) Nastav t_{min} na maximální hodnotu z t_{min} t_{0Y} , stejným způsobem nastav t_{max} , ale na minimální hodnotu z t_{max} t_{1Y}
4) Dále se spočítá t_{0Z} a t_{1Z} , znovu se tyto hodnoty porovnají s t_{min} a t_{max} , jestliže $t_{min} > t_{1Z} \vee t_{max} < t_{0Z}$, pak trasovaný paprsek neprotne kvádr.
5) Potom se znovu nastaví t_{min} na maximální hodnotu z t_{min} t_{0Z} a t_{max} , ale na minimální hodnotu z t_{max} t_{1Y}

Souřadnice průsečíku kvádru pak vypočteme dosazením t_{min} do rovnice $R(t_{min}) = R_0 + t_{min} * R_d$ kde $t_{min} > 0$.

2.3 Urychlování trasování paprsku

Urychlování trasování paprsků je velmi podstatné, protože samotný algortimus ray tracingu je velmi časově náročný. Algoritmus spotřebuje 95% strojového času z celkové doby zobrazení, jen při hledání nejbližšího průsečíku. Proto se optimalizační metody zaměřují hlavně na nalezení nejbližšího průsečíku. Pokud použijeme některou z optimalizačních metod, tak může dojít k urychlení výpočtu o 2 až 3 řády. Tyto urychlovací metody se mohou vhodně kombinovat.

Urychlovací metody se dělí na:

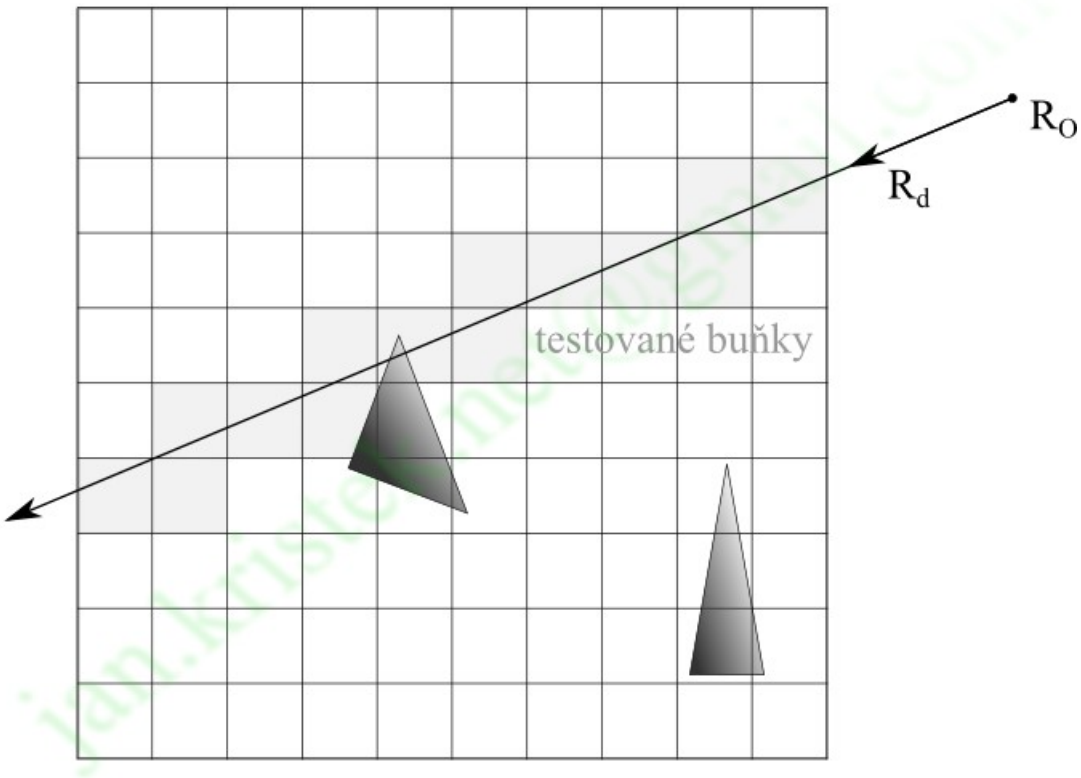
1) Urychlení výpočtů průsečíků
   - 1.1) Rychlejším výpočtem průsečíku
     - Předvypočítáním některých hodnot, lepší analytické metody, použitím jednoduchých obálek (bounding box)
   - 1.2) Zmenšení počtu výpočtů průsečíků
     - Dělení prostoru scény, kooherence paprsků
2) Snížením počtu trasovaných paprsků
   - Adaptivní podvzorkování, nastavení hloubky rekurze trasovaného paprsku
3) Trasováním více paprsků současně
   - Paralelní zpracování, využití GPGPU jednotek (např. Nvidia za pomocí technologie CUDA, které patří část mé bakalářské práce)

Následující část se bude věnovat metodám urychlování trasování paprsku za pomocí dělení prostoru scény, ke které se důkladně věnuje jedna z kapitol mé bakalářské práce Octree. Dělení prostoru scény nám umožňuje vhodné uspořádání informací o scéně tak, aby algoritmus byl schopen určit, která tělesa se nachází v určité prostorové oblasti. A tudíž algoritmus pro určení nejbližšího průsečíku s objektem a zastínění světla tohoto bodu nepracuje se všemi tělesy v prostoru scény, ale pouze s objekty, které zapadají do prostorové oblasti, kterou trasovaný paprsek prochází. Za pomocí dělení prostoru dochází k redukci výpočtů průsečíků a proto nárůstá radikálně i rychlost celého procesu. Dělení prostoru potřebuje pro svou funkcionalitu zavedení pomocných struktur, které se generují při načtení prostorové scény a tyto nové informace se o scéně zaznamenávají.

V počítačové grafice rozlišujeme dva typy dělení prostoru:

1) Uniformní
   - Buňky po dělení jsou stejně velké
2) Neuniformní
   - Buňky se dělí adaptivně, velikosti se liší podle objektů ve scéně

2.3.1 Uniformní dělení prostoru scény



Obrázek č. (5) - Uniformní dělení prostoru scény

Hlavní myšlenka tohoto dělení je v rozdělení prostoru scény do buněk o stejné velikosti, které jsou rovnoběžné se souřadnicovým systémem. Buňky mohou obsahovat více těles nebo těleso může zasahovat do více buněk současně, ale mohou obsahovat i prázdné buňky, protože tato metoda dělení se nepřizpůsobuje prostorové scéně. Tento problém je hlavní nevýhodou této metody na dělení prostoru. Dochází k tomu, že tyto prázdné buňky neefektivně zabírají místo v paměti a má velké paměťové nároky. Výhodou je, že tyto buňky mají pevně definované sousedy a pro trasovaný paprsek je jednoduché a rychlé přes ně procházet.

Pro trasování paprsku v uniformním dělení prostoru 3D-DDA, které vychází z Bresenhamova algoritmu pro vykreslování přímky v 2 rozměrném prostoru. Toto dělení prostoru není vhodné pro řídce obsazený prostor scény tělesy. Dochází k tomu, že mnoho objektů je nahromážděných v malých prostorech a většinu scény zabírají jen prázdné buňky uniformního dělení. Při této kritické situaci docházi k tomu, že trasovaný paprsek zpracovává velké množství prázdných buněk a také při zpracování neprázdné buňky docházi k tomu, že v ní může být mnoho objektů .V nejhorším případě může dojít k situaci, že toto dělení prostoru nám neposkytne žádné časové urychlení. Většinou uniformní dělení prostoru představuje metodu dělení do pravidelné mřížky.

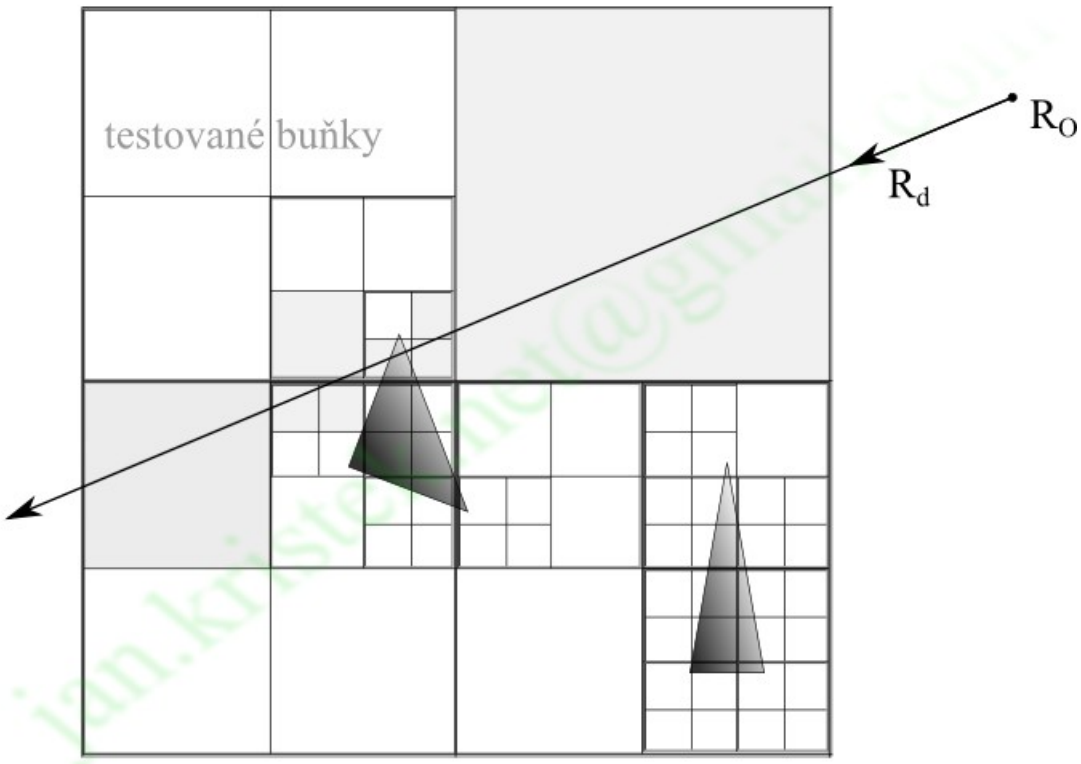
2.3.2 Neuniformní dělení prostoru scény

Hlavní myšlenkou tohoto dělení prostoru je v adaptivním rozdělení velikosti buněk podle rozložení objektů v prostoru scény. Obecně můžeme o tomto rozdělování prostoru říct, že pracuje efektivněji s pamětí a provádí méně početních kroků, než je tomu u uniformního dělení prostoru. Jeho malou nevýhodou je potřeba složitější datové struktury, ve většině případů se jedná o nějaký typ stromu, jehož implementace a efektivní průchod paprsku touto datovou strukturou je poměrně složitá.

Ve většině případů jsou použity struktury Octree, BSP stromy a jejich speciální případ KD strom. V následující kapitole se budu věnovat struktuře Octree.

2.4 Octree

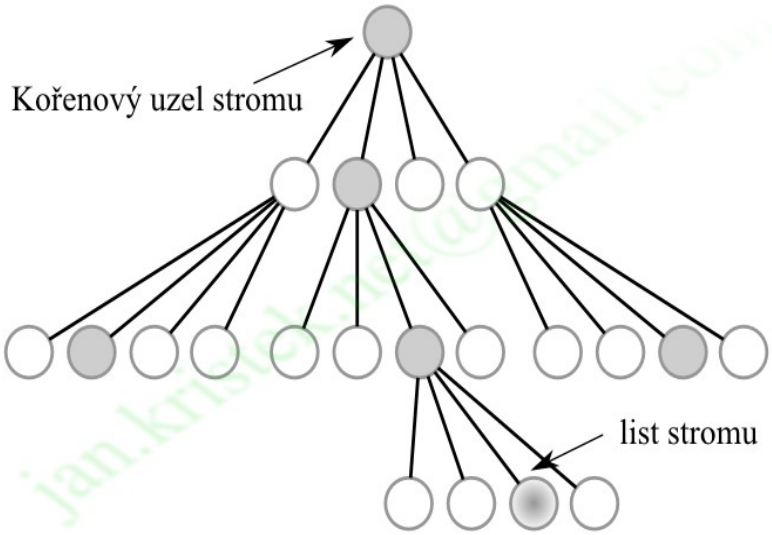
Tato metoda neuniformního dělení prostoru Octree (Octal tree) je struktura uchovávající rozdělení 3D prostoru, která má i svou alternativu v 2D prostoru a známe ji pod názvem Quadtree. Ve většině případů se jedná o stromovou nebo hašovanou strukturu buněk kolmých k souřadnicovému systému. Přitom platí, že každá z buněk se může rozdělit na 8 menších stejně velkých buňek (oktantů), jen pokud tento prostor buňky sdílí svůj prostor s minimálním počtem objektů z prostoru scény.



Obrázek č. (6) - Neuniformní dělení prostoru scény, speciální případ Quadtree

Pří výstavbě Octree struktury v podobě stromu začínáme z kořenového uzlu, který ukazuje na

celý prostor scény. Tento prostor rozdělíme na 8 identických částí – potomků (oktantů), které jsou navázané na kořenový uzel (rodiče). Každého potomka rozdělíme identickým způsobem za pomocí rekurzivního volání.Tato rekurze je zastavena, pokud došlo k dosažení nastavené maximální hloubky stromu nebo minimálnímu nastavení počtu odkazů na objekty v listu.



Obrázek č. (7) - Stromová struktura Quadtree

Průchod paprsku přes Octree strukturu je oproti uniformního rozdělení prostoru mnohem komplikovanější. Pro zjednodušení pochopení tohoto průchodu trasovaného paprsku bude popsáno ve 2D prostoru.

Uzel stromu oktant O je ohraničen v prostoru těmito parametry a $O_{min}=[X_{min}, Y_{min}]$ $O_{max}=[X_{max}, Y_{max}]$. Platí, že aspoň jeden bod P objektu, je ve vztahu s oktantem $X_{min} \leq P_x < X_{max} \wedge Y_{min} \leq P_y < Y_{max}$. Z této rovnice nám vyplývá, že průsečík trasovaného paprsku R a uzlu (oktantu) stromu existuje parametr t , pro který platí:

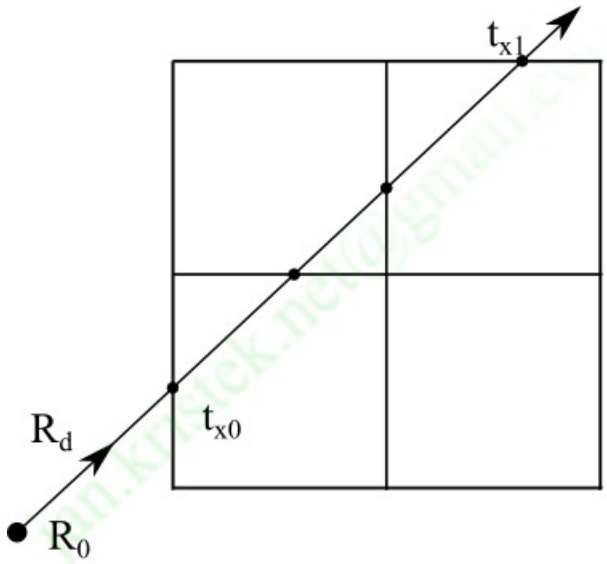
$X_{min} \leq R_x(t) < X_{max} \wedge Y_{min} \leq R_y(t) < Y_{max}$. Tento algoritmus byl nazván parametrický[4], protože pro všechny hodnoty t platí to, že $(R_x(t), R_y(t))$ jsou v prostoru uzlu (oktantu). Pro trasovaný paprsek r , uzel (oktant) O $t_{x0}(O,r), t_{y0}(O,r)$ a $t_{x1}(O,r), t_{y1}(O,r)$ jsou parametry definované pro průsečík trasovaného paprsku s obálkou oktantu. Obecně můžeme říct, že:

$$R_x(t_{xi}(O,r)) = x_i(O) \quad \text{, kde} \quad i \in \{0,1\} \tag{14}$$
$$R_y(t_{yi}(O,r)) = y_i(0) \quad \text{, kde} \quad i \in \{0,1\}$$

Za pomocí inverzní funkce můžeme parametry definovat explicitně:

$$t_{xi}(O,r) = ((x_i(O) - R_{0x}))/R_{dX} \quad \text{Kde} \quad i \in \{0,1\} \tag{15}$$
$$t_{yi}(O,r) = ((y_i(O) - R_{0y}))/R_{dy} \quad \text{Kde} \quad i \in \{0,1\}$$

Tyto 4 hodnoty jsou počítány pro každý z uzlů rodiče, když trasovaný paprsek prochází skrz prostor uzlu. Prvně dochází k výpočtu hodnot kořenového uzlu, když existují, poté se spočítají hodnoty pro každého z potomků kořenového uzlu.



Obrázek č. (8) - Trasovaný paprsek procházející kořenovým oktantem

Pro prostor definovaný mezi těmito body můžeme napsat, že:

$$\Delta t_x(O,r)=t_{x1}(O,r)-t_{x0}(O,r) \tag{16}$$
$$\Delta t_y(O,r)=t_{y1}(O,r)-t_{y0}(O,r)$$

Dosazením rovnice (11) do (12) , dostaneme $\Delta t_x(O,r)=s(O)/R_{dx}$ pro každý z uzlů. Platí, že tento uzel nesmí být list stromu. Pro každé dítě O_i ze stromové struktury O můžeme říct, že pro ně platí rovnice $\Delta t_x(O_i,r)=\Delta t_x(O,r)/2$, můžou být počítaný rozdělením prostoru kořenového uzlu na polovinu. Dokážeme ve správném pořadí rozdělovat další potomky za pomocí rekurzívního volání.

$$x_0(O_i,r)=x_0(O,r)+s(O_i)\Delta x_i \tag{17}$$
$$y_0(O_i,r)=y_0(O,r)+s(O_i)\Delta y_i$$

Po dosazeni do (11) dostáváme pro:

$$t_{x0}(O_i,r)=(x_0(O)+s(O_i)\Delta x_i-R_{0x})/R_{dx} \tag{18}$$
$$t_{x0}(O_i,r)=(x_0(O)-R_{0x})/R_{dx}+(s(O_i)/R_{dx})\Delta x_i$$
$$t_{x0}(O_i,r)=t_{x0}(O,r)+\Delta t_x(O_i,r)\Delta x_i$$

O Δx_i a Δy_i víme, že nabývají hodnot:

$$\Delta x=\{0,0,1,1\}$$
$$\Delta y=\{0,1,0,1\}$$

Stejným způsobem počítáme také pro hodnotu $t_{y0}(O_i,r)$. Ukázali jsme si, jak lze postupně počítat hodnoty děti 0_i nadřazeného uzlu 0 . Počítání pro kořenový uzel je provedeno na základě vzorce (11). Ze znalostí předchozích definic pro uzel a trasovaný paprsek, můžeme obecně říct, že pro tyto dva objekty platí vztah:

$$x_0(O) \leq R_{x0}(t) < x_1(O) \wedge y_0(O) \leq R_{y0}(t) < y_1(O) \tag{19}$$

Založeno na jednoduchém principu:

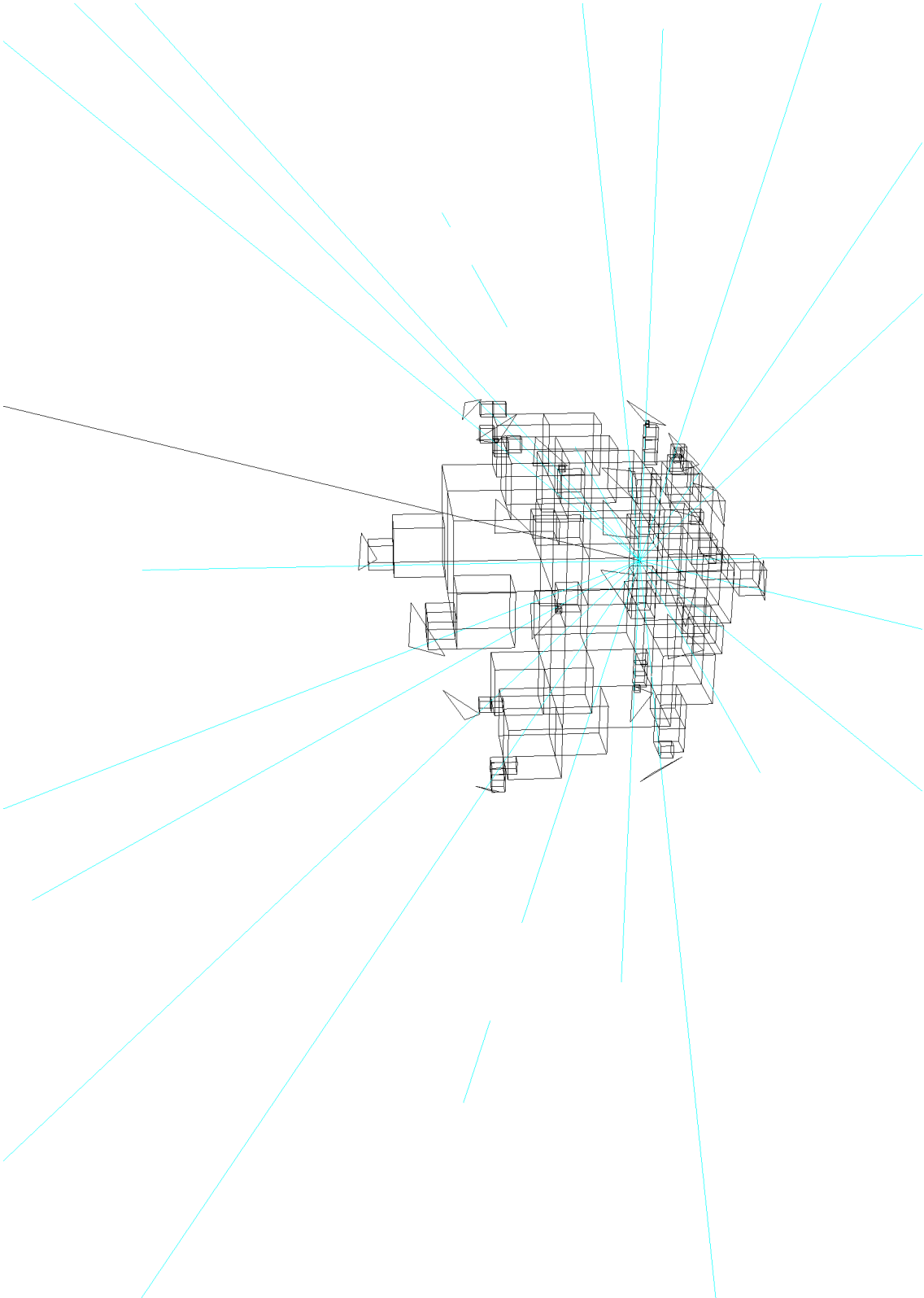
1) Transformace paprsku do požadované roviny, zaznamenání v jakých rovinách se paprsek transformoval do pomocné hodnoty.
2) Zjištění , zda li paprsek protne kořenový uzel oktantu, podle rovnice(15) pokud ne, vrať že nebyl nalezen průsečík, pokud ano, zavolej traversal($t_{0,} t_1$) s parametry kořenového uzlu, které získáme z rovnice (15)
3) traversal($t_{0,} t_1$)
   - 3.1) Vypočítání t_m , ($t_m = t_1$ jednoho z oktantů)
   - 3.2) Zjištění, kterého potomka trasovaný paprsek první protne podle Tabulky č. (1) rekurzivní volání traversal($t_{0,} t_m$) na zjištěného potomka
   - 3.3) Nalezení nového oktantu, který protne paprsek podle Tabulky č.(2), volaní rekurze pro nový oktant
   - 3.4) Když je potomek list stromu porovnaní přiřazených trojúhelníků testem na průsečík. Není li nalezen průsečík, vrať se zpět a hledej dál. Pokud paprsek neopustí kořenový uzel. Je-li nalezen, vrať příslušné parametry a ukonči algoritmus.

Vstupní rovina	Podmínky pr	Bit afekt
XY	$t_{xm}(O) < t_{z0}(O)$ $t_{ym}(O) < t_{z0}(O)$	0 1
XZ	$t_{xm}(O) < t_{y0}(O)$ $t_{zm}(O) < t_{y0}(O)$	0 2
YZ	$t_{ym}(O) < t_{x0}(O)$ $t_{zm}(O) < t_{x0}(O)$	1 2

Tabulka č.(1) - převzato z [3] porovnání rovin a vracení prvniho oktantu

Rodič potomka	Opuší rovinu YZ	Opuší rovinu XZ	Opuší rovinu XY
0	4	2	1
1	5	3	Venku z oktantu
2	6	Venku z oktantu	3
3	7	Venku z oktantu	Venku z oktantu
4	Venku z oktantu	6	5
5	Venku z oktantu	7	Venku z oktantu
6	Venku z oktantu	Venku z oktantu	7
7	Venku z oktantu	Venku z oktantu	Venku z oktantu

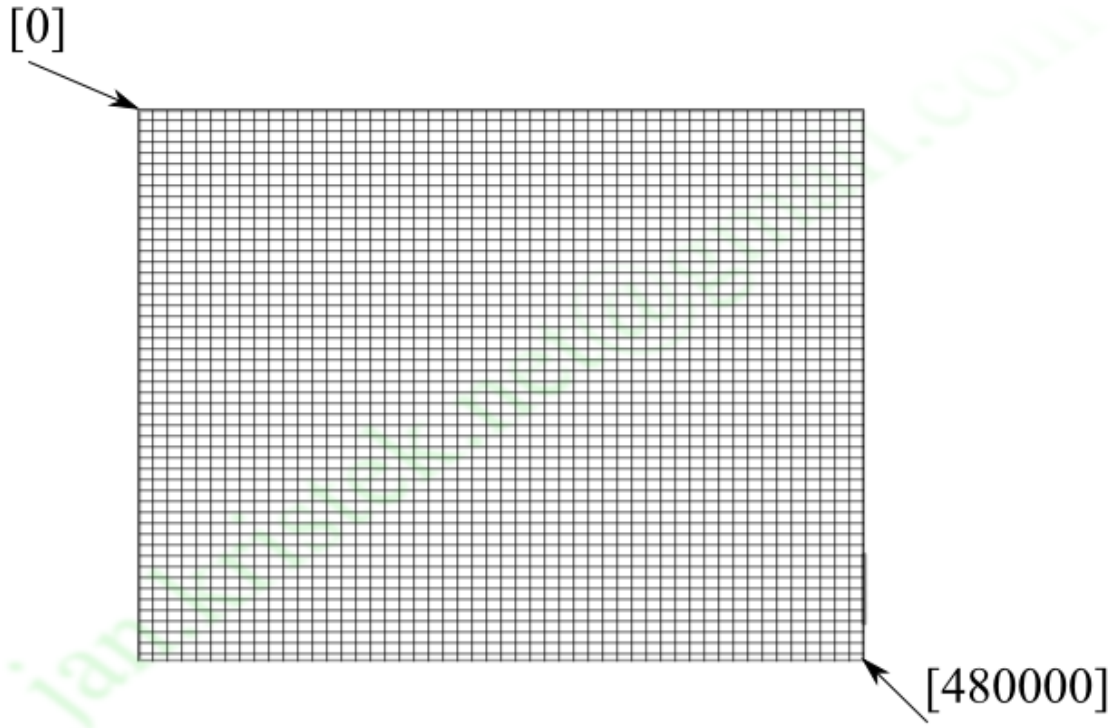
Tabulka č.(2) - převzato z [3] porovnání rovin a vracení dalšího oktantu



Obrázek č. (9) - Octree traverzace vyzobrazena pomocí trial Rhino

3 Paralelní zpracování

Paralelní zpracování trasovaných paprsků nám umožní zvýšit vykreslování výsledného obrazu až o jednotky řádů. S použitím grafických akcelerátorů, dokážeme tuto akcelerační metodu, využít na maximum. Na trhu existuje a existovalo mnoho výrobců grafických akcelerátorů. Mnoho z nich vzniklo pro účely počítačových her, což vedlo k psychologickému bloku, používat je i k jiným účelům. S příchodem GPGPU roku 2003, došlo k revolučnímu objevu. Pro jednoduchost si představme obdélník o rozměrech 800x600 pixelů, které posléze promítáme na obrazovku.

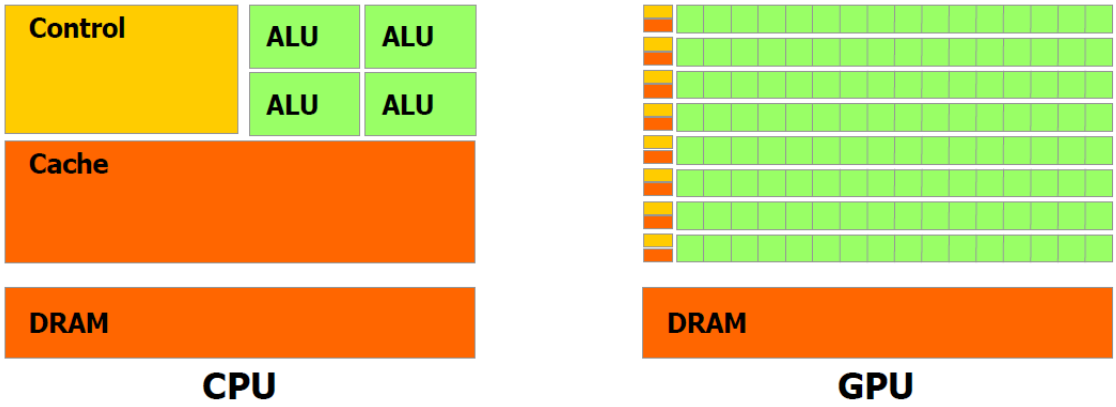


Obrázek č. (10) - GPGPU při zobrazované matici 800x600

Budeme-li se bavit o metodě trasovaní paprsků, tak v našem případě každý z pixelů je jeden paprsek, kterému se následně počítá barva podle seskupení těles v prostoru scény.

Výhody použití GPU:

1) GPU umožňuje až 512 vláknům běžet současně, nebavíme se o virtuálním běhu (stovky tisíc vláken).
2) Vlákna GPU musí splňovat podmínku nezávislosti, programátor nemůže manipulovat s pořadím zpracovávání jednotlivých vláken.
3) GPU není vhodné pro kód s velkým výskytem podmínek, ale k intenzivnímu počítání.
4) GPU je vhodně upraveno pro sekvenční přístup do paměti až194GB/s.
5) Gpu umožňuje podporu cache paměti.



Obrázek č. (11) - Převzat z Nvidia.com zobrazuje větši paralelizaci GPU oproti CPU

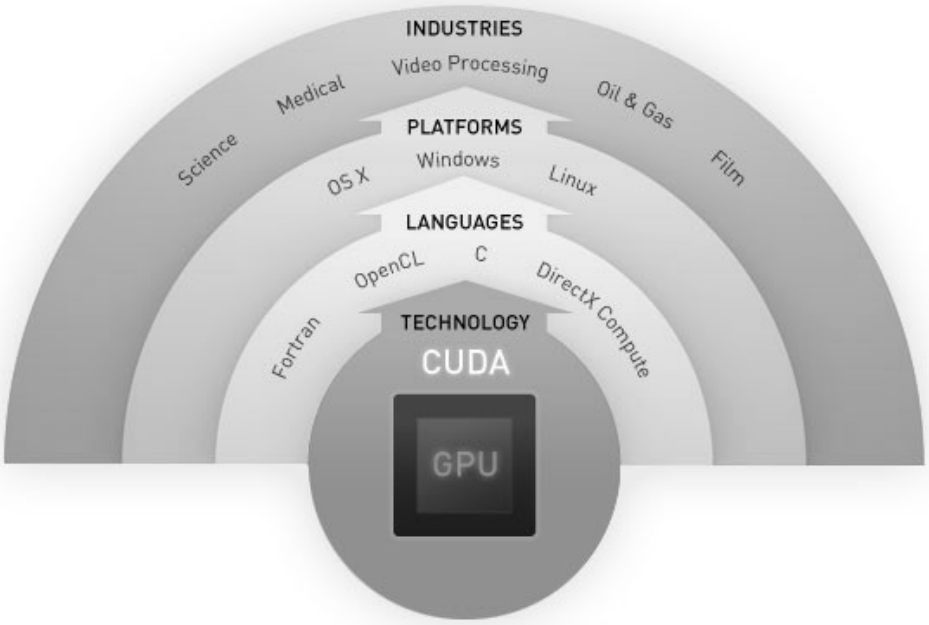
Jeden z prvních produktů pro využívání tzv virtuálního GPGPU byl průmyslový standart OpenGL (Open Graphics Library) specifikující multiplatformní rozhraní API (Application Programming Interface). Nevýhodou rozhraní tkví v implementaci na grafický hardware, který lze sice použít, ale musíme počítat se sníženým výkonem. Základní funkčnost OpenGL je ve vykreslování různých základních primitiv (bod, úsečka, mnohoúhelník) do obrazového rámce (framebuffer). Primitiva jsou definované za pomocí tzv. Vertexů (vrcholů) - jsou to body v prostoru scény definujíci koncový bod hrany nebo vrchol mnohoúhelníků. APIOpenGL je založeno na architektuře client-server – klient (program) vydává příkazy a server (grafický akcelerátor) vykonává. Z toho plyne, že tato architektura umožňuje využívat cloud(ON-Live) - HPC(High-Performance computing) složená z grafických akcelerátorů představující server, uživatel jakožto klient , na kterém se vykonávají příkazy a komunikace se provádí za pomocí počítačové sítě.

Postupem času došlo k tomu, že velké grafické korporace vyvinuly své vlastní programovací prostředí pro vývoj aplikací.

1) Nvidia CUDA– Jedna z prvních průkopníku se stala společnost Nvidia sídlící v Californii (U.S.) , přišla v roce 2006 na trh s technologií CUDA (Compute Unified Device Architecture). O této technologií se budu více věnovat v další podkapitole 3.1.
2) ATI Stream – Ve stejném čtvrtletí, kdy se objevila CUDA, přišla s alternativou firma AMD (tehdejší ATI) narychlo zkonstruovaným rozhraním CTM (Close To Metal), se svou podporou, funkcemi a stavem, které nebylo konkurence schopné vytlačit technologii CUDA z popředí. V brzké době bylo toto rozhraní nahrazeno technologií ATIStream, založeném na jazyku Brook+. Firma AMD udělala nešťastné rozhodnutí, povolila toto rozhraní jen na svém profesionálním grafickém akcelerátoru FireStream a tudíž běžný uživatel nemohl toto rozhraní použít. Její názor se změnil z důvodů velkého rozmachu architektury CUDA mezi běžnými uživateli v roce 2009, důvodem mohlo být i nové uvedení rozhraní OpenCL.
3) Open CL – Tento programátorský jazyk byl vyvinut firmou Apple, jakožto zamýšlený doplněk k OpenAI a OpenGL. Základem pro OpenCL byla technologie CUDA. V polovině roku 2008 pracovní skupina Khronos Compute Group tlačila na společnosti jako jsou Nvidia, IBM, INTEL a AMD, aby z toho rozhraní udělaly standart pro své technologie. Za necelých 5 měsíců se tyto firmy rozhodly, že tento jazyk podpoří na svých platformách.

3.1 CUDA

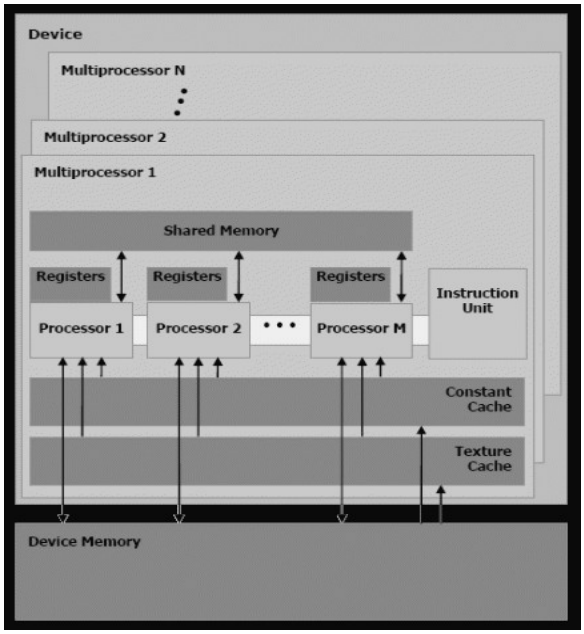
Na obrázku/(((() lze vidět, že tato architektura podporuje i jiné jazyky a API, kupříkladu C, OpenCL, Fortran nebo DirectX. Z této rozmanité množiny můžeme usoudit, že zmiňovaná architektura je plně multiplatformní, tedy dá se použít na operačních systémech jako je např. Windows, Linux nebo OS X. Konkrétně má bakalářská práce byla naimplementována v jazyce C na platformě Microsoft Windous XP. CUDA se dá používat v mnoha odvětví jako jsou například medicína, věda, zpracování obrazu, film, apod.



Obrázek č. (12) - Převzat z root.cz zobrazení flexibility technologie CUDA

Z pohledu Hardwaru je jádro GPU rozdělené na tyto části:

1) multiprocesor – obsahuje 32 výpočetních jader, 32k 32 bitových registrů. 64KB SRAM. Tento multiprocesor dokáže provést na jeden takt u floatu, na dva takty u double operaci FMA. Obsahuje 16 jednotek pro zápis a načítaní dat. Umožňuje převádění dat mezi různými typy např. int a float. 4 speciální jednotky se starají o výpočet složitějších funkcí sin, cos, tan,a exp.



Obrázek č. (13) - Převzat z Nvidia.com zobrazuje multiprocesory

Dále obsahuje různé typy pamětí optimalizované na konkrétní data pro práci:

1) Globální paměť - Tuto paměť můžeme nazvat jako hlavní paměť grafického akcelerátoru, dá se přirovnat k hlavní paměti počítače. Pro každé vlákno platí, že z této paměti může číst data, ale i do ní zapisovat. Nevýhoda této paměti tkví v nepoužití cachovaní přístupu. Hostitelský program dokáže kopírovat data globální paměti do hostitelské paměti a naopak. Jen hostitelský program dokáže alokovat tuto pamět na grafickém akcelerátoru. Tato pamět může dosahovat až 3GB.
2) Registry - Registry můžeme považovat za hlavní pamět pro proměnné kernelu. Na přístupovou cenu se ve většině případů nemusí brát ohled, z důvodů zahrnutí v množství cyklů potřebných pro vykonání jedné instrukce. Počet registrů je zahrnut ve specifikaci podle výpočetní schopnosti multiprocesoru, jejich maximální počet na verzi 2.x je stanoven na 32768.
3) Lokální paměť - V případech, kdy kernel přesáhne určitého množtví proměnných, dochází k nedostatku počtu registrů a tudíž dochází k přemístění části proměnných do této paměti. K přesunu paměti dochází automaticky, je zcela pod režií překladače. Programátor tento přesun nemůže jakkoliv ovlivnit. Takový přesun můžeme považovat za negativní vliv na rychlost programu, z důvodu rychlosti lokální paměti, která je totožná s globální.

4) Sdílená paměť - Hlavní učel této paměti je založen na sdílení dat mezi vlákny. Tato paměť umožňuje lepší komunikaci mezi nimi. Rozděluje se až do 32 tzv. bank. Rychlost přístupů do paměti je rovnocena s registry pod podmínkou, že každé vlákno přistupuje do jiné z bank. Multiprocesor obsahuje malé množství paměti až (48 KB ve verzi 2.x), která je sdílená všemi proudovými multiprocesory.
5) Pamět konstant - Jedná se o paměť, která se používá pro hodnoty, které jsou známe před spuštěním samotného jádra. Pro zrychlení je tato paměť cachovaná, její rychlost je totožná s registry, pokud jsou požadovaná data umístěna v cache. Paměť dokáže obsluhovat pouze jeden požadavek v jednom kroku.

6) Cache textur - Pro zrychlení přístupu k texturám uložených v hlavní paměti slouží tato Cache. Tato pamět se hodí i pro rychlé zapisování a čtení z paměti, která je srovnatelná s registry.

Výhody:

1) S touto technologií přišlo výrazné ulehčení práce s GPGPU.
2) Pro práci nepotřebujeme používat OpenGL.
3) Tento jazyk je založen na rozšířeni C/C++.

Nevýhody:

4) Pro práci je nutné mít grafický akcelerátor od společnosti Nvidia.

Rozdělení operací na jednotlivá zařízení pomocí režimu kernelu:

5) CUDA host – Kernel používá pro výpočty CPU a pro data operační paměť.
6) CUDA device – Toto zařízení je přizpůsobeno pro paralelní zpracování až stovek tisíc nezávislých vláken tzv. Threads. Thread je velmi jednoduchá struktura, která se dokáže vytvářet velmi rychle a také se s vysokou rychlostí přepíná při zpracování.

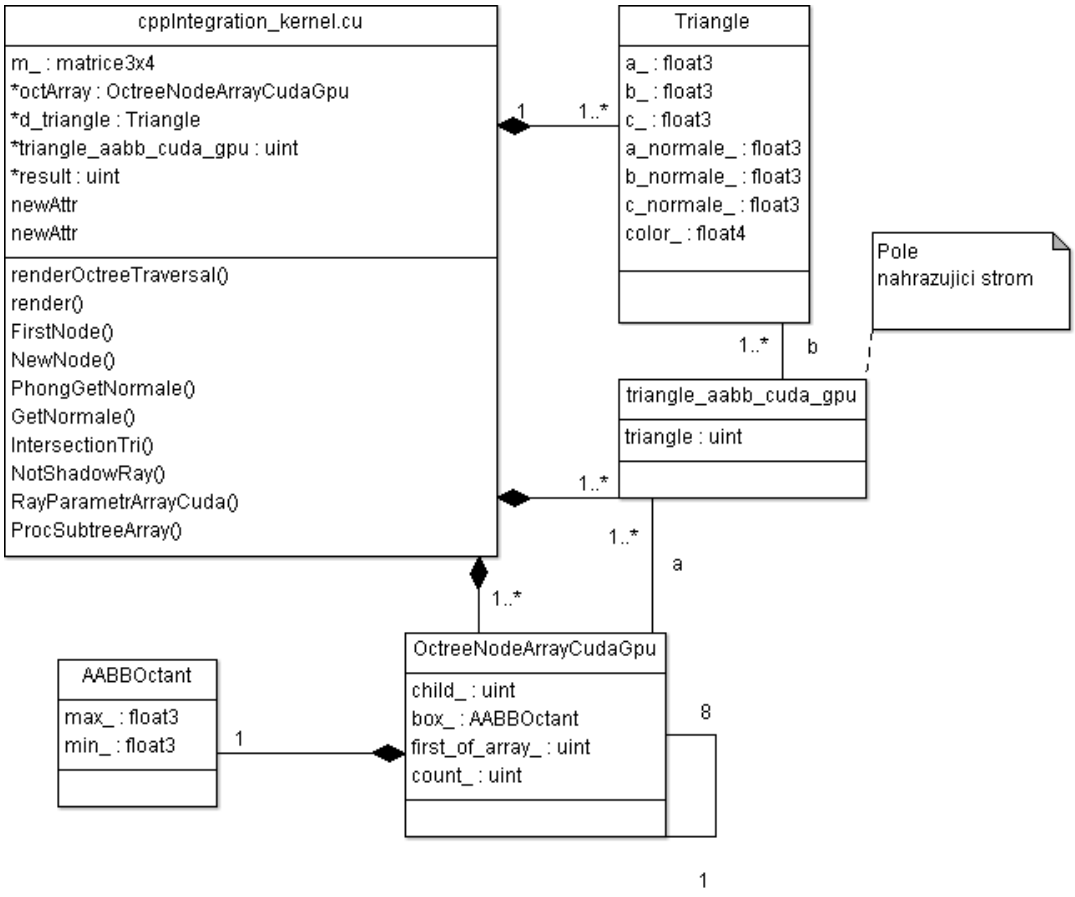
4 Implementace

Před popisem implementace si musíme zmínit o pár základních pravidlech. Nvida CUDA zavedla pro lepší práci s vlákny jednotku block, z důvodů lepší synchronizace a komunikace mezi jednotlivými vlákny. Jeden block může obsahovat až 1024 vláken . Každý ze skupiny bloků se zpracovává na jednom multiprocesoru. Vlákna v bloku sdílejí velmi rychlou pamět s malou latencí. Bloky vláken se potom seskupují do tzv. Grid (mřížek).

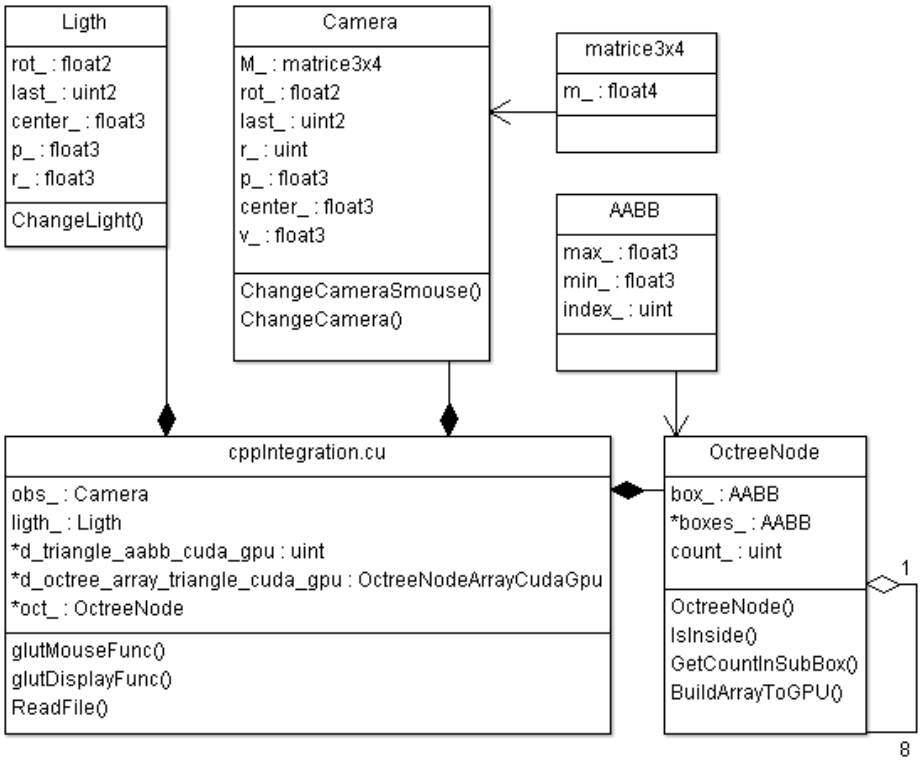
Programování v CUDA je založené na psaní kernelů (kernels), jsou to instrukce pro zpracovávání jedním z vláken.

Pro práci s technologií CUDA musíme znát:

1) Kernely nepodporují rekurzivní volání.
2) Větvení kódu kernelu má neblahý vliv na efektivitu.
3) Jejich parametry nemůžeme použít jako reference.
4) Dokáží použit šablony C++.
5) CUDA 2.x podporuje funkci printf(), pro debugování kernelu.
6) Redukovat přenos dat mezi CUDA host (CPU) a CUDA device (GPU) z důvodů komunikace, PCI-Express je sice optimalizována (pipeline), ale stále pomalá méně než 5 GB/s.
7) Dokonale optimalizovat přístup do globální paměti.
8) Omezit množství divergentních vláken.
9) Optimalizovat velikost bloks.



Obrázek č. (14) - UML diagram – kernelu (GPU)



Obrázek č. (15) - UML diagram – (CPU)

4.1 GPU

a_, b_, c_	float3	Hrany trojúhelniku.
a_normale,b_normale, c_normale	float3	Normály k daným hranám.
color_	floatč	Barva a materiál trojúhelniku.

Tabulka č.(3) - Struktura triangle (primitivum).

max_	float3	Maximální hodnota oktantu.
min_	float3	Minimální hodnota oktantu.
index_	uint	Indexy trojúhelníků

Tabulka č.(4) - Struktura AABB (ohraničující obálka mesh objektu).

max_	float3	Maximální hodnota oktantu.
min_	float3	Minimální hodnota oktantu.

Tabulka č.(5) - Struktura AABBOctant (ohraničující obálka mesh objektu, použita na CUDA).

child_	uint	První z dětí oktantu odstranění stromové struktury.
first_of_array_	uint	První z trojúhelníků v prostoru oktantu.
count_	uint	Počet trojúhelníků v prostoru oktantu.
box_	AABBOctant	Vyjádření octantu v prostoru.

Tabulka č.(6) - Struktura OctreeNodeArrayCudaGpu (použita pro trasovaní na CUDA).

m_	float4[3]	3D kamera.
octArray	*OctreeNodeArrayCudaGpu	Seznam oktantů.
d_triangle	*Triangle	Seznam trojúhelniků.
triangle_aabb_cuda_gpu	*uint	Seznam trojúhelniků pro oktant.
result	*uint	Výsledné GPGPU.

Tabulka č.(7) - Kernel CUDA cppIntegration_kernel (samostatný kernel pro každé vlákno z GPGPU), v obsahu tabulky jsou důležité data pro samotné zobrazovaní prostoru scény.

renderOctreeTraversal()	Hlavní kernel pro renderovaní scény a použití Octree.
render()	Hlavní kernel pro renderovaní scény za použití sdílené paměti.
PhongGetNormale()	Získaní normál pro Phong Shading.
GetNormale()	Získaní normál pomocí cross product.
IntersectionTri()	Test průsečíku trojúhelníku a paprsku.
NotShadowRay()	Test pro zastínění objektu jiným tělesem.
ProcSubtreeArray()	Hlavní funkce pro traverzaci v octree.
FirstNode()	Vedlejší funkce pro traverzaci v octree.
NewNode()	
RayParametrArrayCuda()	

Tabulka č.(8) - kernel CUDA cppIntegration_kernel (samostatný kernel pro každé vlákno z GPGPU), v obsahu tabulky jsou důležité funkce pro samotné zobrazovaní prostoru scény.

4.2 CPU

Obs_	Camera	Kamera v prostoru scény.
ligth_	Ligth	Světlo v prostoru scény.
octArray	*OctreeNodeArrayCudaGpu	Seznam oktantů.
d_triangle	*Triangle	Seznam trojúhelniků.
triangle_aabb_cuda_gpu	*uint	Seznam trojúhelniků pro oktant.

Tabulka č.(9) - OpenGL cppIntegration (spuštění kernelu v prostředí CPU), v obsahu tabulky jsou důležité data pro manipulaci se scénou.

glutMouseFunc()	Funkce pro pohyb myší.
glutDisplayFunc()	Funkce pro vykreslování scény.
ReadFile()	Načtení mesh modelu z disku.

Tabulka č.(10) - OpenGL cppIntegration (spuštění kernelu v prostředí CPU), v obsahu tabulky jsou důležité funkce pro manipulaci se scénou.

box_	AABB	Kořenový uzel (oktant) popsán AABB.
boxes_	*AABB	Potomci (oktanty) kořenového uzlu popsané AABB
count_	uint	Počet trojúhelníku v daném oktantu.

Tabulka č.(11) - Třída OctreeNode (vytvoří strukturu vhodnou pro zobrazování na CUDA), v obsahu tabulky jsou důležitá data pro samotné rekurzivní vytvoření stromové struktury. A následné konvertace na formát vhodný pro CUDA.

OctreeNode()	Vytvoření stromu dle zadaných zákonitostí.
IsInside()	Funkce pro zjištění zda libovolný trojúhelník náleží danému uzlu.
GetCountInSubBox()	Zjištění počtu libovolných trojúhelníků náležící danému uzlu.
BuildArrayToGPU()	Transformování stromové struktury do struktury vhodné pro Nvida CUDA

Tabulka č.(12) - Třída OctreeNode (vytvoří strukturu vhodnou pro zobrazování na CUDA), v obsahu tabulky jsou důležité funkce pro samotné rekurzivní vytvoření stromové struktury. A následné konvertace na formát vhodný pro CUDA.

M_	matice3x4	Matice pro zajištění kamery ve scéně
rot_	float2	Proměnné pro pohyb kamery za pomocí glutMouse.
last_	float2	
r_	float	Poloměr koule, po které se pohybuje kamera.
p_	float3	Bod, ve kterém je kamera v prostoru
center_	float3	Střed bodu (koule), na který se kamera dívá.

Tabulka č.(13) - Důležitá data třídy reprezentující kameru v prostoru scény

ChangeCameraS()	Přepočítávání kamery při pohybu myší

Tabulka č.(14) - Důležité funkce pro kameru v prostoru scény

rot_	float2	Proměnné pro pohyb kamery za pomocí glutMouse.
last_	float2	
r_	float	Poloměr koule, po které se pohybuje světlo.
p_	float3	Bod, ve kterém je světlo v prostoru
center_	float3	Bod střed koule, po kterém světlo putuje.

Tabulka č.(15) - Důležitá data třídy reprezentující světlo v prostoru scény

ChangeLigth	Přepočítávání světla při pohybu myší

Tabulka č.(16) - Důležité funkce pro světlo v prostoru scény

Pro debugování byl použit režim host, data z tohoto testování byla zobrazena v programu Trial Rhino.

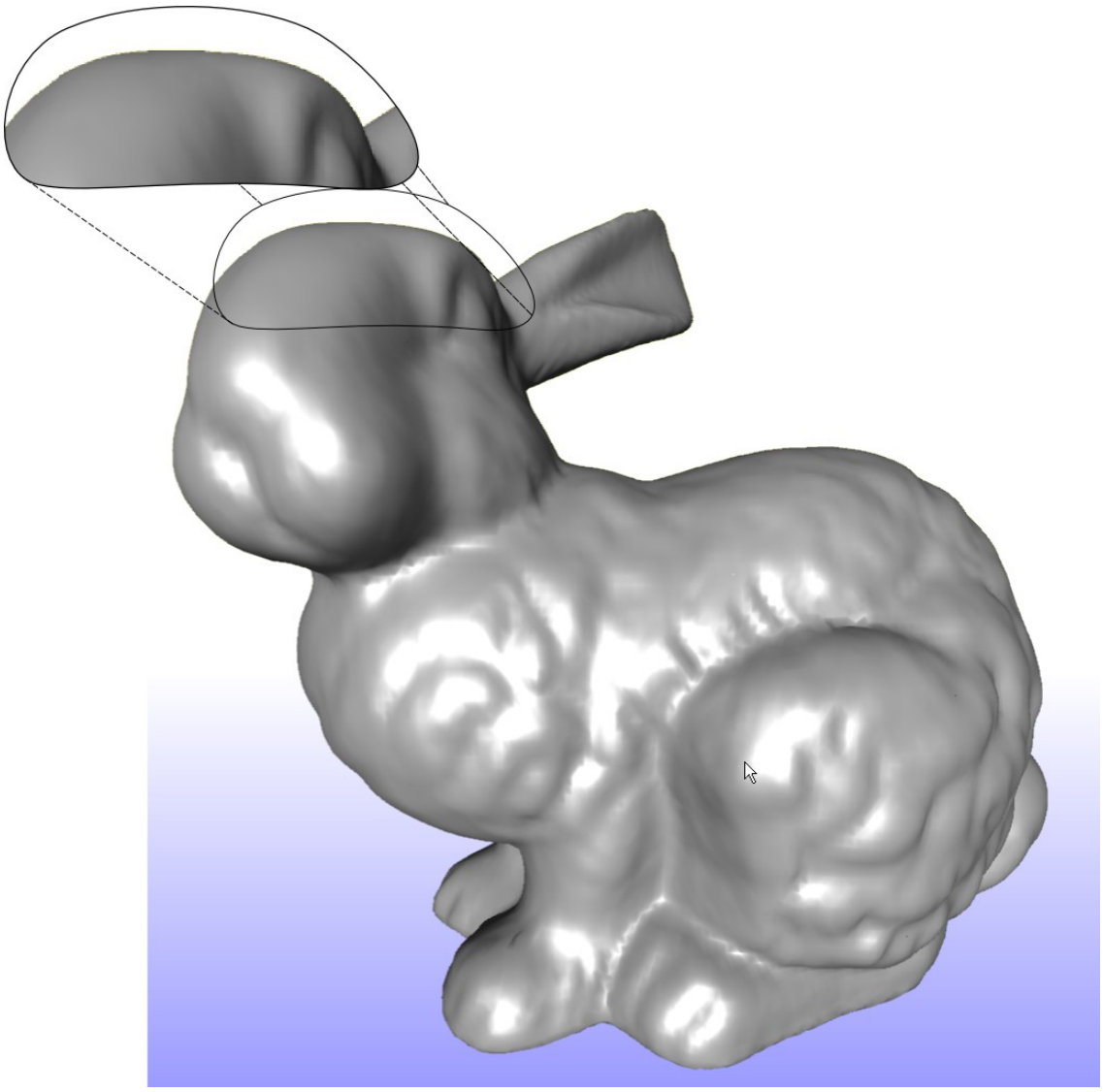
5 Testování

Pro testování mého ray tracingu za pomocí technologie CUDA, bylo použito formátu mesh modelu .ply, který používá Univerzita Standford pro své skenování 3D prostoru. Tento formát jsem konvertoval do mé struktury, vyjadřující trojúhelník v mém prostoru scény. Poté jsem testoval různé technologie pro zlepšení výsledného obrazu, které byly implementované např. dělení prostoru scény, antialiasing (4xAA), phong shading, atd.

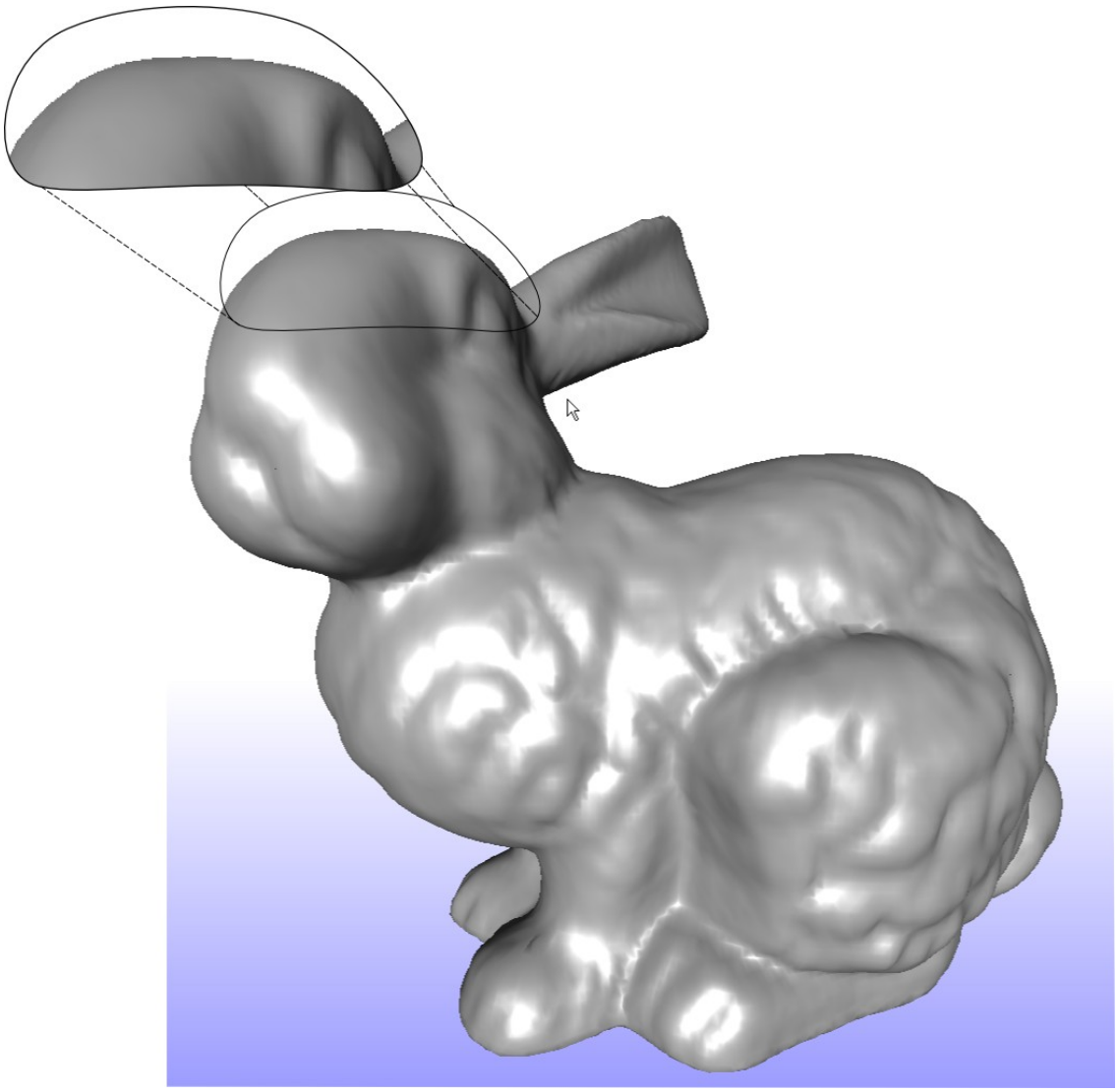
Mesh modely byly zkonstruované za pomocí programu Blender, některé z nich byly převzaty z Univerzity Standford a poté převedené na požadovaný formát, pro další zpracování. Tento program dokázal výpočet normál pro Phong shading. Tyto hodnoty byly použity pro výsledné zobrazování Phong shadingu. GPGPU je 1280x968 tomu odpovídá 1 239 040 nezávislých vláken.

Typ zpracování	Bez zlepšení obrazu	Phong shading	antialiasing	Phong shading + antialiasing
Bunny (69 666 triangles)	0,94	0,93	0,28	0,26
Balls (4032 triangles)	1,85	1,54	0,58	0,58

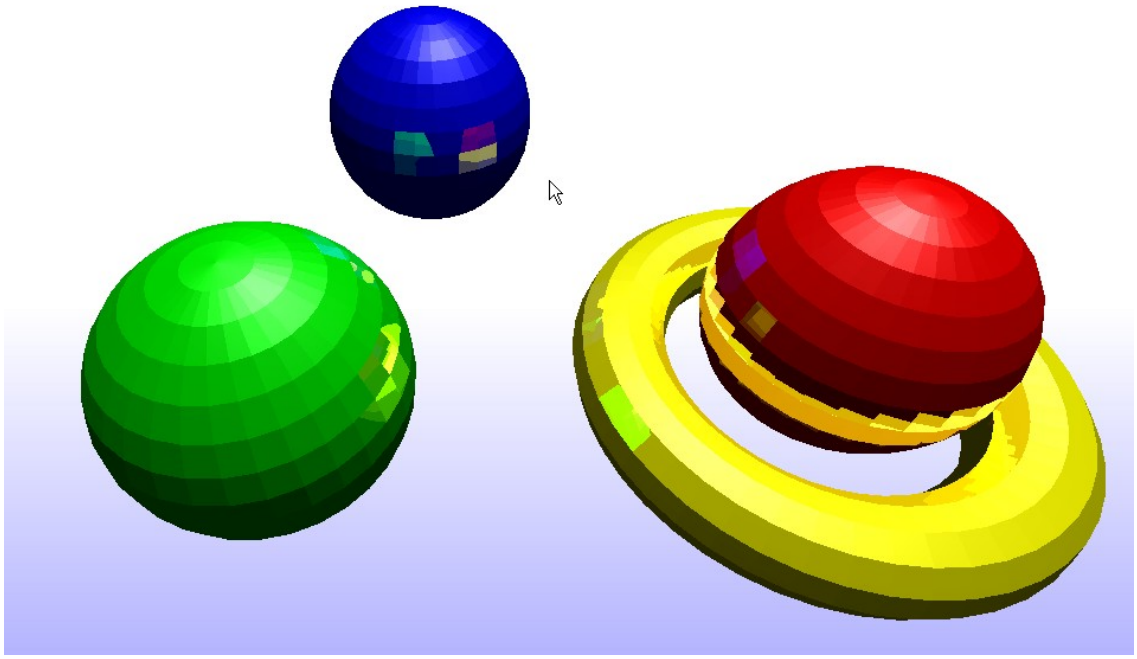
Tabulka č.(17) - Výsledné hodnoty testů



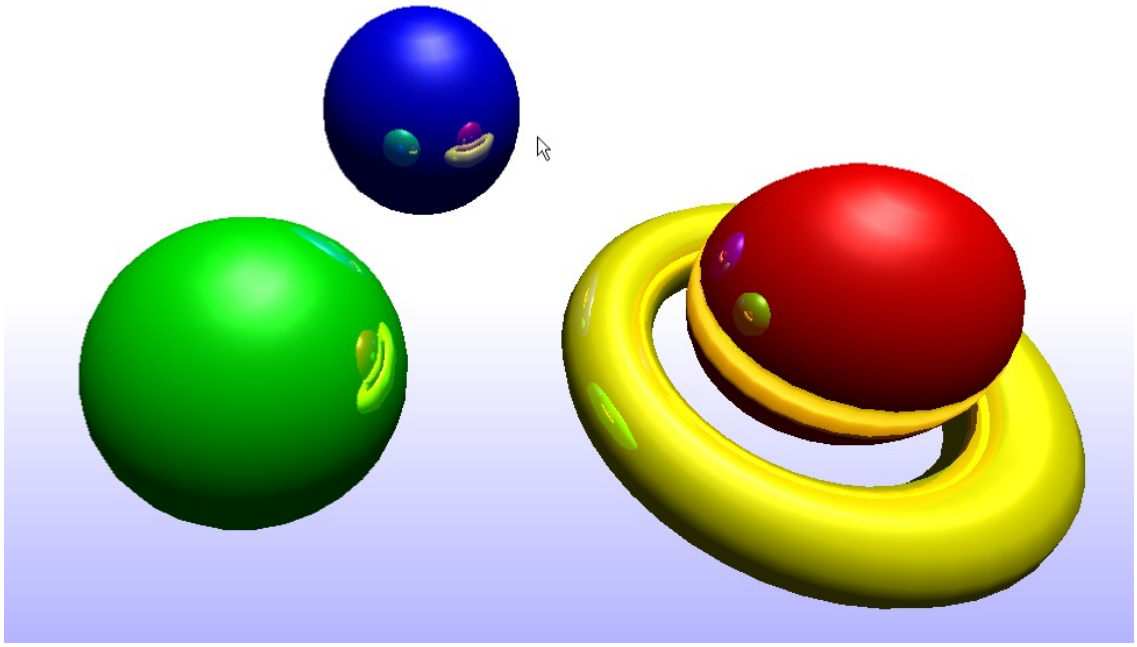
Obrázek č. (16) - bunny.ply (PS, 4xAA) - můžeme pozorovat zlepšení antialialiazingu



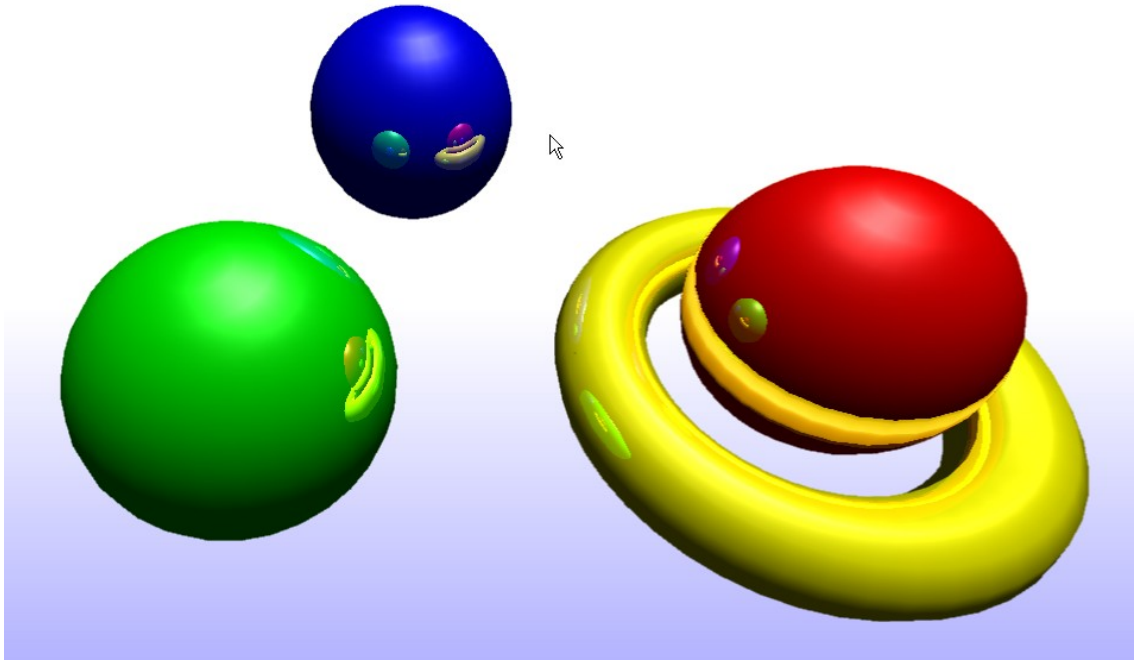
Obrázek č. (17) - bunny.ply (PS) – pozorujeme vliv bez antialiazingu



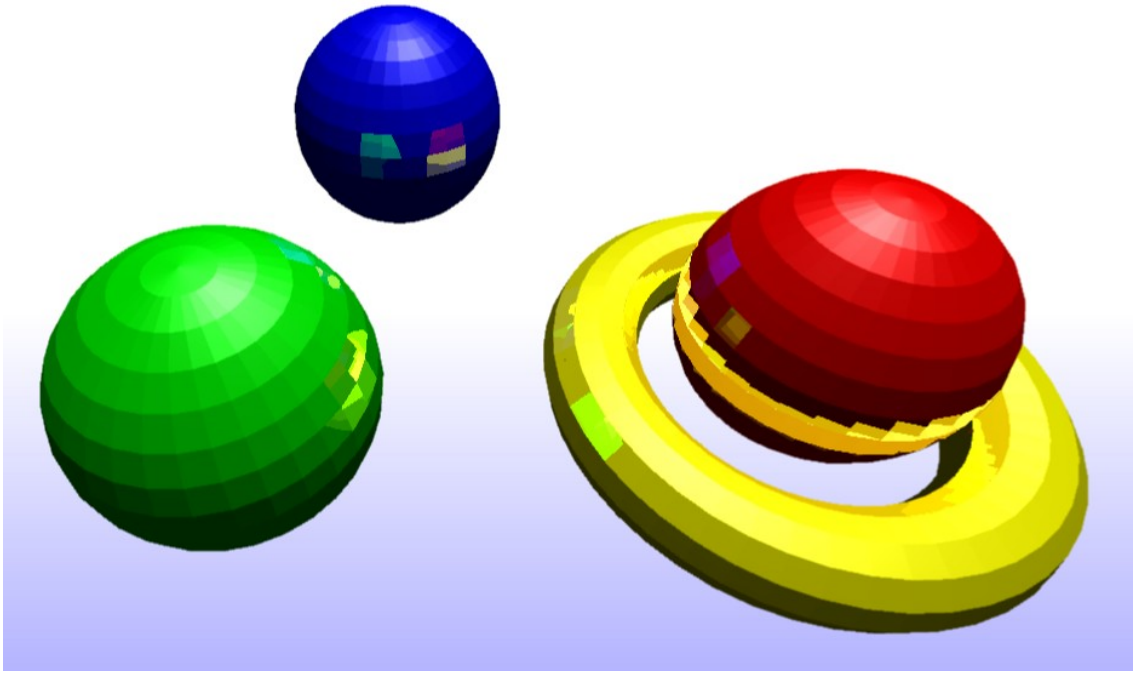
Obrázek č. (18) - balls.ply – pozorujeme vliv bez Phong shading



Obrázek č. (19) - balls.ply (PS) – pozorujeme vliv Phong shading



Obrázek č. (20) - balls.ply (4xAA) – pozorujeme vliv antialiasingu



Obrázek č. (21) - balls.ply (4xAA, PS) – pozorujeme vliv antialiasingu a Phong shading

6 Závěr

Touto bakalářskou prací jsem dokázal, že lze zobrazovat interaktivní ray tracing v počítačové grafice. Naimplementoval jsem pomocí optimalizačních postupů (Octree, Nvidia Cuda, Moller, C/C++) a postupů pro zlepšení výsledného obrazu (4x antialiasing, Phong shading, samotný ray tracing). Možné implementace do budoucna:

1) Použití Kinect for Windows SDK pro interaktivní ovládání pomocí snímání lidského těla
2) Pasivní (polarizace) a aktivní (Nvidia 3D vision) 3D pro reálnější požitek
3) Bump mapping pro zjednodušení mesh modelu a lepší podání textur ve výsledném obrazu
4) Propojení více grafických akcelerátorů, technologie Nvidia SLI
5) Možnost použití HPC vybaveny technologií Nvida CUDA, pro možný cloud
6) Možný přechod na Nvidia CUDA 2.x nebo OpenCL
7) Pro operace na cpu na technologie DirectX 11, Microsoft Silverligth, Python, Html 5
8) Použití technologie Nvida PhysX SDK
9) Zavedení více druhů primitiv (bod (částice), úsečka, 4hran)

Z tohoto plyne, že nikdy nic není dokonalé a proto musíme získávat stále nové informace o technologiích používané v daném odvětví např. počítačová grafika.

7 Literatura

[1] Nvidia cuda, *NVIDIA CUDA C Programming Guide v3.2* 2011

[2] B. T. Phong, Illumination for computer generated pictures, Communications of ACM 18 (1975), no. 6, 311–317.

[3] J. Revelles, C. Urena, An Efficient Parametric Algorithm for Octree Traversal

[4] Leler, William J. (July 1980). "Human Vision, Anti-aliasing, and the Cheap 4000 Line Display".*ACM SIGGRAPH Computer Graphics* **14** (3): 308–313 http://dx.doi.org/10.1145%2F965105.807509

[5] Mark Harris. Mapping computational concepts to GPUs.

[6] Eduard Sojka, Počítačová grafika II http://mrl.cs.vsb.cz/people/sojka/pocitacova_grafikaII.pdf

[7] A. Chalmers, T. Davis, and E. Reinhard. Practical parallel rendering. AK Peters, Ltd.,

8 Příloha

Součástí práce je disk svýslednou implementací projektu. Struktura adresářů je následující

1 /cppIntegration - obsahující implementaci na GPU a CPU
2 /obrázky – výsledky testování v plné kvalitě.
3 /mesh modely ve formátu .ply – obsahují modely použité při renderování.