VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RAY TRACING NA ARCHITEKTUŘE CUDA

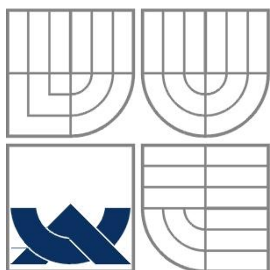
BAKALÁŘSKÁ PRÁCE

BACHELOR‘S THESIS

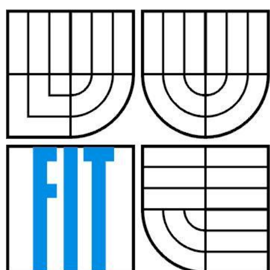
AUTOR PRÁCE LUKÁŠ BIDMON

AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

RAY TRACING NA ARCHITEKTUŘE CUDA

RAY TRACING ON CUDA ARCHITECTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ BIDMON

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. DAVID BAŘINA

BRNO 2010

Abstrakt

Tato práce se zabývá využitím grafických karet podporujících CUDA pro výpočet ray tracingu. Nejdříve je představen klasický rekurzivní algoritmus pro ray tracing a je uveden matematický základ, použitý při výpočtech, pro implementovaná tělesa. Dále je představena architektura nVidia CUDA a jsou uvedeny odlišnosti od výpočtů prováděných na CPU. Následuje návrh algoritmu, kde jsou rozebrány úpravy nutné pro běh na GPU. Část o implementaci se zabývá průběhem programu a využitím paměti. Nakonec jsou uvedeny výsledky testování a porovnání výkonu CPU a GPU implementace.

Abstract

This work presents utilization of CUDA capable graphic cards for ray tracing. First, the classic recursive ray tracing algorithm is presented and necessary math is explained for implemented objects. nVidia CUDA architecture is introduced in next chapter with explained differences from CPU computations. Following is the implementation scheme where modifications necessary for CUDA are discussed. Implementation chapter covers details about flow of the program and memory usage. Finally the CPU and GPU testing results are presented.

Klíčová slova

Ray tracing, ray casting, nVidia CUDA, grafická karta, GPU, metoda sledování paprsku, generování obrazu, paprsek, odraz paprsku, lom paprsku, průnik paprsku

Keywords

Ray tracing, ray casting, nVidia CUDA, graphic card, GPU, image generation, ray, reflection, refraction, ray sphere intersection, ray triangle intersection

Citace

Bidmon Lukáš: Ray tracing na architektuře CUDA, bakalářská práce, Brno, FIT VUT v Brně, 2010

Ray tracing na architektuře CUDA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Davida Bařiny
Další informace mi poskytl Ing. Petr Pospíchal
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Lukáš Bidmon
18. května 2010

Poděkování

Poděkování patří Ing. Davidu Bařinovi a Ing. Petru Pospíchalovi za cenné informace při vytváření této práce.

© Lukáš Bidmon, 2010
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah ... 1
1 Úvod ... 3
2 Ray tracing ... 4
2.1 Princip základního ray tracing algoritmu ... 4
2.1.1 Pseudokód klasického rekurzivního algoritmu ... 5
2.2 Základní objekty ... 6
2.2.1 Paprsek ... 6
2.2.2 Koule ... 6
2.2.3 Trojúhelník ... 8
2.3 Povrch objektu ... 9
2.3.1 Odraz paprsku ... 10
2.3.2 Lom paprsku ... 10
2.3.3 Difúzní odraz ... 11
2.3.4 Spekulární odraz ... 11
3 nVidia CUDA ... 12
3.1 Popis ... 12
3.1.1 Hardwarový model ... 12
3.1.2 Softwarový model ... 13
3.2 Odlišnosti od výpočtů na CPU ... 14
4 Návrh algoritmu ... 15
4.1 Možnosti programu ... 15
4.1.1 Formát vstupních dat ... 15
4.2 Paralelizace ... 16
4.3 Odstranění rekurze ... 16
5 Implementace ... 18
5.1 Struktura kódu ... 18
5.2 Překlad kódu ... 18
5.3 Průběh programu ... 18
5.4 Dráha paprsku ... 20
5.5 Využití paměti ... 20
6 Testování ... 22
6.1 Testovací sestava ... 22
6.2 Způsob měření ... 22

6.3 Složitost scény 22
6.4 Porovnání s CPU verzí 25
7 Závěr 27
7.1 Zhodnocení výsledků 27
7.2 Možnosti dalšího vývoje 27
Literatura 28
Seznam příloh 29
Vzorové výstupy programu 30
Popis souboru pro uložení scény 35

1 Úvod

Ray tracing neboli metoda sledování paprsku, je v počítačové grafice způsob generování obrazu, který je používán k vytvoření dvourozměrného zobrazení prostorové scény. Je jednou z několika možných metod renderování, jako je rasterizace nebo ray casting.

Pomocí ray tracingu je možné generovat velmi kvalitní fotorealistické obrazy, ale jedná se o metodu velmi výpočetně náročnou. Proto se v dnešní době používá hlavně tam, kde není důležitá rychlost zobrazení, ale jeho věrnost (filmové efekty, průmyslová vizualizace…). V zobrazování v reálném čase převládá rasterizace, která má ovšem hardwarovou podporu. Vývoji hardwarové podpory pro raytracing se věnovala například společnost Intel.

Myšlenka ray tracingu je založena na principu sledování světelných paprsků, které ve virtuální scéně, od zdroje světla dorazí k pomyslnému pozorovateli (kameře). Je sledována jejich interakce s objekty scény – v případě průniku paprsku s objektem to znamená, že na objekt dopadá světlo a může být určena jeho barva (dle fyzikálních vlastností povrchu objektu). U většiny objektů je nutné k určení barvy kombinovat několik různých paprsků. Nejvýstižnější přirovnání k reálnému světu je pravděpodobně k focení fotoaparátem, popřípadě způsob vnímaní světla naším okem.

Díky modelu šíření světla je dosaženo věrného zobrazení některých efektů (stíny, odrazy, lom světla…), které je nutné při použití jiných technik renderingu různými způsoby nahrazovat. Díky nezávislosti jednotlivých paprsků je také vhodným kandidátem pro paralelizaci.

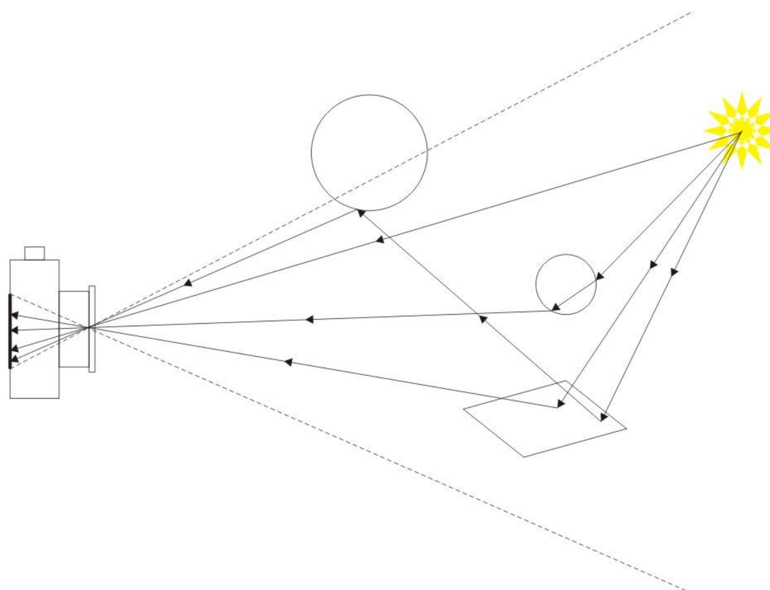
Mezi nevýhody základního algoritmu patří použiti pouze bodových zdrojů světla a ostré stíny u objektů. Dále také vlastnost, že lesklé plochy se nestávají druhotnými zdroji světla a také velká výpočetní náročnost.

2 Ray tracing

Tato kapitola seznamuje s principem klasického algoritmu pro ray tracing a zabývá se matematickými postupy použitými ve vyvíjené aplikaci. Je zde popsáno chování paprsku ve scéně, výpočet průsečíků se základními objekty a popsán postup při určení barvy paprsku.

2.1 Princip základního ray tracing algoritmu

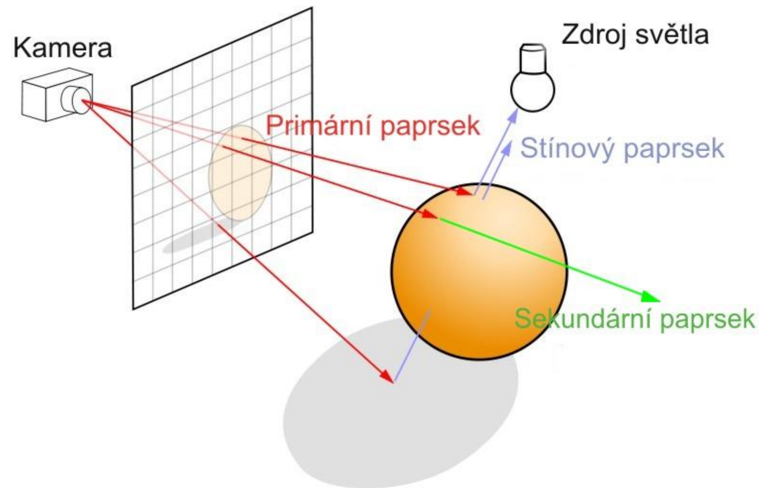
Algoritmus se snaží simulovat chovaní světla ve skutečném světě. Zdroj světla vyzařuje paprsky, které se šíří přímo prostorem, než narazí na překážku. Zde může dojít k několika událostem: pohlcení, odražení a lom. Podle typu povrchu může dojít ke kombinaci těchto událostí. Odražený paprsek pokračuje dále, a pokud dorazí až do místa kamery (oka), přispívá k tomu, co vidíme.



Obrázek 2.1: Odraz a lom paprsků – čočkou pronikne pouze část vyzářených paprsků

V počítačové grafice je cílem získat obraz scény složený z obrazových bodů (pixelů). Je tedy zapotřebí znát barvu paprsků, které směřují jednotlivými pixely do pozice kamery. Protože zdroje světla vyzařují prakticky nekonečné množství paprsků do všech směrů, není možné sledovat všechny. A také pouze velmi malá část paprsků se nakonec odrazí požadovaným směrem do kamery. Proto se v praxi postupuje opačně. Paprsky jsou vysílány směrem z kamery přes pixely generovaného obrazu do scény. Takto zjistíme jaký objekt (resp. bod objektu) je v daném pixelu vidět. Dále zjišťujeme, zda je tento bod ve stínu a pokud se jedná o objekt, který odráží nebo láme světlo, vytvoříme odpovídající pokračování původního paprsku. Takto mohou vzniknout tyto druhy paprsků:

- Primární (primary ray) – paprsky vyslané z kamery do scény
- Sekundární (secondary ray) – vzniklé odrazem nebo lomem paprsku
- Stínový (shadow ray) – vyslaný z průsečíku paprsku s objektem ke zdrojům světla, za účelem zjištění zda bod leží ve stínu.



Obrázek 2.2: Typy paprsků (převzato z [7])

Metoda, která pro výpočet barvy používá pouze primárních paprsků, se nazývá ray casting.

2.1.1 Pseudokód klasického rekurzivního algoritmu

```
pro každý pixel obrázku {
        vytvoř paprsek z pozice kamery procházející tímto pixelem
        nastav nejbližší_vzdálenost = nekonečno a nejbližší_objekt = NULL
        zavolej Sleduj_paprsek
}
Sleduj_paprsek {
        pro každý objekt ve scéně {
                pokud paprsek protne tento objekt {
                        pokud vzdálenost_průniku < nejbližší_vzdálenost {
                                nastav nejbližší_vzdálenost = vzdálenost_průniku
                                nastav nejbližší_objekt na tento objekt
                        }
                }
        }
        pokud nejbližší_objekt = NULL {
                pixel bude mít barvu pozadí
        }
        jinak {
                vyšli stínové paprsky ke zdrojům světla
                pokud je povrch zrcadlový, vytvoř odražený_paprsek a zavolej Sleduj_paprsek
                pokud povrch láme světlo, vytvoř lomený_paprsek a zavolej Sleduj_paprsek
                výsledná barva je součtem barvy osvětlení, odraženého a lomeného paprsku
        }
}
```

Jako omezení rekurze v algoritmu lze předem stanovit hloubku zanoření nebo nastavit hodnotu, jakou měrou musí paprsek přispět ke změně výsledné barvy.

Podstatnou část algoritmu tvoří testování na průnik paprsku s objekty na scéně. Objektem je zde myšlen nějaký základní grafický útvar, zpravidla trojúhelník nebo koule, ale obecně se může jednat o

3D objekt vytvořený různými metodami (CSG, šablonování, implicitní plochy…). Všechny složitější útvary na scéně jsou pak složeny kombinací těchto základních.

2.2 Základní objekty

V textu je použito toto značení:

$\cdot$	pro skalární součin
$\times$	pro vektorový součin
$*$	pro násobení

Důležitou součástí ray traceru jsou funkce pro výpočet průsečíku paprsku s objekty na scéně. Tato kapitola se zabývá způsoby nalezení průsečíku s objekty implementované ve vyvíjené aplikaci. Různé metody nacházení průsečíku jsou rozebírány v [1].

2.2.1 Paprsek

Prvním krokem je definování paprsku, který budeme testovat na průsečík s objektem. Paprsek je definován jako:

Výchozí bod

$$R_p = [R_x, R_y, R_z]$$

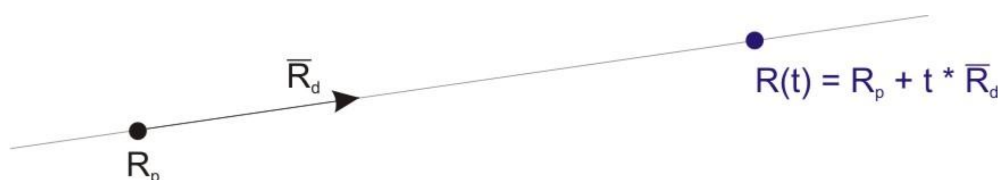
Směrový vektor

$$R_d = [R_{dx}, R_{dy}, R_{dz}] \text{ kde } R_{dx}^2 + R_{dy}^2 + R_{dz}^2 = 1$$

Pro všechny body ležící na paprsku platí:

$$R(t) = R_p + t * R_d$$

kde **t > 0** a značí vzdálenost bodu od počátku paprsku



Obrázek 2.3: Reprezentace paprsku

2.2.2 Koule

Koule je jedním ze základních tvarů používaných při ray tracingu.

Definována je středem $S_c = [S_{cx}, S_{cy}, S_{cz}]$ a poloměrem r.

Test na průsečík je poměrně nenáročný, a proto je koule vhodná také jako obálka (bounding volume) pro jiné objekty. Existuje více způsobů jak průsečík určit, ale z hlediska výpočetní náročnosti se používá zpravidla následující postup.

2.2.2.1 Výpočet průsečíku s paprskem

Postup zjištění zda paprsek protíná danou kouli je následující:

1. Určení zda počátek paprsku leží uvnitř/vně koule
2. Nalezení nejbližšího přiblížení paprsku ke středu koule t_c
3. Určení (umocněné) vzdálenosti t_s – nejbližšího přiblížení od povrchu koule
4. Určení vzdálenosti t od počátku paprsku
5. Výpočet souřadnic $[P_x, P_y, P_z]$ bodu průsečíku P

---

1. Vektor od počátku paprsku ke středu koule

$$V = S_c - R_p$$

Umocněná velikost vektoru V

$$V_s = V \cdot V$$

Pokud je V_s menší než r^2 (poloměr koule), paprsek má počátek uvnitř koule.

2. Nejbližší přiblížení paprsku ke středu koule

$$t_c = V \cdot R_d$$

pokud pro paprsek ležící vně koule platí $t_c < 0$ znamená to, že směřuje směrem od koule a nemůže ji protnout. V tom případě může být testování ukončeno již po dvou krocích výpočtu.

3. Vzdálenost D nejbližšího přiblížení paprsku ke středu koule

$$D^2 = V_s - t_c^2$$

Vzdálenost od povrchu koule

$$t_s = r^2 - D^2$$
$$t_s = r^2 - V_s + t_c^2$$

pro $t_s < 0$ paprsek kouli neprotne

4. Pro parsky začínající mimo kouli

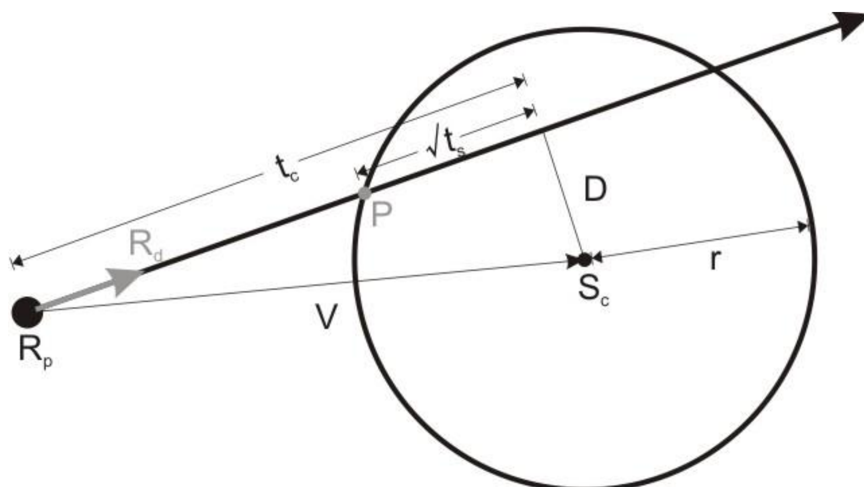
$$t = t_c - \sqrt{t_s}$$

pro paprsky začínající uvnitř koule

$$t = t_c + \sqrt{t_s}$$

5. $P = [P_x, P_y, P_z]$

$$P_x = R_x + R_{dx} * t$$
$$P_y = R_y + R_{dy} * t$$
$$P_z = R_z + R_{dz} * t$$



Obrázek 2.4: Nalezení průniku koule s paprskem

Pro další výpočty bude také zapotřebí znát normálu P_n v nalezeném bodě:

$$P_n = \left[\frac{\mathrm{P_x} - \mathrm{S_{cx}}}{\mathrm{r}}, \frac{\mathrm{P_y} - \mathrm{S_{cy}}}{\mathrm{r}}, \frac{\mathrm{P_z} - \mathrm{S_{cz}}}{\mathrm{r}}\right]$$

2.2.3 Trojúhelník

Dalším velmi používaným objektem je trojúhelník. Prvním krokem pro určení zda paprsek prochází trojúhelníkem (obecně jakýmkoli polygonem) je nalezení průsečíku s rovinou ve které trojúhelník leží.

2.2.3.1 Průsečík paprsku s rovinou

Hledáme bod P ležící na paprsku:

$$P = R_p + t * R_d$$

$R_p = [R_x, R_y, R_z]$ je výchozí bod a $R_d = [R_{dx}, R_{dy}, R_{dz}]$ normalizovaný směrový vektor

Rovina je dána rovnicí

$$P \cdot N + d = 0$$

Kde *P* je bod roviny

$$P = [P_x, P_y, P_z]$$

a *N* normála

$$N = [N_x, N_y, N_z] \text{ a platí } N_x^2 + N_y^2 + N_z^2 = 1$$

Dosazením získáme

$$(R_p + t * R_d) \cdot N + d = 0 \rightarrow t = \frac{-(R_p \cdot N + d)}{R_d \cdot N}$$

Pro $R_d \cdot N = 0$ průsečík neexistuje – paprsek je rovnoběžný s rovinou
Souřadnice bodu získáme dosazením t do původní rovnice paprsku
Nyní známe bod, ve kterém paprsek protíná rovinu trojúhelníku. Zbývá určit, zda leží uvnitř.

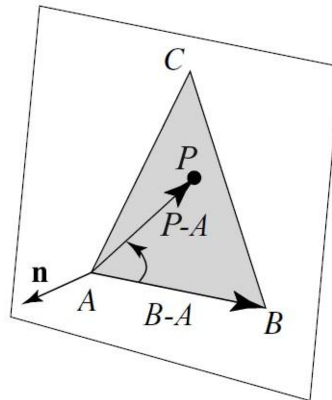
2.2.3.2 Poloha bodu vůči trojúhelníku

Trojúhelník je definován body A, B, C a normálou N.

$$A = [A_x, A_y, A_z]$$
$$B = [B_x, B_y, B_z]$$
$$C = [C_x, C_y, C_z]$$
$$N = [N_x, N_y, N_z]$$

Pro každou stranu pomocí vektorového součinu určíme, ve které polorovině se bod P nachází. Pro stranu AB: vytvoříme vektory $B - A$ a $P - A$ jejich vektorový součin by měl mít stejný směr jako normála trojúhelníku ABC neboli:

$$[(B - A) \times (P - A)] \cdot N \geq 0$$



Obrázek 2.5: Poloha bodu vůči trojúhelníku

Stejná podmínka platí i pro zbylé dvě strany. Má-li bod ležet uvnitř trojúhelníku, musí platit všechny podmínky současně:

$$[(B - A) \times (P - A)] \cdot N \geq 0$$
$$[(C - B) \times (Q - B)] \cdot N \geq 0$$
$$[(A - C) \times (Q - C)] \cdot N \geq 0$$

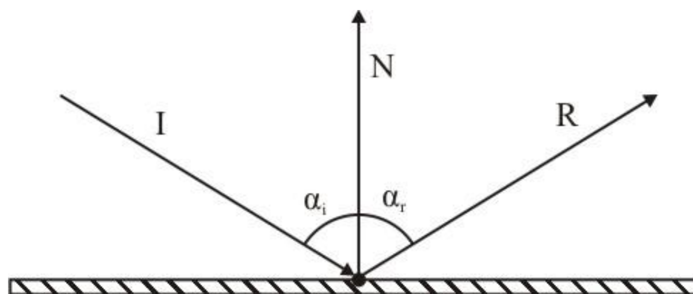
2.3 Povrch objektu

V reálném světě může po dopadu světla na povrch objektu dojít k jeho pohlcení, odražení nebo k lomu paprsku. K jakému efektu (nebo jejich kombinaci) dojde, závisí na fyzikálních vlastnostech materiálu. Pravděpodobně každý materiál část světla pohltí a tím ovlivní barvu odraženého paprsku. Ideálně hladký povrch odrazí paprsek pod stejným úhlem, pod jakým paprsek dopadne, zatímco

nepravidelný povrch může odrazit paprsek jakýmkoli směrem. Tyto dva typy odrazu nazýváme spekulární a difúzní (specular and diffuse reflection).

2.3.1 Odraz paprsku

Pro zrcadlové objekty potřebujeme pro určení jejich barvy znát pokračování primárního (příchozího) paprsku. Tak zjistíme, jaký objekt se bude „zrcadlit“. Vlastnosti odraženého paprsku R vyplývají z obrázku:



Obrázek 2.6: Odraz paprsku

Kde příchozí paprsek I svírá s normálou povrchu N úhel α_i. Úhel mezi odraženým paprskem R a normálou je $\alpha_r = \alpha_i$. Cílem je získat řešení pro odražený paprsek R, který je lineární kombinací N a I:

$$R = a * I + b * N$$

Také platí

$$\cos \alpha_i = -I \cdot N$$

$$\cos \alpha_r = N \cdot R$$

řešením dostáváme

$$-I \cdot N = N \cdot R$$

$$-I \cdot N = N \cdot (a * I + b * N)$$

Protože *N* má velikost 1 a pokud zvolíme $a = 1$

$$b = -2 * (N \cdot I)$$

rovnice pro *R* je tedy

$$R = I - 2 * (N \cdot I) * N \tag{1}$$

2.3.2 Lom paprsku

Směr paprsku přecházejícího z jednoho prostředí do druhého popisuje Snellův zákon. Pro výpočet směru lomeného paprsku je v aplikaci použita následující rovnice:

$$R = I * \frac{n_1}{n_2} + n * \left(\frac{n_1}{n_2} * (-I \cdot n) - \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 * (1 - (I \cdot n)^2)} \right)$$

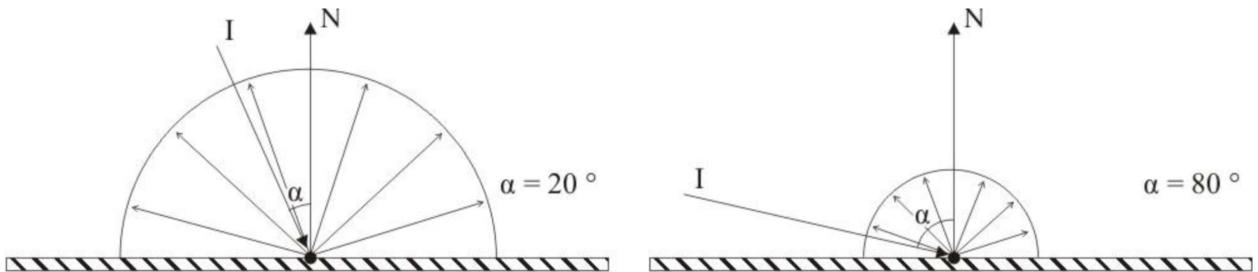
Kde:

I je dopadající paprsek

n je normála
n_1, n_2 jsou indexy lomu původního/nového prostředí

2.3.3 Difúzní odraz

K difúznímu odrazu dochází na materiálech s nepravidelným povrchem (např. plast), který dopadající paprsky rozptýlí rovnoměrně do všech směrů. Intenzita odražených paprsků je závislá pouze na úhlu dopadajícího světla. Čím menší je úhel mezi paprskem a normálou, tím větší je intenzita odraženého světla.



Obrázek 2.7: Difúzní odraz

Barvu odraženého paprsku získáme tak, že vynásobíme barvu objektu difúzním koeficientem, což je kosinus úhlu mezi dopadajícím paprskem světla a normálou:

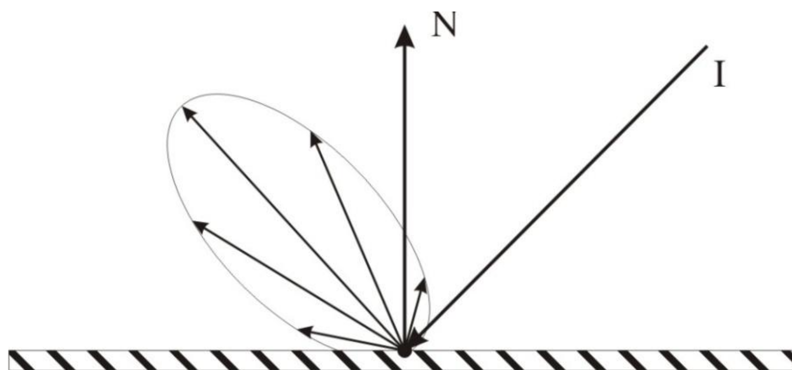
$$\text{dif. k.} = N \cdot I$$

2.3.4 Spekulární odraz

Spekulární odraz je používán ke znázornění lesklého povrchu, který ovšem neodráží své okolí. Na rozdíl od difúzního odrazu zde výsledná barva již závisí na úhlu, ze kterého povrch pozorujeme. K modifikaci výsledné barvy opět použijeme koeficient získaný takto:

$$spek. koef. = (R \cdot L)^{lsk}$$

R je odražený paprsek. Pokračování primárního p.
L je osvětlující paprsek
lsk reprezentuje lesk materiálu



Obrázek 2.8: Spekulární odraz

3 nVidia CUDA

V následující části je představena architektura CUDA a popsány její možnosti při provádění obecných výpočtů. Je zde popsán hardwarový a softwarový model a jsou uvedeny základní odlišnosti v programování pro CPU a CUDA

3.1 Popis

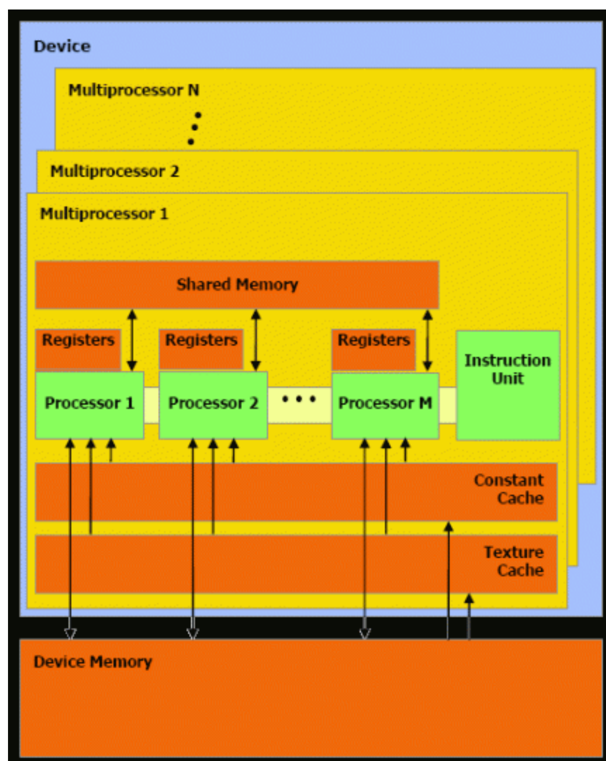
CUDA (Compute Unified Device Architecture) je architektura vyvinutá společností nVidia, uvedená na trh v roce 2006, umožňující provádět na jejích grafických kartách obecné výpočty (GPGPU). Díky paralelnímu zaměření grafických čipů poskytuje CUDA velké možnosti při zpracování paralelních instrukcí.

Využívat ji lze na kartách GeForce řady 8 a novějších, dále Quadro a Tesla. Pro psaní programů lze využít jazyky C (s rozšířením pro CUDA), OpenCL, DirectCompute a další. Z operačních systémů jsou podporovány Windows, Linux a MacOS.

S uváděním nových grafických karet na trh jsou spojeny i novější verze CUDA. Jejich detailní popis je uveden v [2].

3.1.1 Hardwarový model

Hardwarový model ilustruje následující obrázek:



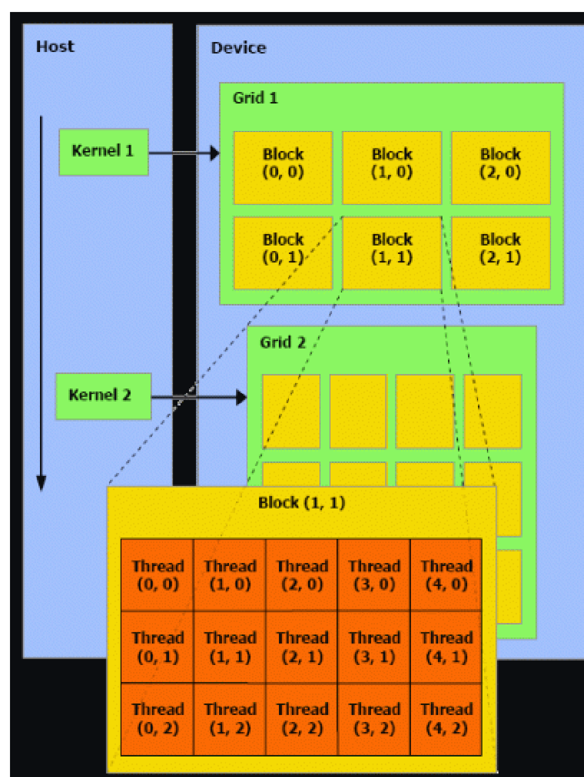
Obrázek 3.1: CUDA – hardwarový model (převzato z [2])

Grafický čip obsahuje několik multiprocesorů (např. 30 u GeForce GTX 280) a každý multiprocesor je složen z 8 procesorů. U starších karet je k dispozici 8192 registrů na multiprocesor, u novějších již dvojnásobek. Dále je zde vidět několik paměťových prostorů:

- Sdílená paměť – má kapacitu 16 KB na multiprocesor, jedná se o velmi rychlou paměť, do které mají přístup všechna vlákna v bloku (kapitola 3.1.2)
- Paměť textur – kapacita 64 KB, pouze pro čtení, je výhodná v případě, že načítaná data mají 2D strukturu
- Paměť konstant – kapacita 64 KB, je pouze pro čtení
- Globální paměť – největší a nejpomalejší paměť, velikost závisí na modelu karty (až 1,5 GB), slouží k přenosu dat mezi CPU a GPU, teoretická propustnost u GF GTX 280 je 140 GB/s (údaje převzaty z [3]).

3.1.2 Softwarový model

Jak již bylo řečeno hlavní výhodou architektury CUDA je paralelní zpracování instrukcí. Části programu, jejichž výpočet má probíhat na GPU jsou definovány ve vlastních funkcích zvaných jádra (kernels). Tyto funkce jsou při zavolání spuštěny několikrát paralelně ve vláknech (threads). Vlákna jsou rozdělena do bloků (blocks) a ty jsou umístěny do mřížky (grid). Situaci znázorňuje obrázek:



Obrázek 3.2: CUDA – softwarový model (převzato z [2])

Toto rozdělení umožňuje rozprostření výpočtu na libovolné množství multiprocesorů.
Po přidělení bloku vláken multiprocesoru ke zpracování je blok rozdělen na tzv. warpy. Warp je skupina vláken (na současných kartách 32) prováděných paralelně. Warp provádí najednou pouze

jednu instrukci, takže je důležitě, aby jednotlivá vlákna měla stejný průběh výpočtu. V případě, že se průběh liší je vypočet prováděn sériově. Detailní popis problematiky je uveden v [2].

Pro grafickou kartu je používán termín „zařízení“ (device) a pro zbytek systému termín „hostitel“ (host).

3.2 Odlišnosti od výpočtů na CPU

Při programování aplikace pro GPU je nutné mít na paměti odlišnost architektury a z toho vyplývající rozdíly, případně omezení. Kromě již zmíněného paralelismu jsou další hlavní rozdíly:

- Podpora výpočtů pouze s jednoduchou (32-bit) přesností (do verze CUDA 1.3)
- CUDA nepodporuje rekurzivní funkce

Definování funkce (v jazyce C) určené k běhu na grafické kartě (kernelu) se provádí pomocí klíčového slova „`__global__`“, které následuje již běžný zápis funkce. Toto klíčové slovo značí, že se jedná o funkci prováděnou na GPU. Například:

```
__global__ void primRays(Screen scr, Hit *stack, Camera cam){tělo funkce}
```

Návratová hodnota všech funkcí s prefixem `__global__` musí být `void`. Takovouto funkci je možné zavolat pouze z hostitele a volání takovéto funkce vypadá následovně:

```
primRays<<<10, 20>>>(screen, stack, camera);
```

Výraz v ostrých závorkách se nazývá execution configuration a udává, kolikrát bude funkce spuštěna. První údaj je počet bloků v mřížce, druhý počet vláken v bloku. Pro výše uvedený příklad tak dostáváme 200 vláken provádějících naši funkci. Počty bloků i vláken mohou být zadány 3-složkovým vektorem. Můžeme tak pracovat až s trojdimenzionálním polem bloků/vláken. Nejsou-li zadány všechny 3 složky vektoru, jako v našem případě, jsou nezadané hodnoty nastaveny na 1.
K identifikaci jednotlivých vláken a bloků slouží jejich identifikační čísla, dostupná přes předdefinované proměnné, což jsou opět 3-složkové vektory.

Dále je možné definovat funkci pomocí klíčového slova „`__device__`“. Tato funkce již může mít návratovou hodnotu jinou než `void`, ale je možné ji volat pouze ze zařízení.

K dosažení co největšího přínosu při výpočtech na grafické kartě je především nutné zaměřit se na možnost paralelizace implementovaného algoritmu. Dále je také dobré dodržovat tato doporučení:

- Omezit přenos dat mezi hostitelem a grafickou kartou
- Zajistit optimalizovaný přístup do globální paměti
- Vyhýbat se větvení kódu v rámci warpu
- Minimalizovat práci s globální pamětí

4 Návrh algoritmu

V této části jsou popsány úpravy v klasickém algoritmu pro ray tracing, aby byla možná jeho implementace pomocí knihoven CUDA a dalo se očekávat dosažení lepších výsledků oproti CPU verzi. Dále jsou zde popsány další volby učiněné při navrhování programu.

4.1 Možnosti programu

Protože hlavním cílem této práce bylo převést algoritmus ray tracingu na GPU, byly zvoleny pouze dvě základní primitiva pro tvorbu scén – trojúhelník a koule. Pro demonstraci možností je to dostatečné a dodatečné objekty bude možné doplnit při případném dalším rozvoji aplikace.

Pro zobrazení výsledku renderování byly zvoleny knihovny GLUT pro OpenGL, což by mělo zajistit přenositelnost mezi platformami. Výsledek tedy bude zobrazen pouze v okně aplikace o zadaném rozlišení. Také nebylo nutné vytvářet animace, proto byla zvolena pouze statická scéna a renderovnání proběhne při každém spuštění programu pouze jednou.

Pro uložení informací o scéně byly zvoleny jednoduché textové soubory, jejichž název je předán programu jako jediný parametr při spuštění.

Vzhledem k paralelnímu provádění výpočtů bude vhodné zvolit jako podmínku ukončení algoritmu limit pro počet odrazů/lomů primárního paprsku ve scéně.

4.1.1 Formát vstupních dat

Pro rozlišení jednotlivých objektů při načítání byl zvolen jednoduchý systém párových značek, ve kterých jsou uzavřeny jejich vlastnosti. Popis začíná značkou určující typ objektu (např. „`<Sphere>`“ pro kouli) a končí značkou „`<end>`“. Soubor má také pevně danou hlavičku, aby ho bylo možné rozpoznat a dále slouží pro určení velikosti potřebné paměti.

Následuje popis scény: umístění kamery a světel, popis materiálů a velikosti renderovaného obrázku a samotné informace o pozici a vlastnostech objektů na scéně. Pořadí jednotlivých prvků může být libovolné. Následuje-li po sobě několik objektů stejného typu, mohou být odděleny značkou „`<next>`“. Při zadávání číselných vlastností objektů musí být dodržovány mezery a oddělovače jak je uvedeno v příkladu:

```
scene file version 1.0
Number of lights: 1
Number of objects: 3
Number of materials: 2

<Camera>
  position: 0, 0, 1000
  direction: 0, 0, -1
  up direction: 0, 1, 0
  right direction: 1, 0, 0
  zoom: 3400
<end>

<Triangle>
  vertices: 30, -100, -150; 100, -100, -100; 30, 0, -150
  color: 0.12, 0.57, 0.87
  material: 2
<next>
  vertices: 30, 0, -150; 100, -100, -100; 100, 0, -100
  color: 0.24, 0.4, 0.5
  material: 2
<end>
```

Hlavička tvoří první 4 řádky souboru a udává počet objektů scény, pro které bude alokována paměť. Následuje popis scény. Všechny značky jsou uvedeny a popsány v příloze B. Definovány jsou v souboru `read_file.cpp`.

4.2 Paralelizace

Paralelizace je prvním krokem při transformaci algoritmu pro provádění pomocí knihoven CUDA. Cílem je dosáhnout pokud možno co největšího stupně paralelizmu, protože tak je možné naplno využít možností grafické karty. Z tohoto pohledu je ray tracing velmi vhodným kandidátem, protože barva každého paprsku může být určena zcela nezávisle – to znamená pro několik paprsků najednou.

To je zřejmé i z algoritmu uvedeného v kapitole 2.1.1. Všechny v něm uvedené operace se provádí opakovaně pro každý pixel obrazu. Zde se paralelní zpracování přímo nabízí. Provádění výpočtů pro více pixelů najednou by mělo poskytnout dostatečný stupeň paralelizmu k plnému využití potenciálu karty.

To znamená, že pro každý primární paprsek bude vytvořeno jedno vlákno, které bude provádět výpočty. Všechna vlákna budou sdílet paměť, ve které bude uložena reprezentace scény a z které budou pouze číst požadovaná data.

4.3 Odstranění rekurze

Poněkud složitějším problémem je odstranění rekurze z klasického algoritmu (kapitola 2.1.1). Běžným postupem při odstraňování rekurze je použití zásobníku. Při sériovém zpracování by stačil jeden zásobník, který by se používal postupně při výpočtu barvy každého pixelu (primárního

paprsku). Zde vzniká první problém, neboť nyní je počítáno více pixelů najednou, takže je třeba mít zásobník pro každý zvlášť.

Paměť pro tyto zásobníky je na grafické kartě také nutné alokovat před začátkem výpočtu. To znamená určit dopředu velikost zásobníku. Protože pro omezení výpočtů je zadána hodnota maximálního počtu odrazů/lomů paprsku, je možno velikost zásobníku určit v předstihu. Nevýhoda je ta, že takto je alokována paměť i pro paprsky u kterých nedojde k maximálnímu počtu odrazů. Jistá výhoda je naopak v tom, že se sníží větvení kódu v rámci warpu, neboť je možné provádět stejné operace nad všemi pixely bez ohledu na to, zda paprsek nějaký objekt protnul či ne. Nerekurzivní postup je následující:

1. Vygenerovat primární paprsek
2. Určit průsečík s objekty na scéně
3. Určit přímé osvětlení v bodě průsečíku a uložit na zásobník
4. Vygenerovat odražený a lomený paprsek
5. Opakovat od bodu 2 dokud není dosažena maximální hloubka
6. Od maximální hloubky postupně kombinovat barvy odražených a lomených paprsků s přímým osvětlením až je získána výsledná barva primárního paprsku

Práce s pamětí je detailně rozebrána v kapitole 5.5.

5 Implementace

Následující kapitola popisuje implementaci programu. Přibližuje rozdělení zdrojových kódů a možnosti při překladu. Následuje detailní rozebrání průběhu programu a přiblížení úprav algoritmu provedených při implementaci.

5.1 Struktura kódu

Pro překlad zdrojových kódů slouží překladač `nvcc` od společnosti nVidia. Ten překládá části kódu určené pro běh na GPU a pro ostatní volá běžný překladač jazyka C. Pro soubory s částmi kódu určených pro GPU je vyhrazena přípona `.cu`.

Zdrojové kódy programu tvoří 10 souborů:

- `definitions.h` – definuje základní struktury: vektor, barvu a bod
- `glut_wndw.gpp, glut_wndw.h` – funkce mající na starost vykreslení okna aplikace
- `main.cpp` – hlavní funkce programu, inicializuje paměť hostitele
- `math_func.cu` – funkce pro počítání s vektory a maticemi
- `read_file.cpp, read_file.h` – načte soubor se scénou do paměti hostitele, definuje strukturu souboru
- `structures.h` – definuje struktury reprezentující renderovanou scénu a zásobníky
- `trace_func.cu, trace_func.h` – hlavní část programu obsahuje kernely a pomocné funkce

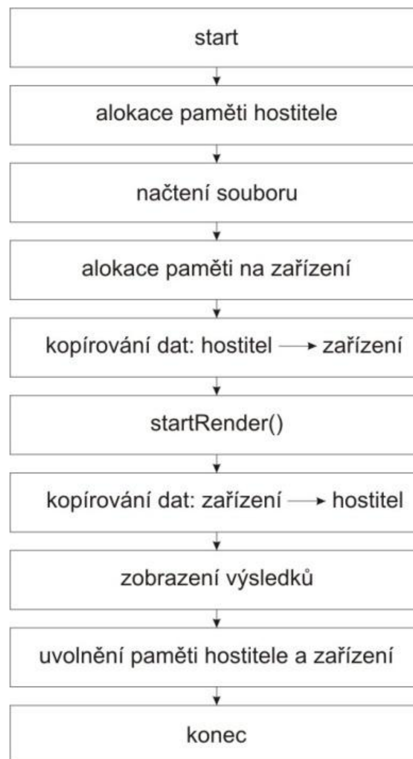
5.2 Překlad kódu

Program byl vyvíjen a testován v linuxovém prostředí pro všechna zařízení schopná compute capability 1.0 a vyšší. Detailní specifikace různých verzí compute capability jsou uvedeny v [2].

Součástí zdrojových kódů je makefile, který umožňuje kromě standardního překladu také vytvoření simulace (volba `-info`), takže veškeré výpočty budou prováděny na CPU. To je výhodné hlavně při ladění programu, protože běžně není možné volat z kernelu funkce, které nejsou definovány jako „`__device__`“. Další možností při překladu je vypsání informací o využití registrů a paměti GPU jednotlivými kernely (volba `-info`).

5.3 Průběh programu

Obrázek 5.1 představuje jednoduchý diagram průběhu programu. Základním prvkem je funkce startRender() která prochází stromem možné dráhy paprsku (kapitola 5.4) a volá GPU funkce počítající osvětlení.



Obrázek 5.1: Průběh programu

Také je zde vidět standardní postup při používání CUDA zařízení. Zdrojová data jsou vytvořena na hostiteli a zkopírována do paměti zařízení, kde jsou nad nimi provedeny požadované výpočty a výsledek je zkopírován zpět do paměti hostitele.

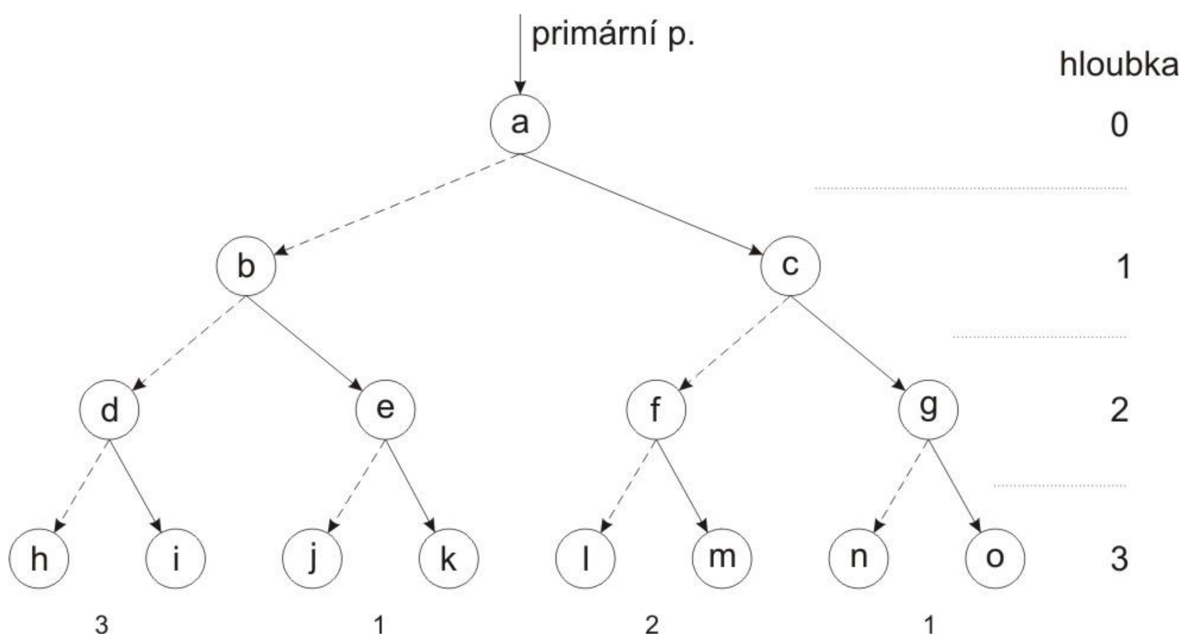
Již v prvních verzích implementace vyšla najevo nutnost použití méně komplexních kernelů, které využívají méně zdrojů zařízení (především registrů). Pokud jsou nároky kernelu na zdroje GPU vysoké, je k úspěšnému spuštění nutné zmenšit velikost bloku, což by mohlo vést ke snížení výkonu. Proto je výpočet rozdělen do několika méně náročných kernelů. Rozdělení téměř odpovídá postupu uvedenému v kapitole 4.3:

- `primRays()` – generuje primární paprsky v závislosti na poloze kamery
- `findIntersection()` – hledá průsečík paprsku s nejbližším objektem a určí normálu v nalezeném bodě
- `directLight()` – určí barvu v bodě průsečíku pouze z přímých zdrojů (spekulární a difúzní barevnou složku, bez odrazu/lomu)
- `continueRay()` – vytvoří odražený a lomeny paprsek
- `indirectLight()` – k již vypočítanému přímému osvětlení přidá barvy odražených a lomených paprsků
- `finalColor()` – konečné barvy primárních paprsků zkopíruje ze zásobníků do frame bufferu určenému k zobrazení

5.4 Dráha paprsku

Na obrázku 5.2 je znázorněno možné pokračování primárního paprsku s hloubkou sledování odrazů/lomu nastavenou na 3, za předpokladu, že pro každý segment paprsku pokaždé dojde k protnutí s objektem a následnému lomu i odrazu zároveň. Pro minimalizaci větvení kódu v rámci kernelu, je pro každý primární paprsek dopředu počítáno s tímto kompletním větvením paprsku.

Již před zahájením výpočtů musí být alokována paměť, do které bude možné uložit informace o každém potencionálním lomu/odrazu primárního paprsku (na obrázku uzly stromu). Pro každý je nutné uložit informace o příchozím paprsku, bod průsečíku, odkaz na protnutý objekt, normálu v bodě, barvu a index lomu materiálu.



Obrázek 5.2: Dráha paprsku (3 odrazy): plné šipky = odražený paprsek, čárované = lomený, čísla u listů udávají, kolik výsledných barev bude možné určit (kapitola 5.5)

Barva primárního paprsku se skládá ze tří složek. Z přímého osvětlení a barev odraženého a lomeného paprsku. Totéž platí pro všechna pokračování primárního paprsku. Výjimkou jsou paprsky v nejvyšší hloubce, jejich barva je určena pouze z přímého osvětlení.

5.5 Využití paměti

Pokud by mělo být alokováno místo pro všechny uzly stromu z obrázku 5.2, byly by paměťové nároky aplikace s rostoucí hloubkou sledování paprsku velmi brzy neúnosné: Velikost struktury pro uložení informací o průsečíku (kapitola 5.4) pro každý uzel je 56 bytů, počet uzlů je $(2^{hloubka+1} - 1)$. Pro rozlišení 800x600 pixelů a hloubku 3 dostáváme téměř 400MB paměti pro zásobníky. A velikost se zdvojnásobí s každým zvýšení hloubky sledování paprsku.

Ovšem jedinou důležitou informací pro výsledek je barva jednotlivých úseků paprsku (a index průhlednosti/odrazivosti materiálu). Ostatní informace (příchozí paprsek, bod průsečíku, odkaz na

protnutý objekt, normála v bodě, barva materiálu) slouží pouze k výpočtu pokračování paprsku a barvy přímého osvětlení. Proto je vytvořen druhý zásobník, kde je ukládána pouze barva a potřebný index. V původním zásobníku se takto mohou, po určení přímého osvětlení a pokračování paprsku, data přepisovat. Zvolením vhodného způsobu procházení (metodou „preorder“) stromem z obrázku 5.2 bylo dosaženo dalšího zmenšení paměťových nároků.

Nároky na paměť ve výsledné implementaci jsou tyto: Zásobník pro informace o průsečíku musí pojmout pouze [*hloubka_procházení+1*] uzlů (1 uzel = 56 bytů) a zásobník pro barvu paprsků [2**hloubka_procházení+1*] barev (16 bytů). Pro příklad uvedený výše je to přibližně 153MB paměti a s rostoucí hloubkou se tato hodnota zvyšuje o konstantu (v tomto případě přibližně 40MB).

Tabulka 5.1: Využití paměti při průchodu stromem

krok	zásobník průsečíků					zásobník barev						
1	a				→	a						
2	b	c			→	a	b	c				
3	b	f	g		→	a	b	c	f	g		
4	b	f	n	o	→	a	b	c	f	g	*n*	*o*
5	b	f				a	b	c	f	G		
6	b		l	m	→	a	b	c	f	G	*l*	*m*
7	b					a	b	c	*F*	*G*		
8	b					a	b	C				
9		d	e		→	a	b	C	d	e		
10		d	j	k	→	a	b	C	d	e	*j*	*k*
11		d				a	b	C	d	E		
12			h	i	→	a	b	C	d	E	*h*	*i*
13						a	b	C	*D*	*E*		
14						a	*B*	*C*				
15						A						

Tabulka 5.1 demonstruje postup pocházení stromem z obrázku 5.2. Písmena v zásobníku průsečíků udávají aktuálně uložené uzly, v zásobníku barev pak malá písmena znamenají zjištěnou barvu z přímého osvětlení a velkými písmeny je značena výsledná barva paprsku.

Po přidání uzlu do zásobníku je spočítána barva z přímého osvětlení (v tabulce označeno šipkou). Následuje vygenerování uzlů pro odražený a lomený paprsek (ten nahradí rodičovský uzel). Následuje opět výpočet barvy pro nové uzly a generování následníků pro uzel reprezentující odražený paprsek (nejdříve se prochází pravý podstrom). V kroku 4 je zjištěna barva paprsků v největší hloubce a může být určena výsledná barva paprsku pro uzel [g] (krok 5). V dalším kroku je určeno osvětlení v uzlech [l] a [m] takže je možné určit výslednou barvu uzlu [f] a následně již i uzlu [c]. Kolik výsledných barev takto může být určeno, udávají na obrázku čísla uvedená pod listy.

6 Testování

Následující část přibližuje způsob testování programu a uvádí dosažené výsledky. Testována je časová náročnost programu a také je provedeno srovnání s CPU implementací ray traceru.

6.1 Testovací sestava

Vývoj a testování probíhaly na notebooku Acer Aspire 5920G. Detailní specifikace ukazuje tabulka 6.1. Byla testována rychlost programu v závislosti na složitosti scény a také proběhlo porovnání s CPU implementací ray tracingu.

Tabulka 6.1: Testovací sestava

Procesor:	Intel Core 2 Duo T7300 – 2.0GHz
Grafická karta:	nVidia GeForce 8600M GT 512MB jádro: 475MHz paměti: 400MHz
Operační paměť:	2GB DDR2
Operační systém:	Ubuntu 9.10 Karmic koala – 2.6.31-20
Ovladač grafické karty:	195.36.17

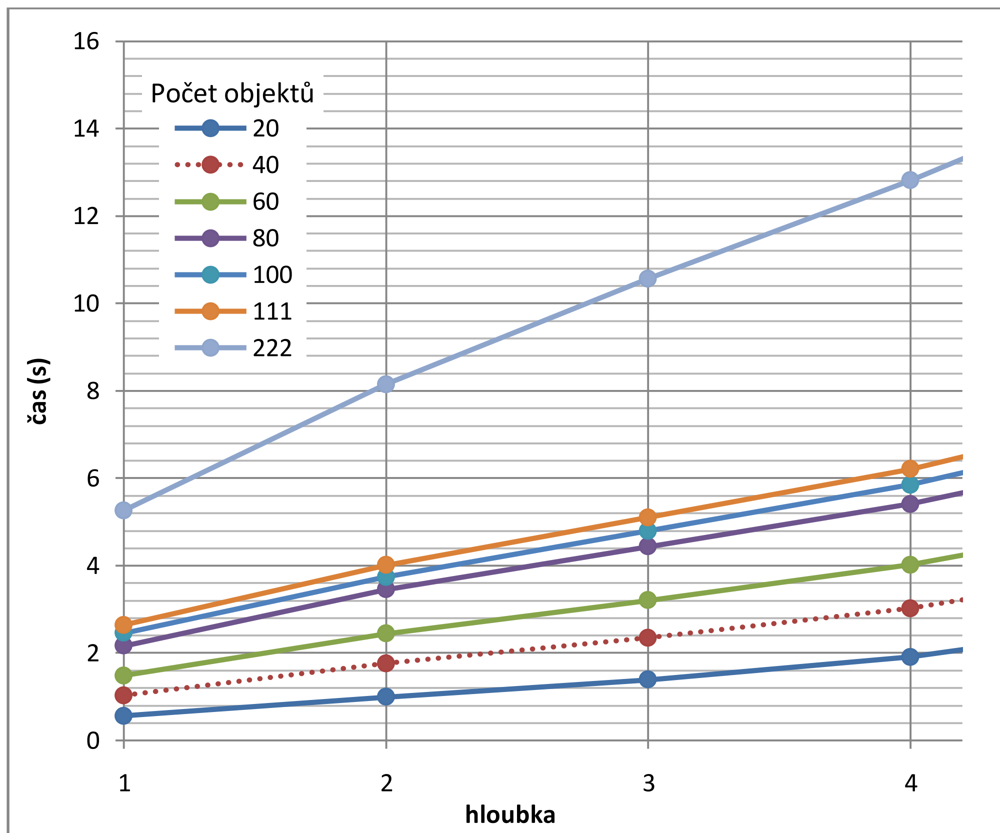
Jedná se dnes již o poměrně zastaralou sestavu, které je výkonově několikanásobně překonána, ale i přesto jsou zde patrné značné výhody GPU implementace řešeného problému.

6.2 Způsob měření

Knihovny CUDA poskytují funkce pro měření času výpočtu prováděného na GPU. Způsob měření vychází z postupu uvedeného v [3]. Použité funkce jsou `cudaEventCreate()` pro vytvoření událostí, `cudaEventRecord()` pro zaznamenání času událostí a `cudaEventElapsedTime()` pro zjištění časového rozdílu mezi dvěma událostmi. Začátek měření je před spuštěním prvního kernelu a končí po zkopírování výsledného frame bufferu ze zařízení do paměti hostitele. Zjištěné časové údaje poskytují přesnost v řádu milisekund. Pro zjištění orientačního času běhu celého programu bylo také použito funkce `time()` ze standardní knihovny jazyka C.

6.3 Složitost scény

Obrázek 6.1 ukazuje závislost doby výpočtu pro jednotlivé hloubky sledování paprsku na množství objektů (pouze grafických primitiv – trojúhelník a koule, nikoli světel) v zobrazované scéně. Výsledný obraz byl renderován v rozlišení 700x700 pixelů s jedním zdrojem světla (příloha A – `sceneTime.txt`).



Obrázek 6.1: Závislost doby výpočtu na počtu objektů a hloubce sledování paprsku (GPU verze)

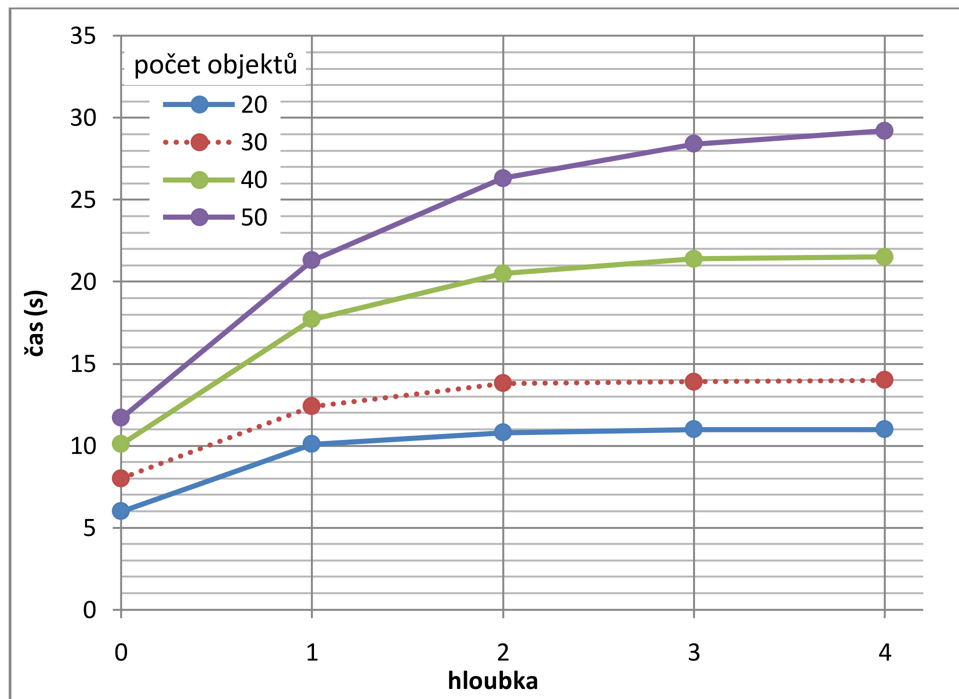
Zdrojová data jsou uvedena v následující tabulce:

Tabulka 6.2: Časy renderování v sekundách (700x700pix, 1 světlo)

objektů:	20	40	60	80	100	111	222
hloubka 0:	0,566	1,032	1,489	2,163	2,461	2,644	5,260
hloubka 1:	0,992	1,762	2,446	3,453	3,742	4,011	8,148
hloubka 2:	1,393	2,347	3,207	4,434	4,797	5,100	10,567
hloubka 3:	1,911	3,025	4,016	5,412	5,852	6,206	12,809
hloubka 4:	2,749	3,992	5,119	6,689	7,233	7,646	15,305

Z grafu vyplývá, že časová náročnost se pro tutéž scénu s rostoucí hloubkou sledování paprsku zvyšuje lineárně. Nezáleží ani na vlastnostech jednotlivých objektů (ne/průhlednost, zrcadlení), protože je vždy procházen kompletní strom dráhy paprsku.

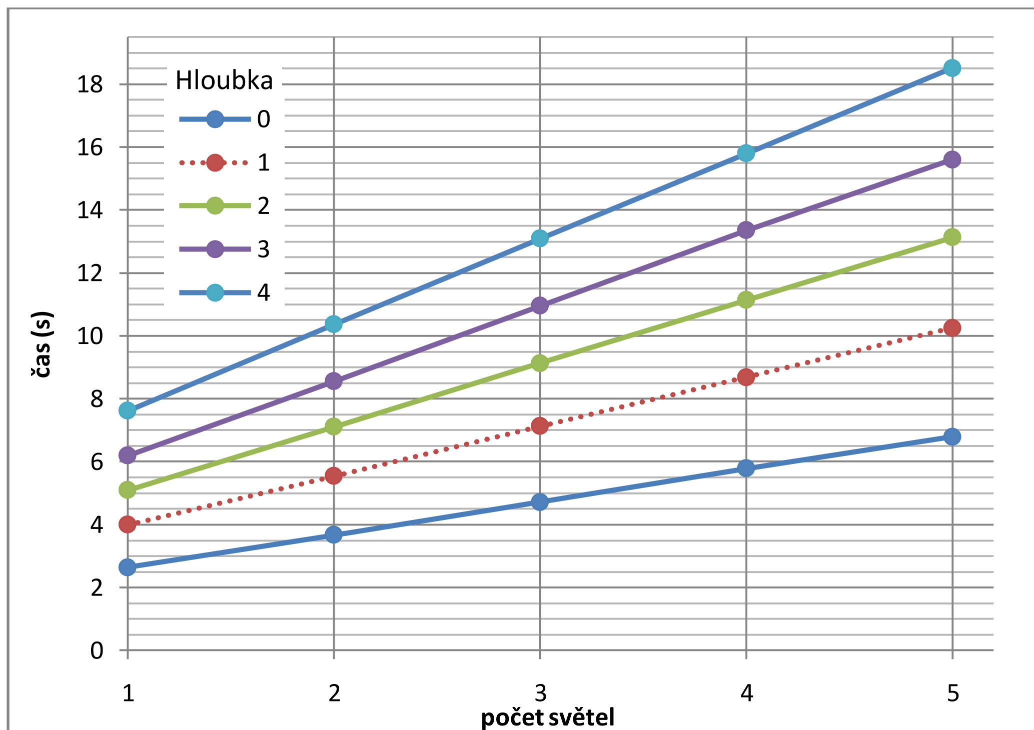
Obrázek 6.2 představuje odpovídající situaci počítanou pomocí CPU. Pro srovnání byl použit ray tracer vytvořený v rámci semestrálního projektu, napsaný v jazyce C++. Pracuje pouze s jedním grafickým primitivem (koulí) a používá zjednodušený výpočet lomeného paprsku.



Obrázek 6.2: Závislost doby výpočtu na počtu objektů a hloubce sledování paprsku (CPU verze) (700x700pix, 3 světla)

V tomto případě, s narůstající hloubkou sledování paprsku se časový přírůstek snižuje. Zde ovšem velmi záleží na vlastnostech objektů. Výpočet barvy paprsku zde končí v případě, že nedojde k žádnému průsečíku (nebo objekt neodráží/neláme paprsek), což v případě jednoduchých scén (včetně testované) vede k rychlému určení výsledných barev po několika krocích (pro objekt, který světlo neodráží ani neláme již v hloubce 0). Pro složitější scény by se časová náročnost zvyšovala znatelněji.

Obrázek 6.3 znázorňuje časovou náročnost v závislosti na počtu světel umístěných v zobrazované scéně. Testovací scéna byla použita stejná jako u testu znázorněného na obrázku 6.1 (příloha A, `sceneTime.txt` - verze obsahující 111 objektů). Při určování přímého osvětlení se stínové paprsky testují na průsečík se všemi objekty scény, zvlášť pro každé světlo. Proto má přidání dalšího zdroje světla výrazný dopad na dobu výpočtu.



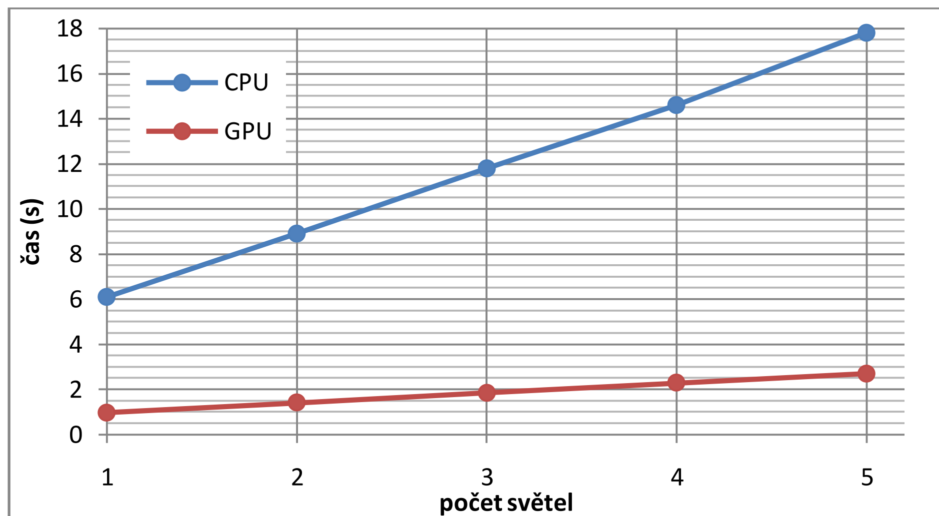
Obrázek 6.3: Závislost doby výpočtu na počtu světel a hloubce sledování paprsku (GPU verze)

6.4 Porovnání s CPU verzí

Jak již bylo zmíněno výše, pro porovnání byl použit velmi jednoduchý ray tracer, který podporuje pouze jeden objekt (kouli). Stejně jako GPU verze také implementuje pouze základní verzi algoritmu bez optimalizací, s tím rozdílem, že používá jednodušší výpočet pro určení lomeného paprsku.

Vzhledem k omezení CPU verze je testovací scéna složena pouze z koulí. Testy byly provedeny pro různé počty objektů, světel a hloubku sledování paprsku, v rozlišení výstupu 700x700 pixelů. (příloha A, `sceneTest.txt`)

Obrázek 6.4 znázorňuje první test – dobu výpočtu pro scénu s 50 objekty a měnícím se počtem světel.



Obrázek 6.4: Závislost času výpočtu na počtu zdrojů světla

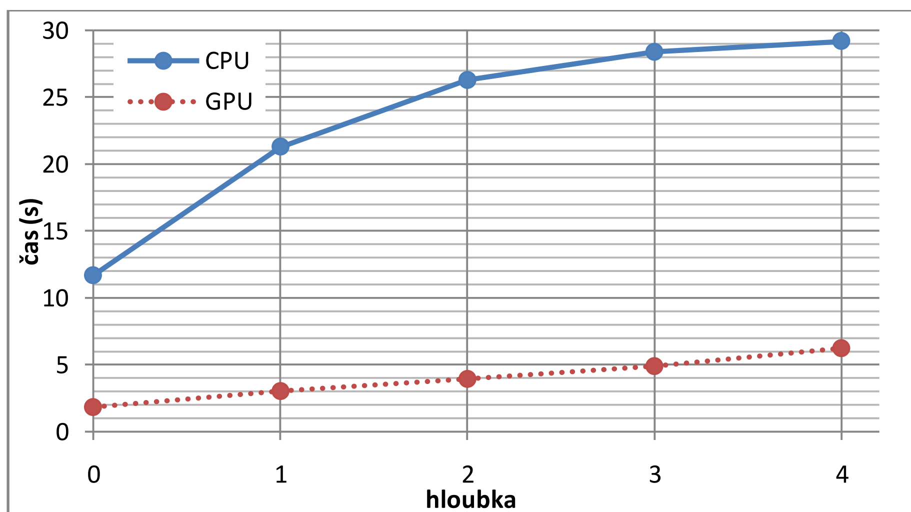
GPU implementace zde dosahuje více, jak šestinásobného urychlení proti CPU verzi.

Následující tabulka porovnává časy výpočtu na GPU a CPU. Obrázek 6.5 znázorňuje poslední řádek tabulky, což je scéna s 50 objekty při různých hloubkách sledování paprsku.

Tabulka 6.3: Porovnání výpočtu na GPU a CPU

hloubka	0		1		2		3		4	
počet objektů	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
20	6,00 s	0,95 s	10,10 s	1,49 s	10,80 s	1,82 s	11,00 s	2,23 s	11,00 s	2,93 s
30	8,00 s	1,27 s	12,40 s	1,92 s	13,80 s	2,32 s	13,90 s	2,74 s	14,00 s	3,62 s
40	10,10 s	1,56 s	17,70 s	2,53 s	20,50 s	3,10 s	21,40 s	3,67 s	21,50 s	4,50 s
50	11,70 s	1,84 s	21,30 s	3,03 s	26,30 s	3,95 s	28,40 s	4,92 s	29,20 s	6,25 s

Průměrné urychlení proti CPU implementaci je opět téměř šestinásobné. Ovšem výkon CPU verze je závislý na vlastnostech a uspořádání objektů ve scéně. Zatímco časová náročnost GPU algoritmu roste s přibývající hloubkou lineárně.



Obrázek 6.5: Porovnání doby výpočtu pro scénu s 50 objekty a třemi světly

7 Závěr

Poslední kapitola obsahuje zhodnocení dosažených výsledků a diskutuje možnosti pro rozšiřování programu.

7.1 Zhodnocení výsledků

Výsledkem této práce je transformace klasického rekurzivního algoritmu ray tracingu pro použití na grafických kartách společnosti nVidia. Což obnášelo odstranění rekurze z algoritmu a jeho co možná největší paralelizaci. Na GPU se podařilo přenést veškeré výpočty týkající se ray tracingu a tím omezit přenos dat mezi hostitelem a zařízením, což je úzké hrdlo při provádění výpočtů na architektuře CUDA.

Architektura CUDA je vhodná především na provádění stejných výpočtů nad velkým objemem dat. Proto byly nutné úpravy, aby docházelo k co nejmenšímu větvení kódu při běhu programu na GPU. Některé výpočty jsou tak prováděny navíc nad „prázdnými“ daty ale díky masivní paralelizaci je takovéto řešení výhodnější.

Testování proběhlo na již poměrně málo výkonném hardwaru a GPU implementace se ukázala v průměru asi 6krát rychlejší. S novými grafickými akcelerátory by měl být výkon několikanásobně vyšší. V nejbližší době by měly grafické akcelerátory dosáhnou takového výkonu, aby bylo možné uvažovat o používání ray tracingu pro renderování v reálném čase.

Vzhledem k poměrné omezeným možnostem programu, co se týká zobrazení grafických primitiv, je vytvořený program vhodný především pro demonstraci možností grafických akcelerátorů při provádění obecných výpočtů.

7.2 Možnosti dalšího vývoje

Algoritmus pro ray tracing není ve své základní podobě příliš efektivní. Existuje mnoho optimalizací, které, pokud by byly implementovány, by měli podstatný vliv na výkon aplikace. Za velmi vhodnou optimalizaci považuje autor např. obálky (bounding volumes), které by mohli velmi urychlit výpočet (nejen) přímého osvětlení.

Dalším krokem v případné další optimalizaci programu by mělo být vhodné využití různých paměťových prostorů zařízení. Globální paměť, použitá pro uložení všech dat na grafické kartě, je ve srovnání s dalšími možnostmi pomalá. Vhodné by bylo ukládat reprezentaci scény (k těmto datům se přistupuje při výpočtech nejčastěji) např. do sdílené paměti, která má mnohem nižší prodlevy. Dále by zajisté také bylo možné více optimalizovat velikost potřebné paměti.

Pro reálné využití programu bude také nutné rozšířit jeho zobrazovací schopnosti. Zde jsou možnosti velmi široké. Od přidání dalších primitiv, implementaci CGS, přes zobrazování textur, antialiasing až například k plošným zdrojům světla. Dalším velmi užitečným krokem by také byla možnost načítání souborů se scénou nějakého standardního 3D formátu.

Literatura

[1] Arvo, J.; Cook, R.; Glassner, A.; Haines, E.; Hanrahan, P.; Heckbert, P.; Kirk, D.:Glassner, A.: An introduction to ray tracing, editace A. Glasner, 2002, ISBN 0-12-286160-4

[2] nVidia Corporation: NVIDIA CUDA Programming Guide 2.3 [online; navštíveno 6. 4. 2010] URL http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[3] nVidia Corporation: NVIDIA CUDA Best Practices Guide 3.0 [online; navštíveno 10. 5. 2010] URL http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf

[4] Forrest Briggs: Ray tracing tutorial [online; navštíveno 10. 5. 2010] URL http://www.groovyvis.com/other/raytracing/index.html

[5] Internetové stránky knihovny pro realtime zobrazování OpenGL [online; navštíveno 21. 3. 2010] URL http://opengl.org

[6] Stránky GPGPU komunity [online; navštíveno 15. 3. 2010] URL http://gpgpu.org

[7] Wikipedia, The Free Encyclopedia: Ray tracing (graphic) [online; navštíveno 7. 2. 2010] URL http://en.wikipedia.org/wiki/Ray_tracing_%28graphics%29#Algorithm:_classical_recursive_ray_tracing

[8] nVidia Corporation: NVIDIA CUDA C SDK Code Samples [online; navštíveno 8. 5. 2010] URL http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html

[9] Lukáš Zaorálek: Úvod do technologie CUDA [online; navštíveno 10. 5. 2010] URL http://www.root.cz/serialy/uvod-do-technologie-cuda/

[10] Grégory Massal: A raytracer in C++ [online; navštíveno 19. 12. 2009] URL http://www.codermind.com/articles/Raytracer-in-C++-Part-I-First-rays.html

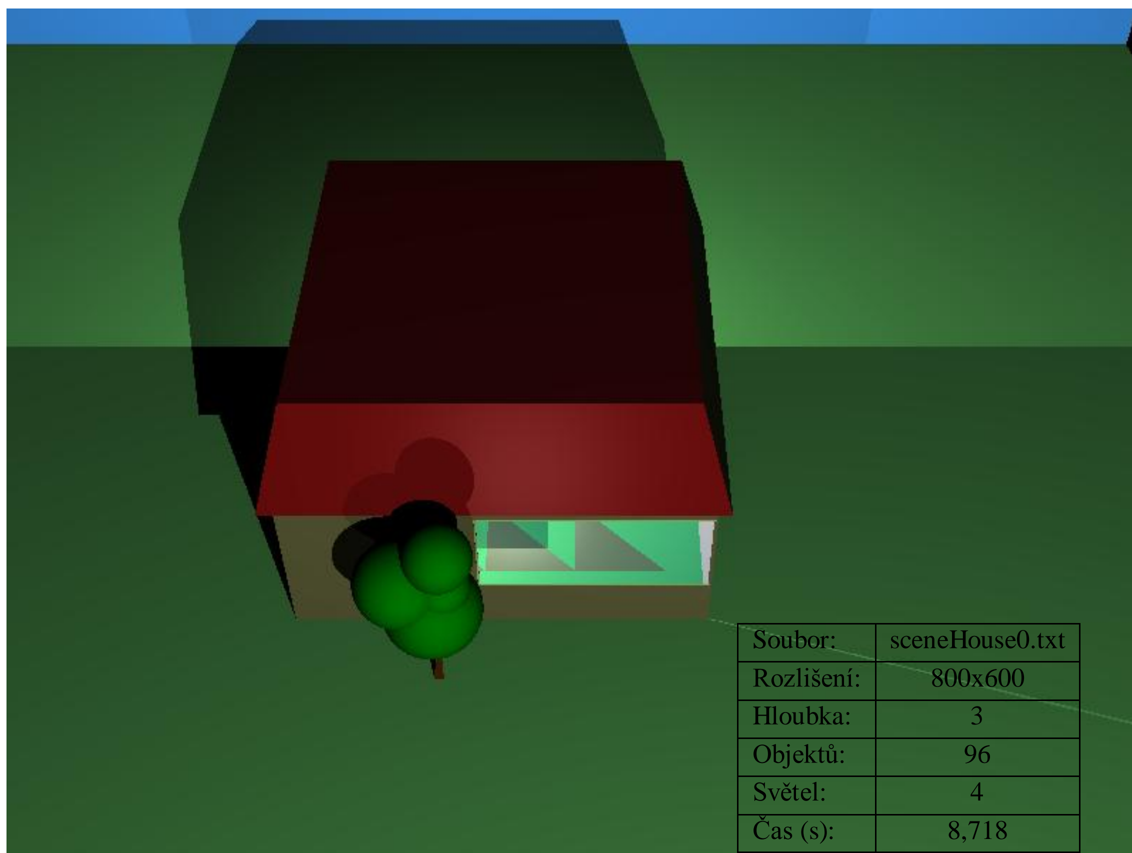
[11] Tom Hammersley: The Basics of Ray Tracing [online; navštíveno 1. 5. 2010] URL http://www.devmaster.net/articles/raytracing/

Seznam příloh

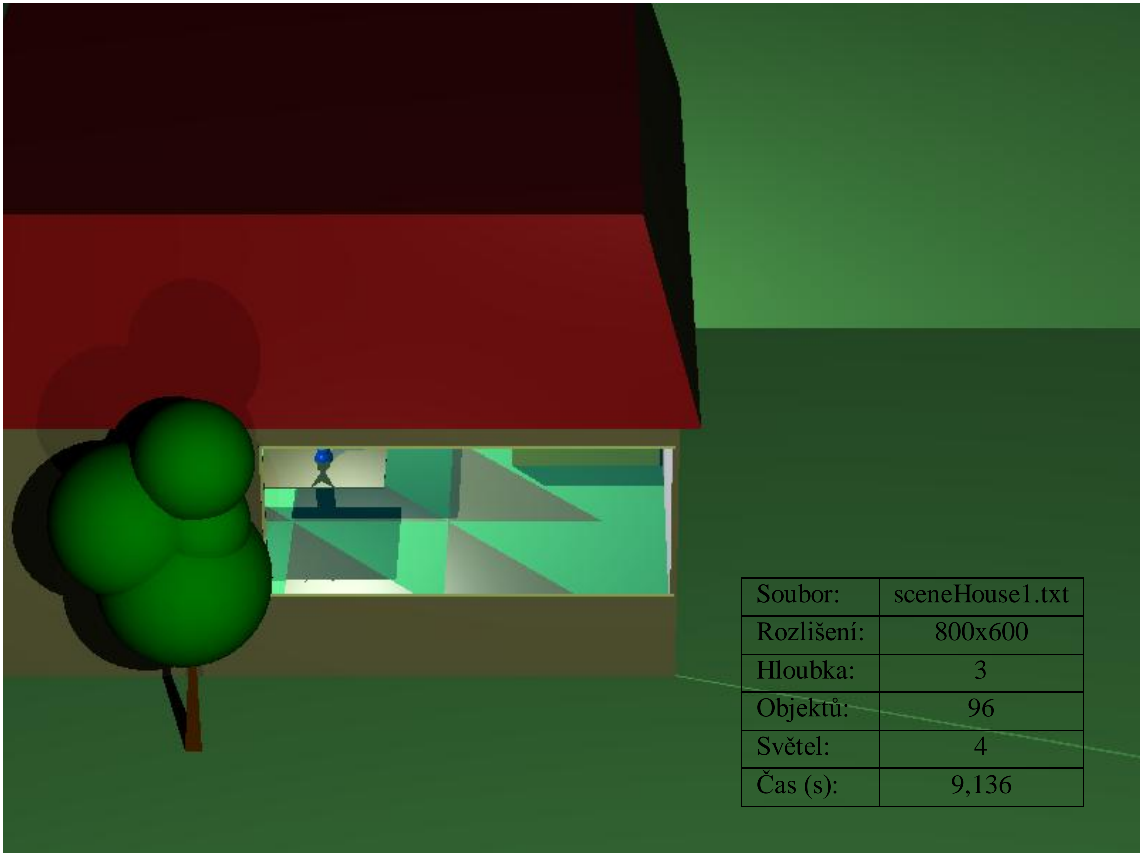
A. Vzorové výstupy programu
B. Popis souboru pro uložení scény
C. CD s implementací a zdrojovými texty práce

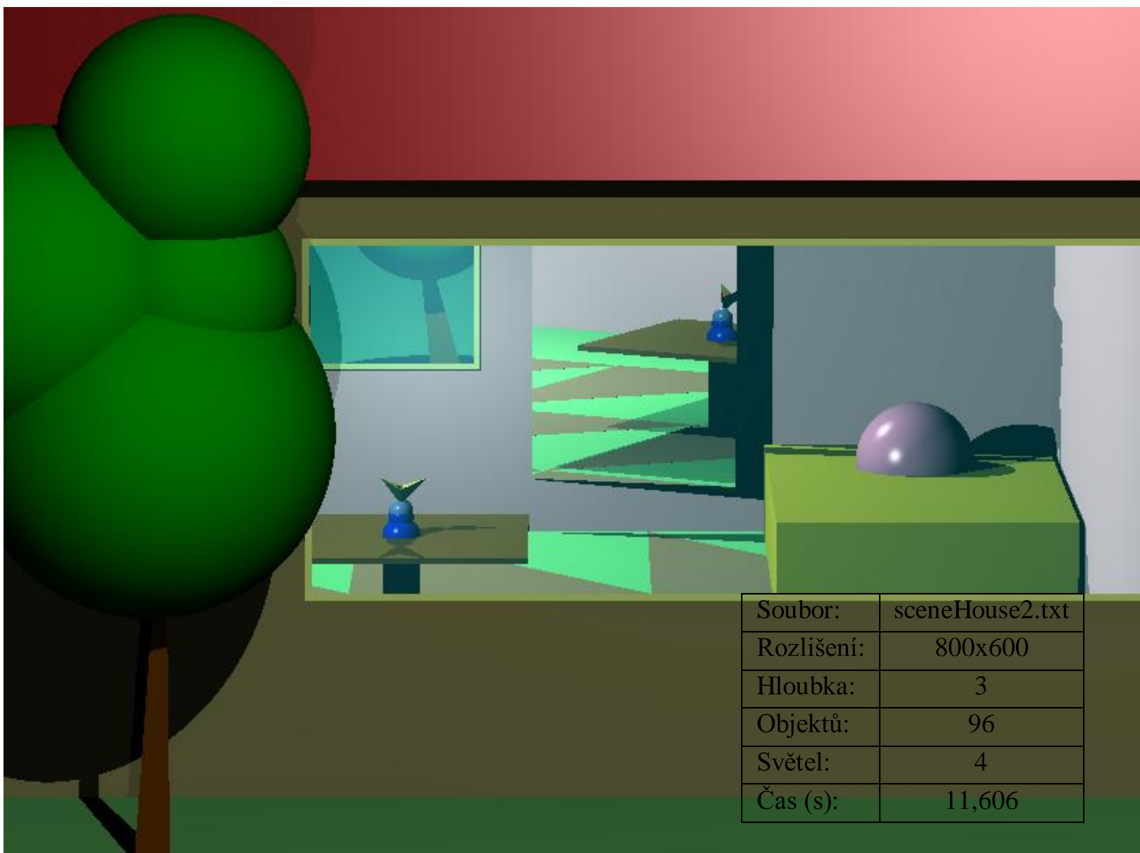
Příloha A

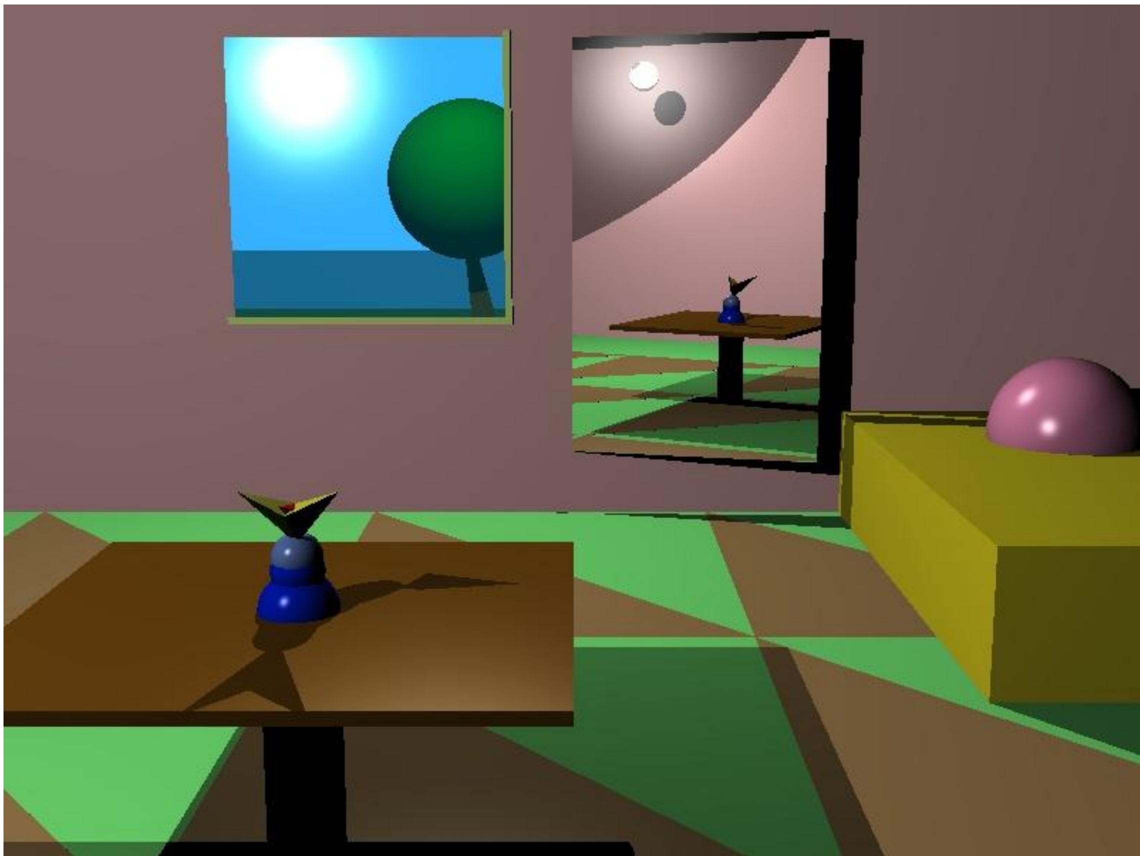
Vzorové výstupy programu



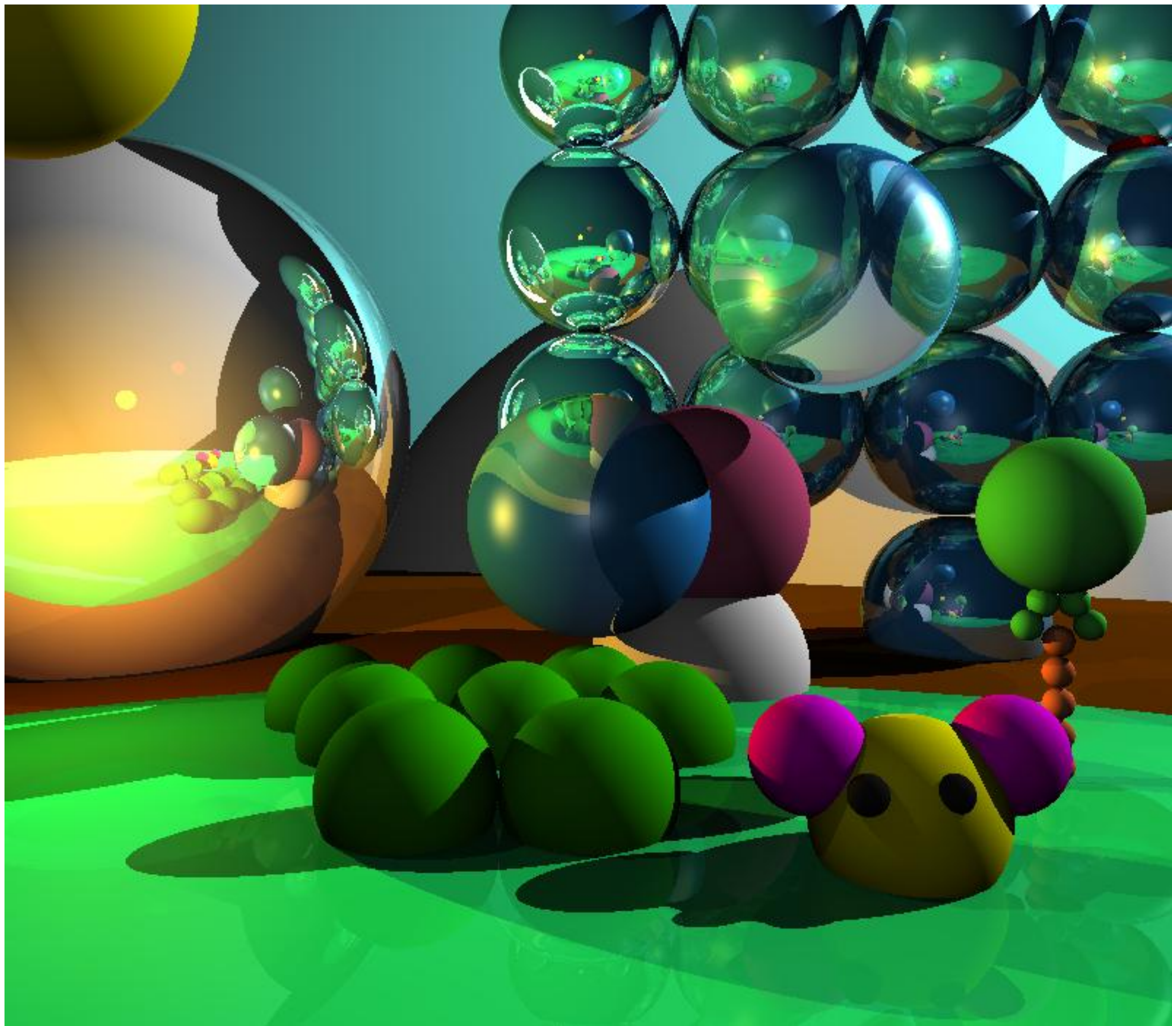
Soubor:	sceneHouse0.txt
Rozlišení:	800x600
Hloubka:	3
Objektů:	96
Světel:	4
Čas (s):	8,718


Soubor: sceneHouse1.txt
Rozlišení: 800x600
Hloubka: 3
Objektů: 96
Světel: 4
Čas (s): 9,136

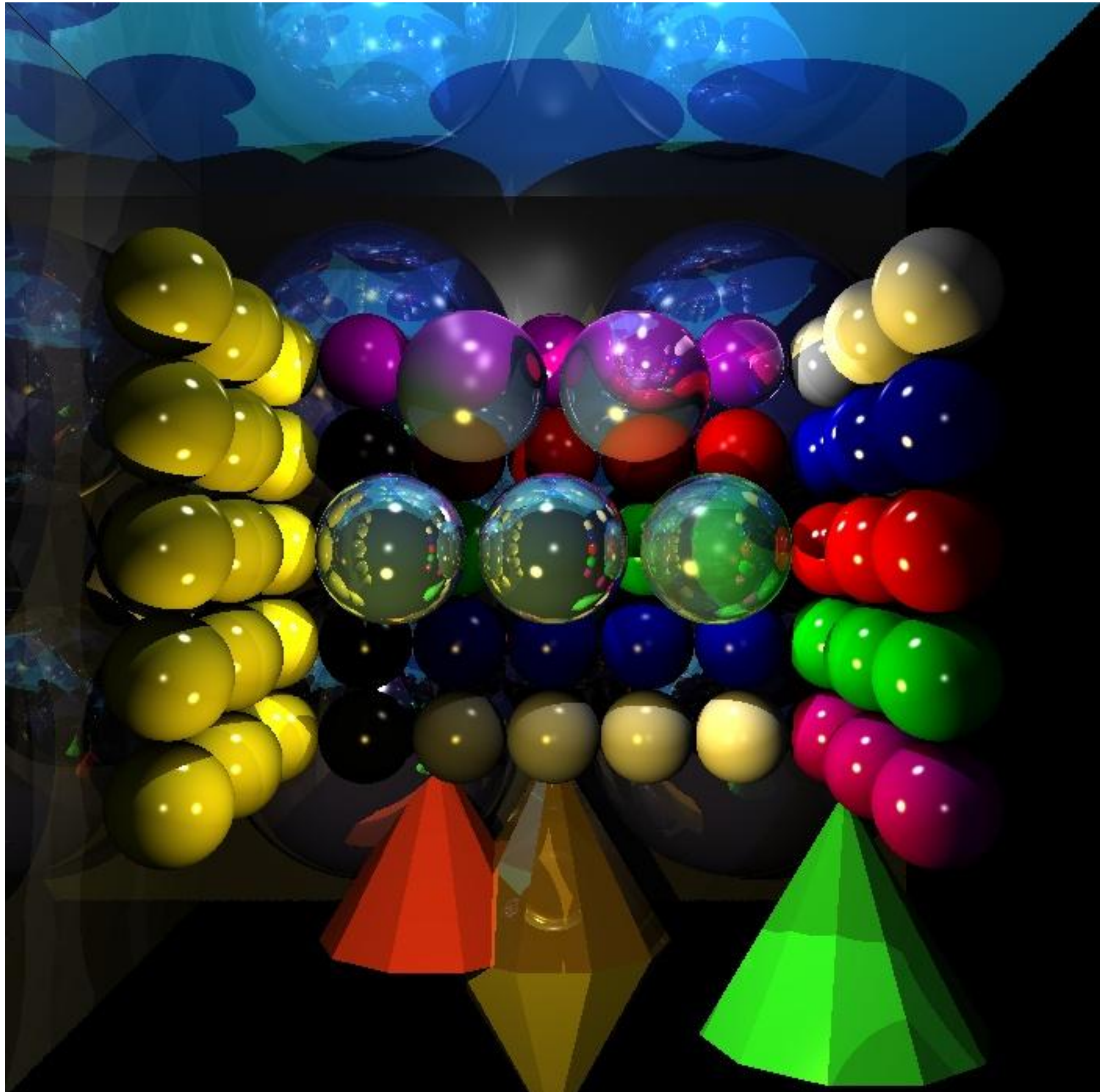

Soubor: sceneHouse2.txt
Rozlišení: 800x600
Hloubka: 3
Objektů: 96
Světel: 4
Čas (s): 11,606



Soubor:	sceneHouse3.txt
Rozlišení:	800x600
Hloubka:	3
Objektů:	96
Světel:	4
Čas (s):	10,226



Soubor:	sceneTest.txt
Rozlišení:	800x700
Hloubka:	6
Objektů:	50
Světel:	4
Čas renderování (s):	13,702



Soubor:	sceneTime.txt
Rozlišení:	700x700
Hloubka:	4
Objektů:	111
Světel:	5
Čas (s):	18,512

Příloha B

Popis souboru pro uložení scény

V následujícím textu je použito toto značení:

x, y, z pro datový typ float

i pro datový typ integer

[…] pro komentář

Povinná hlavička souboru (4 řádky):

```
scene file version x
Number of lights: i
Number of objects: i
Number of materials: i
```

Začátek seznamu definovaných objektů:

```
<Light>                        <Material>
<Sphere>                       <Screen>
<Triangle>                     <Camera>
```

Další položka v seznamu objektů:

```
<next>
```

Konec seznamu objektů:

```
<end>
```

Vlastnosti objektu <Sphere>

```
radius: x
```

Vlastnosti objektu <Triangle>

```
vertices: x, y, z; x, y, z; x, y, z
```

Vlastnosti objektu <Camera>

```
zoom: x
direction: x, y, z          [vektor, směr pohledu kamery]
up direction: x, y, z       [vektor, musí být kolmé k direction:]
right direction: x, y, z    [vektor, kolmý k dvěma předcházejícím]
```

Vlastnosti shodné pro více objektů:

```
position: x, y, z           [<Sphere>,<Camera>,<Light>]
color: x, y, z              [<Sphere>,<Triangle>,<Light>]
material: i  [<Sphere>,<Triangle>][materiály jsou číslovány od
             1, tak jak jsou za sebou uvedeny v souboru]
```

Vlastnosti objektu <Screen>

```
width: x        [rozlišení výstupního obrázku]
height: y
depth: i        [hloubka sledování paprsku]
```

Vlastnosti objektu <Material>

```
shininess: i
specular: i
reflection: x   [viditelnost odrazu, rozsah <0,1>]
refraction: x   [index lomu materiálu]
transparency: x [průhlednost objektu, rozsah <0,1>]
```

Je důležité mít v hlavičce souboru zadaný správný počet jednotlivých objektů uvedených v souboru a mít ukončené seznamy značkou <end>. Také je nutné dodržování mezer mezi čísly.