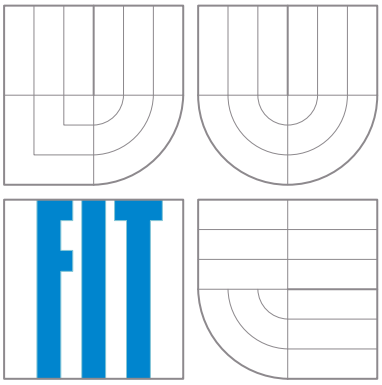


Grafické a multimediální procesory

Raytracer na CUDA

2. ledna 2015

Autor: Pavel Macenauer, xmacen02@stud.fit.vutbr.cz
Jan Bureš, xbures19@stud.fit.vutbr.cz

Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

Obsah

1 Zadání 2
2 Použité technologie 2
3 Výsledky a použité znalosti 2
3.1 Srovnání CUDA a Aurelius raytracerů 2
3.2 Testovací scéna 2
3.3 Výsledky srovnání 5
3.4 Paralelizace na GPU 5
3.5 Optimalizace 6
3.6 NVidia CUDA 6
4 Ovládání vytvořeného programu 7
5 Rozdělení práce v týmu 7
6 Co bylo nejpracnější 7
7 Zkušenosti získané řešením projektu 7
8 Autoevaluace 8
9 Doporučení pro budoucí zadávání projektů 8
Literatura 11

1 Zadání

- Implementace Raytraceru pomocí technologie CUDA v následujícím rozsahu:
  - Geometrická primitiva: roviny, koule, trojúhelníky, válce
  - Načítání modelů z běžně používaných formátů
  - Phongův osvětlovací model
  - Bodové zdroje světla a stíny
  - Odlesky
- Akcelerace raytracingu na GPU
- Vygenerování demonstrační scény a její vykreslení
- Srovnání s CPU implementací

2 Použité technologie

Použité technologie:

- NVidia CUDA 6.5, `C++11`
- Microsoft Visual Studio 2013 + NVidia NSight
- GLUT

Je přiložen projekt pro Visual Studio (testováno na Microsoft Visual Studio 2013). Pro samotný překlad je pak třeba mít nainstalované NVidia CUDA 6.5. a používat překladač podporující `C++11`.

3 Výsledky a použité znalosti

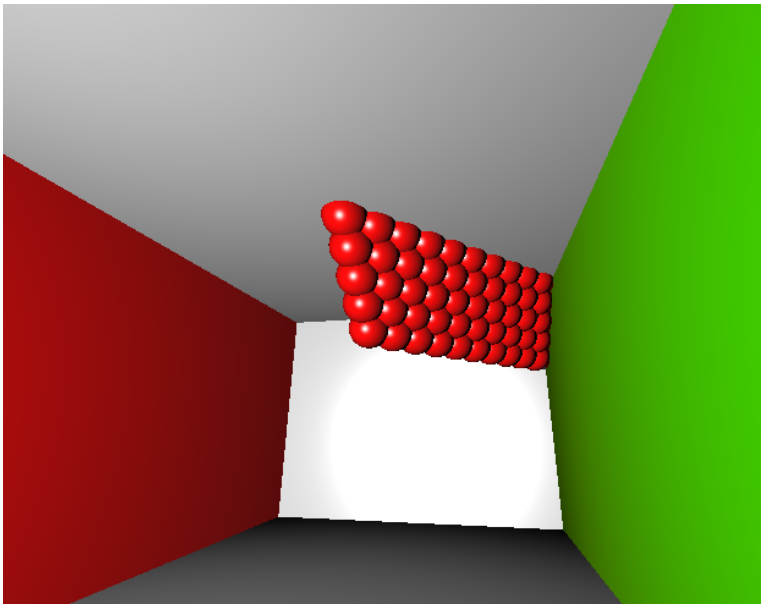
3.1 Srovnání CUDA a Aurelius raytracerů

Raytracer jsme testovali ve srovnání s CPU Raytracerem Aurelius, veřejně dostupným Raytracerem pro předmět PGR.

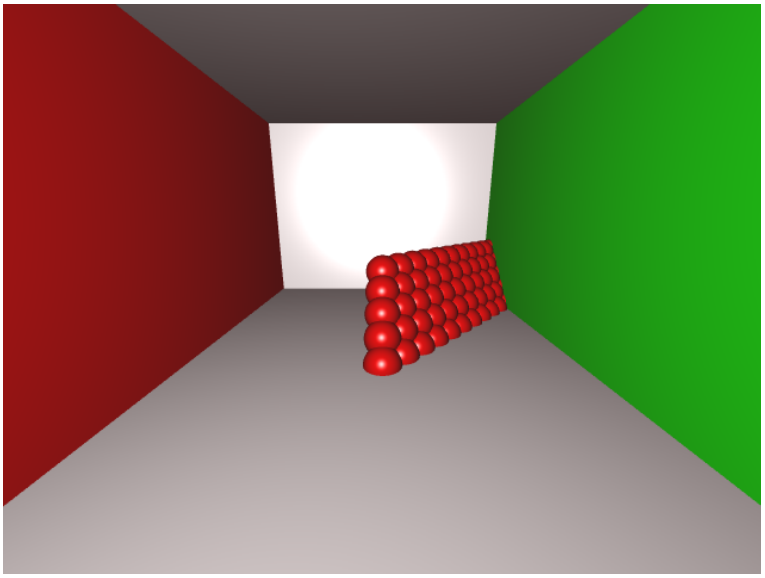
Oba Raytracery jsou implementovány rozdílně, je tak komplikované vytvořit identické scény. Co ale lze je vytvořit scény podobně komplexní. Vzhled na CUDA raytraceru je mírně kontrastnější a pohled kamery otočen (viz. obrázky 1 a 2).

3.2 Testovací scéna

- počet rovin: **6**
- počet koulí: **5, 10, 20, 50, 100, 500**
- hardware: **NVidia Quadro K1000M, Intel Core i7-3610QM 2.3 GHz**



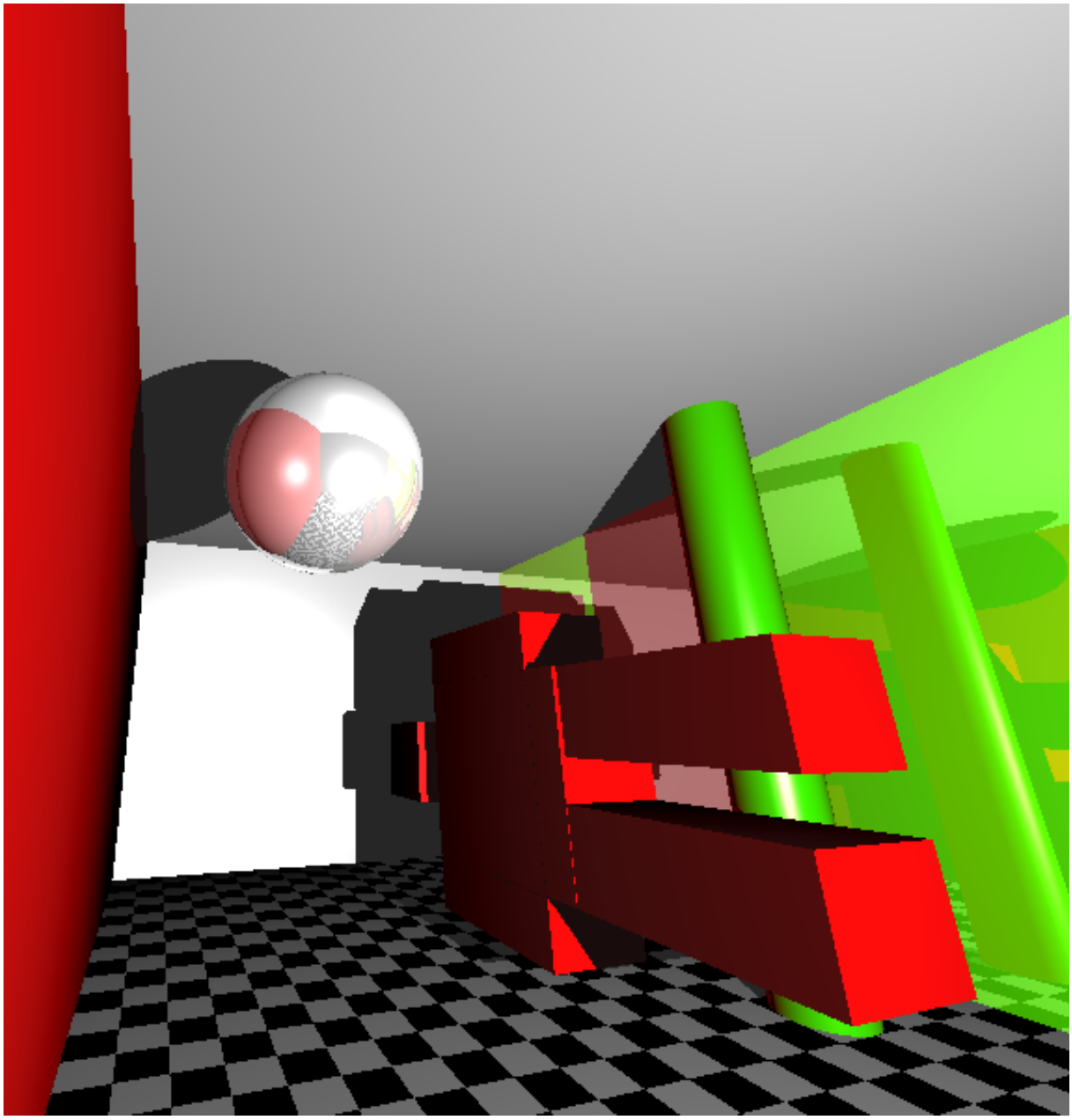
Obrázek 1: Testovací scéna na CUDA raytraceru



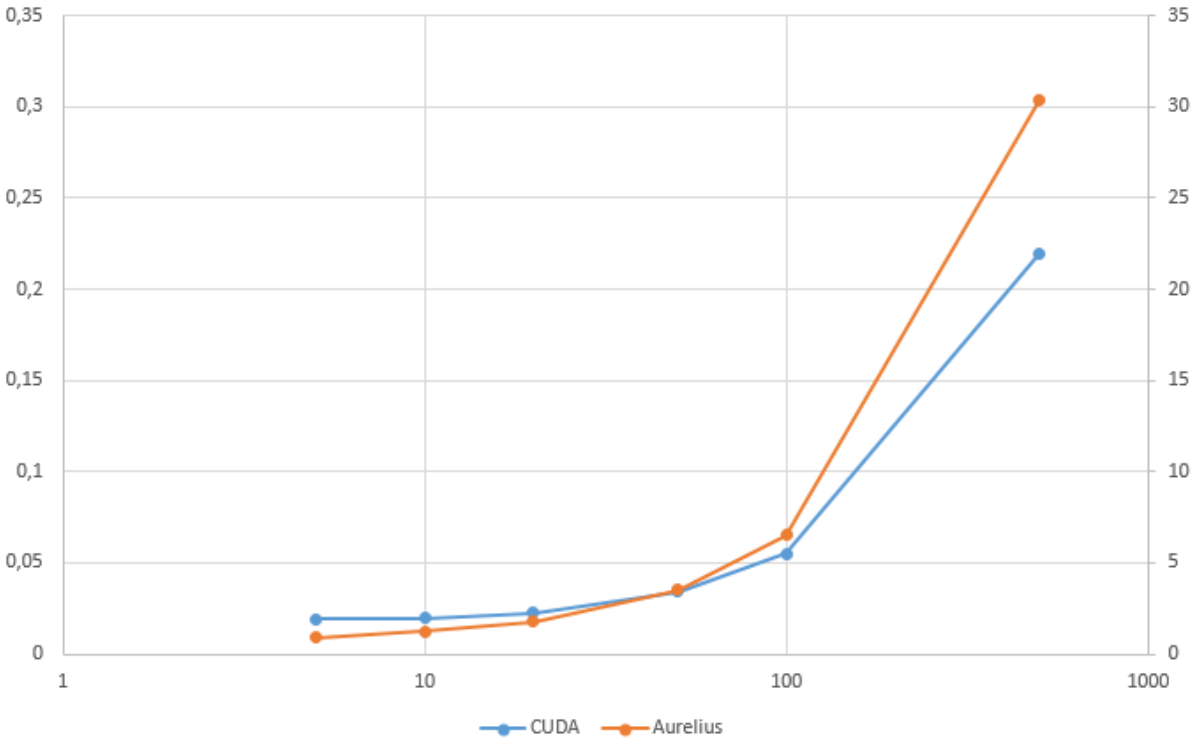
Obrázek 2: Testovací scéna na raytraceru Aurelius

Počet koulí	5	10	20	50	100	500
CUDA [s]	0,0193	0,0195	0,0224	0,0341	0,05521	0,2197
Aurelius [s]	0,905	1,233	1,779	3,541	6,505	30,373

Tabulka 1: Srovnání podobně komplexních scén na raytraceru Aurelius a CUDA raytraceru pro daný počet koulí



Obrázek 3: Komplexnější scéna obsahující koule, roviny, trojúhelníky a válce s odrazivým materiálem (koule a stěna), procedurálními texturami (šachovnice) a načítáním modelů z obj (ležící panák)



Obrázek 4: Srovnání raytraceru Aurelius a CUDA raytraceru *(osa Y vlevo – doba syntézy pro RT CUDA, osa Y vpravo – doba syntézy pro RT Aurelius)*. RT CUDA je v průměru 100x rychlejší, proto jsme pro srovnání nárůstu doby výpočtu zvolili rozdílné osy pro oba RT.

3.3 Výsledky srovnání

Syntéza raytracerem Aurelius byla cca 100x pomalejší, než-li syntéza podobné scény na GPU. S komplexností scény se rozdíly mezi CPU a GPU verzí ještě zvětšují.

3.4 Paralelizace na GPU

Rozvržení paměti

Raytracer ke svému běhu potřebuje pouze informace o scéně:

- světla (constant memory)
- nastavení kamery (constant memory)
- informace o materiálech (constant memory)
- databáze primitiv (constant memory)

Světla, nastavení kamery a materiály jsou data používaná pro výpočet každého paprsku. Zvolili jsme proto uložení v constant memory, která vykazuje nejrychlejší přístupový čas a zároveň je broadcastována pro všechna vlákna. Diskutabilní může být uložení materiálu pro scény,

v případě velkého množství materiálu. Constant memory je nejen omezena velikostí 64 KB, ale je i z důvodu broadcastu neefektivní pro scény, kde paprsky budou trefovat primitiva z různých materiálů.

Databázi primitiv je vhodné ukládat v konstantní paměti z hlediska broadcastu, ale pro scény s více jak 500-600 primitivy se nevejdeme do max. velikosti constant memory (64 KB), což je i omezení velikosti scény CUDA raytraceru. Šlo by využít globální paměť, což by však vedlo ke zpomalení.

Paralelizace výpočtu

Pro každý pixel syntetizovaného obrazu se vyšle paprsek – vytvoří vlákno a to vypočte výslednou barvu (včetně odrazů a stínů).

3.5 Optimalizace

Během implementace jsme tématiku zkoumali a napadly nás následující optimalizace. Z časového hlediska, které by často vyžadovaly kompletně přepsat aplikaci a ve výsledku by nemusely znamenat zlepšení, nejsou z většiny implementovány.

Odraz paprsku

Při odrazu paprsku od odrazivého materiálu dochází k vyslání dalšího paprsku. Vlákna, která nic netrefí nebo trefí neodrazivý materiál vracejí výslednou barvu, nicméně stále se musí čekat na vlákna, která se odrážejí. Počítá tak pouze malé množství vláken a zbytek čeká.

Nápad na zlepšení je následující:

1. Raytracer vyšle paprsek
2. Zjistí se která vlákna vrátila výsledek (trefila neodrazivý materiál nebo nic) a která dál chtějí počítat (odrážet se)
3. Přenese se stav výpočtu zbývajících vláken do paměti
4. Spustí se kernel pro zbývající vlákna

Akcelerační struktury

CUDA raytracer využívá jako akcelerační struktury KD-tree a BVH. Nejedná se sice o rozšíření implementovaná v rámci rozsahu předmětu GMU, nicméně i samotné struktury lze optimalizovat.

Intuitivním přístupem je sestavit struktury na CPU a následně vše nakopírovat na GPU. Struktury je ale možné sestavit i na GPU, což je mnohem rychlejší. [15]

3.6 NVidia CUDA

Hlavní co jsme museli nastudovat jsou vědomosti ohledně technologie CUDA tedy struktura pamětí, jak je používat pomocí jejího `C++` API a v kombinaci s programováním na CPU.

K debugování a programování jsme následně využívali vývojové prostředí Microsoft Visual Studio 2013 s NVidia NSight.

4 Ovládání vytvořeného programu

Raytracer nabízí možnost využití některých akceleračních optimalizací avšak pro tyto optimalizace je nutné raytracer znovu přeložit. Volbu provedeme pomocí podmíněného překladu za použití konstant v souboru constatns.h. Pro BVH je nutné definovat proměnou ACC_BVH pomocí "#define ACC_BVH". Pro KD-Tree konstantu ACC_KD_TREE. Stejně jako optimalizační algoritmy je možné využít různých aplikací raytraceru (neostré stíny – OPT_SOFT_SHADOWS, hloubka ostrosti – OPT_DEPTH_OF_FIELD, automaticky pohyb kamery – OPT_CAMERA_SHIFT nebo bilineární samplování – OPT_BILINEAR_SAMPLING).

V raytraceru se nastavuje pozice kamery pomocí kláves Q, W, E, A, S, D (A a D ve směru osy x, W a S ve směru osy y, Q a E ve směru osy z) avšak bod, na který kamera směřuje se nezmění. Pokud raytracer přeložíte pro výpočet hloubky ostrosti můžete měnit ohniskovou vzdálenost pomocí R a T.

Komplexnější scény zapsané ve formátu OBJ je možné v ratraceru zobrazit tak, že soubor s koncovkou obj vložíme do složky, ze které raytracer spouštíme.

5 Rozdělení práce v týmu

Oba jsme pracovali na všem, nicméně specializovali jsme se následovně:

- **Jan Bureš**: Phongův osvětlovací model, úprava bilineární interpolace, hloubka ostrosti, měkké stíny, válec, trojúhelník, rovina, především raytracing-části raytraceru...
- **Pavel Macenauer**: Základní kostra programu, optimalizace využitých paměťových jednotek, základ pro bilineární interpolaci, načítání formátu OBJ, akcelerační struktury, především CUDA-části raytraceru...

6 Co bylo nejpracnější

- **Jan Bureš**: Na celé práci bylo asi nejsložitější upravit algoritmus sledování paprsku tak, aby fungoval na architektuře CUDA. Dále pak správné rozdělení vláken do warpů a rozhodnout, který druh paměti použít pro které proměnné. Mnoho času bylo zapotřebí také věnovat rekurzi, kterou by sice CUDA měla podporovat od verze 3.0, ovšem stále program havaroval kvůli přetečení zásobníku.
- **Pavel Macenauer**: Celkové zprovoznění CUDA a navázání na OpenGL + následné vykreslení. To především proto, že dokumentace a materiály jsou především v podobě vygenerované dokumentace od společnosti NVidia a materiálů na internetu k této tématice moc není. Dále i parsování souboru OBJ a vkládání jednotlivých trojúhelníku do paměti.

7 Zkušenosti získané řešením projektu

Naučili jsme se více o architektuře CUDA, získali představu o psaní paralelních algoritmů a optimalizovali značně rychlost výpočtu. Zdokonalili jsme raytracer do podoby, kdy již nevykresluje jen koule, ale i komplexnější scény, odráží paprsky. Optimalizovali jsme rychlost, vyzkoušeli si implementovat phongův osvětlovací model a raytracer v praxi. Prostudovali možné algoritmické optimalizace raytraceru a některé z nich implementovali.

8 Autoevaluace

Technický návrh (85%): Tvorbu programu jsme si naplánovali na jednotlivé iterace, a tak až na pár úprav, které nás napadli během implementace nebylo nutné přepisovat již implementované části. Především se jednalo o části týkající se správy paměti, kdy s každým typem paměti na GPU se pracuje trochu jinak.

Programování (75%): Kód je dobře strukturovaný, ale mohl by být více okomentován, např. pro vygenerování použitelné doxygen dokumentace. Implementovaný raytracer je možné rozšiřovat a bez sebevětších komplikací přidávat další doplňky.

Vzhled vytvořeného řešení (80%): Scéna vypadá celkem pěkně. Jednotlivá primitiva mají ostré hrany, tak by neuškodilo implementovat antialiasing. Některé optimalizace kvalitu obrazu mírně zhorší, ale vždy dle očekávání.

Využití zdrojů (90%): Hodně jsme využili již implementovaný raytracer Aurelius k lepšímu pochopení raytracingu a trochu méně pak dostupnou literaturu. Zdroje o CUDě a jejím zapojení jsme získali především z vyhledávače Google.

Hospodaření s časem (70%): Na raytraceru jsme pracovali průběžně, využití GIT repositáře umožnilo, že jsme vždy každou volnou chvíli něco málo naimplementovali. Uprostřed semestru naše snaha mírně opadla kvůli jiným povinnostem ve škole. Na závěr jsme se snažili vše dokončit v čas, což se i povedlo.

Spolupráce v týmu (95%): Od začátku jsme komunikovali ohledně podmínek spolupráce. Následné programování pak probíhalo bez problémů a o všem jsem se navzájem informovali přes instantní mluvítka jako Skype nebo Facebook. Veškeré změny jsme pak evidovali v repositáři na serveru GitHub.com, kde jsme i vybudovali společné vývojové prostředí.

Celkový dojem (90%): S raytracingem se setkáváme ve více předmětech, proto jsme projekt kombinovali i s předmětem PGP, pro který byly implementovány akcelerační struktury. Celý projekt už je celkem komplexní a jednoznačně nad rámec jednoho předmětu. Nejedná se již o jednoduchý GPU raytracer, ale raytracer schopný vykreslovat různé druhy primitiv, modely, využívá různé druhy akcelerace a rozšiřující vizuální aplikace.

Projekt je celkově zajímavý tím, že se jedná o komplexní raytracer, který se postupně formuje do podoby raytracingového enginu/frameworku. Na rozdíl od velkého množství projektů, které zmizí v propadlišti dějin, se jedná o práci, která by i mohla zaujmout své jedinečné místo na internetu.

9 Doporučení pro budoucí zadávání projektů

- Zveřejnit zadání pro projekty již od 1. týdne - umožní to nastudovat tématiku dopředu a vytvořit tak prostor pro diskuzi na přednáškách. Navíc na začátku semestru je nejvíce času, dále pak jsou úkoly, půlsemestrální písemky a je možné se vyhnout práci na poslední chvíli.

- Kostry pro jednotlivé projekty. Některé projekty se zpracovávají během let opakovaně. Určitě by tak pro ně šlo vytvořit kostry (něco jako pro domácí úkoly), které by umožňovali odstranit onu část vše rozchodit, nainstalovat, napsat základy a věnovat se komplexnějším věcem. Umožnilo by to i lehčí kontrolu pro vyučující.

Literatura

[1] Kadi BOUATOUCH. Stochastic ray tracing [pdf], [cit. 2014-12-30]. http://www.irisa.fr/prive/kadi/ESIR_IN2/StochasticRayTracing.pdf.

[2] COMPUTER GRAPHICS LAB, ALEXANDRA INSTITUTE. Triers CUDA ray tracing tutorial. http://cg.alexandra.dk/2009/08/10/triers-cuda-ray-tracing-tutorial/.

[3] Don FUSSELL. Distribution ray tracing [pdf], 2010, [cit. 2014-12-30]. http://www.cs.utexas.edu/~fussell/courses/cs384g/lectures/lecture11-Drt.pdf.

[4] John HART. Distributed ray tracing [pdf], [cit. 2014-12-30]. http://luthuli.cs.uiuc.edu/~daf/courses/ComputerGraphics/Week3/distributed-final.pdf.

[5] L. GRANT. Bounding Volume Hierarchies (BVH) – A brief tutorial on what they are and how to implement them. http://www.3dmuve.com/3dmblog/?p=182http://www.3dmuve.com/3dmblog/?p=182.

[6] N. GUPTA. What is constant memory in CUDA? http://cuda-programming.blogspot.cz/2013/01/what-is-constant-memory-in-cuda.html.

[7] NVIDIA Corporation. CUDA Toolkit Documentation. http://docs.nvidia.com/cudahttp://docs.nvidia.com/cuda.

[8] ÚPGM, FIT VUT. Přednáška PGP (Fotorealistcké zobrazování, optimalizace sledování paprsku).

[9] ÚPGM, FIT VUT. Přednáška PGR (Realistické zobrazování I - Ray Tracing). https://www.fit.vutbr.cz/study/courses/PGR/private/lect/PGR-RayTracing-I.pdf.

[10] ÚPGM, FIT VUT. Ray-Tracer Aurelius. https://www.fit.vutbr.cz/study/courses/PGR/private/Aurelius.zip.

[11] Přispěvatelé Stackoverflow.com. Rady ohledně rekurze v CUDA 4.0. http://stackoverflow.com/questions/19013156/how-does-cuda-4-0-support-recursion.

[12] Přispěvatelé Wikipedie. Phongův osvětlovací model. http://cs.wikipedia.org/wiki/Phong%C5%AFv_osv%C4%9Btlovac%C3%AD_model.

[13] COOK Robert, PORTER Thomas, and CARPENTER Loren. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, New York, NY, USA, 1984. ACM Press.

[14] Pavel STRACHOTA. Raytracing a další globální zobrazovací metody [pdf], 3.5.2013, [cit. 2014-12-30].
http://saint-paul.fjfi.cvut.cz/base/sites/default/files/POGR/POGR2/12.raytracing.pdf.

[15] Tero Karras, NVidia Research. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees.
https://research.nvidia.com/sites/default/files/publications/karras2012hpg_paper.pdf, 2012.