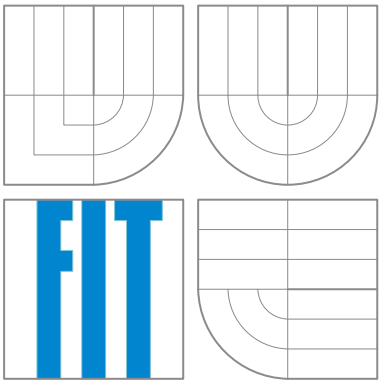


Grafické a multimediální procesory

Raytracer na CUDA

1. 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

- 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

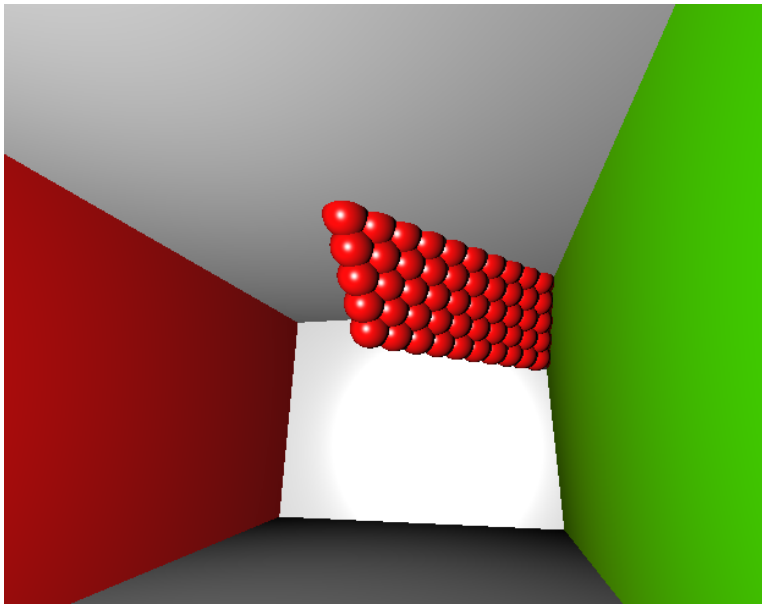
3.1 Srovnání CUDA a Aurelius raytracerů

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

Vzhledem k tomu, že oba Raytracery jsou implementovány rozdílně, je komplikované vytvořit identické scény, nicméně lze vytvořit scény podobně komplexní.

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

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

3.3 Výsledky srovnání

Syntéza raytracerem Aurelius byla cca 100x pomalejší, než-li syntéza podobné scény na GPU. S komplexností scény roste výpočet Aurelia rychleji.

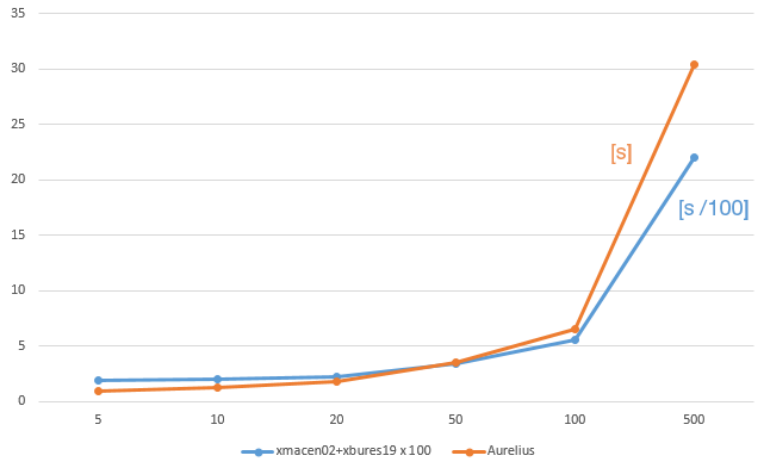
3.4 Paralelizace na GPU

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, je tedy vhodné používat constant memory, která vykazuje nejrychlejší přístupový čas a zároveň je broadcastována pro všechna vlákna. Diskutabilní je uložení materiálu pro scény, kde jich je hodně. Constant memory je nejen omezena velikostí 64 KB, ale je i z důvodu broadcastu neefektivní pro scény, kde paprsky budou trefovat primitiva s různými materiály.



Obrázek 2: Srovnání raytraceru Aurelius [**s**] a CUDA raytraceru [**s/100**]

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

3.6 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 akcelerovat.

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. [18]

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 je možné pomocí kláves Q, W, E, A, S, D nastavit pozici kamery (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 je raytracer přeložen pro výpočet hloubky ostrosti je možné měnit ohniskovou vzdálenost pomocí šipky nahoru a dolů.

4.1 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.

4.2 Krátce o Bounding Volume Hierarchy

Jedná se o akcelerační strukturu při výpočtech v počítačové grafické.

Konstrukce

- Určí se konstanta pro počet objektů v nejnižší vrstvě stromu, tj. kolik listů bude mít každý předposlední uzel stromu.
- Řadíme objekty podle svých souřadnic (zda-li X, Y nebo Z záleží na programátorovi) a dělíme prostor vždy na 2.
- Skončíme ve chvíli, kdy v daném podprostoru je méně objektů, než-li námi daná konstanta, tyto objekty pak přidáme do seznamu daného uzlu

Výhody V případě raytracingu nemusíme testovat zda-li paprsek protne všechny primitiva ve scéně, ale vždy testujeme zda-li trefíme některý z podprostorů: kořen stromu má 2 podprostory, každý z nich zase další 2. Ve chvíli kdy narazíme na podprostor, který již nemá žádné další, ale pouze jednotlivé listy, tak provedeme test na průnik pouze těch primitiv obsažených pod

daným uzlem. Počet testů je tak roven `HLOUBKA STROMU * 2 + POČET LISTŮ NA UZEL`, což bývá podstatně méně u velkých scén, než-li celkový počet primitiv.

5 Rozdělení práce v týmu

- **Jan Bureš**: Phongův osvětlovací model, úprava bilineární interpolace, především raytracing-části raytraceru, hloubka ostrosti, neostré stíny
- **Pavel Macenauer**: Základní kostra programu, optimalizace využitých paměťových jednotek, základ pro bilineární interpolaci, 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. Mno 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**: Nejprve celé zprovoznění CUDy a způsob, kterým předávat data mezi OpenGL, CUDou a následně je vykreslit stálo mnoho nervů a několik šálků kávy navíc. Především proto, že nové CUDA 3.0 API obsahuje nové metody pro komunikaci s OpenGL, nicméně dokumentace a materiály k tomu jsou především v podobě vygenerované dokumentace od společnosti NVidia. Dále ještě za zmínku stojí implementace BVH a celkové nastudování o principů.

7 Zkušenosti získané řešením projektu

Naučili jsme se více o architektuře CUDA a získali představu o psaní paralelních algoritmů, vyzkoušeli si implementovat phongův osvětlovací model a raytracer v praxi. Dále jsme si 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 dále rozšiřovat a bez sebevětších komplikací přidávat další doplňky.

Vzhled vytvořeného řešení (80%): Scéna vypadá celkem pěkně, velmi zřídka lze pozorovat tečky, které narušují plynulost obrazu. Způsobené jsou pravděpodobně používáním datového typu float a nepřesností, které způsobuje. Některé optimalizace kvalitu obrazu mírně zhorší, ale vždy dle očekávání. Není problém do budoucna do implementovat i další primitiva, která by umožnila tvorbu komplexnějších scén.

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%): Začali jsme hned jak jsme obdrželi zadání, nicméně chvíli trvalo, než jsme vůbec měli něco co by něco vypočítalo na CUDě a předalo k zobrazení. 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%): Celý projekt se nám jevil mírně obtížnější vzhledem k ostatním projektům na škole, především z důvodu dostupnosti materiálů, kterých je na internetu spousty, nicméně vyfiltrovat z nich použitelné informace je časově náročné. Získané vědomosti a zkušenosti jsou určitě přínosem. Vybrali jsme si téma projektu kvůli architektuře CUDA, kterou jsme předtím v praxi nikdy nepoužívali a tímto jsem si udělali nejen lepší představu o samotné technologii ale i o moderních grafických kartách a naučili se pracovat s další technologií, která má své místo v softwarové budoucnosti. Celý projekt byl zajímavý také tím že jsme si vyzkoušeli vytvořit grafickou aplikaci úplně od začátku.

9 Doporučení pro budoucí zadávání projektů

Následující body jsou spíše nápady pro vytvoření možnosti v daném časovém rozsahu zpracovat komplexnější projekt a přeskočit onu fundamentální část projektu "aby to něco dělalo":

- Specifikovat požadavky k zadanému tématu (která primitiva implementovat, do jaké míry implementovat raytracer, případně vypsat, co změřit ...)
- Sestavit kostru projektu, např. projekt v MS Visual Studiu/QMake/Makefile s potřebnými hlavičkovými soubory obsahující prototypy metod

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] Grabner, H. ; Sochman, J. ; Bischof, H. ; Matas, J. Training sequential on-line boosting classifier for visual tracking. http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4761678, 2008.

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

[6] Herout, A. ; Josth, R. ; Juranek, R. ; Havel, J. ; Hradis, M. ; Zemcík, P. Real-time object detection on CUDA. http://link.springer.com/article/10.1007%2Fs11554-010-0179-0, 2011.

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

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

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

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

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

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

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

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

[15] 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.

[16] Sochman J. ; Matas J. WaldBoost - learning for time constrained sequential detection . http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1467435, 2005.

[17] 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.

[18] 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.

[19] Zemcik, P. ; Juranek, R. ; Musil, P. ; Musil, M. ; Hradis, M. High performance architecture for object detection in streamed videos. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6645559, 2013.