Ray tracer

Petr Beránek
beranpe6@fel.cvut.cz

závěrečná zpráva k projektu z předmětu A4M39GPU
zimní semestr 2015/2016

I. Popis řešeného problému

Cílem semestrální práce bylo vytvořit raytracer využívající KD strom jako akcelerační strukturu a persistentní vlákna, přičemž jsou porovnány různé mětody procházení daného KD stromu[3].

II. Popis implementace na CPU

Při startu aplikace je uživatel vyzván, aby zadal parametry pro běh programu. Následně je postavena scéna, do které jsou vloženy objekty podle toho, kterou scénu uživatel vybral.

Scény, které je možné vykreslovat podporují pouze základní objekty. Mezi tyto objekty patří trojúhelníky, koule a bodová světla. KD strom obsahuje koule a trojúhelníky a pri jeho vytváření je použit poměrně jednoduchý algoritmus pro vybírání spliting plane. Osa, podle které jsou jednotlivé uzly KD stromu děleny je volena cyklicky a hledání hodnoty s, kde bude uzel rozdělen je získána jako $s = \frac{m+M}{2}$. Hodnota m je v tomto případě minimální dolní souřadnice na dané ose a M je maximální horní souřadnice. Horní a dolní souřadnice objektu se získávají z osově zarovnaného kvádru obalujícího daný objekt. Pokud se objekt nachází na obou stranách dělící roviny, není dělen a je přidán na obě dvě strany. V úplně nejhorším případě, pokud by byly objekty opravdu špatně uspořádány, byly by všechny objekty přidány na obě dvě strany. Tím by se zvýšila hloubka stromu, ale náročnost kolize by zůstala stále stejná. Proto když součet objektů po rozdělení na obě strany překročí 1.8násobek původního počtu objektů, je dělení podle dané osy ukončeno a pokračuje se dělením podle následující osy v cyklu. Pokud jsou osy vyčerpány a nepodařilo se dělení podle žádné osy, je dělení ukončeno a uzel se prohlásí za list.

Pro procházení stromu jsou použity 2 různé algoritmy. První algoritmus je jednoduchý a přímočarý.[3]

```
1  TREE - KD tree with objects
2  RAY - traced ray
3
4  leaf L;
6  point E = RAY.origin;
7  if(E is not in TREE)
8  {
9    E = enter point of RAY in TREE
10 }
11 while(E is in TREE)
12 {
13   L = leaf of TREE that contains E;
14   collide RAY with all objects in L
15   if(collided){break;}
16   E = exit point of RAY from L
17 }
```

V každém průchodu je cyklem je (řádek 13) procházen celý strom od kořene, až k listu při hledání, do kterého uzlu patří bod E. To ovšem může snamenat značný overhead při procházení hodně hlubokých stromů.

Druhý algoritmus[1][2] využívá lehce upravený KD strom. Každý uzel obsahuje 6 odkazů, kde každý ukazuje na sousední uzly. Jelikož u KD stromu dochází k dělení uzlu na různých místech, nelze předem říci, kde se přesně bude dělící rovina v sousedním uzlu nacházet. Proto odkazy budou v každém případě ukazovat na uzly stejné, nebo nižší hloubky. Pokud uzel nemá v daném směru souseda (vystoupí se mimo prostor KD stromu) bude hodnota souseda NULL a dá se rychle zjistit, že paprsek opustil strom.

Při použití této metody je nezbytné použít modifikovanou metodu na hledání výstupního bodu z uzlu (metoda kromě bodu vrací také integer v rozmezí 0-5 určující ze které stěny uzlu paprsek vystoupil) a navrácená hodnota je využita pro indexoání pole sousedů. Následně je list hledán pouze z daného sousedního bodu a ne z kořene.

Samotné sledování paprsku probíhá tak, že paprsky jsou na počátku vygenerovány a uloženy do fronty. Následně je fronta postupně procházena a paprsky jsou zpracovávány jeden po druhém. Paprsky u sebe drží veškeré informace potřebné pro jejich zpracování i kam má být výsledná hodnota zapsána, takže nezáleží na pořadí jejich zpracování. V původní verzi programu

bylo zpracování napsáno tak, že když paprsek vytvoří další stínové paprsky, refracted paprsky nebo reflected paprsky, tak byly přidány na konec fronty. Později bylo zpracování upraveno tak, jsou paprsky procházeny rekurzvně a žádně další nejsou do bufferu přidány.

III. Paralelní řešení problému

Jedním ze základních rozdílů u paralelního řešení je převedení všech věcí potřebných pro zpracování paprsků na buffery (paprsky, KD strom, trojúhelníky, koule, světla, materiály). Jelikož se jeden objekt může nacházet ve více listech bylo by zbytečné tento objekt duplikovat. Proto jsou přidány pole s index buffery pro koule a trojúhelníky. Každý list KD stromu má v bufferech vyhrazený prostor, kde má zaznamenané indexy koulí a trojúhelníků, které obsahuje (indexy odkazují do přísluších bufferů).

Další významnou změnou je využití persistentních vláken.

```
while(rayCount > 0)
{
  if(thrId == 0)
  {
    if(rayCount < thrCount)
    {
      countActive = rayCount;
      rayCount = 0;
    }else{
      countActive = thrCount;
      rayCount -= thrCount;
    }
  }

  syncthreads();
  if(thrId < countActive)
  {
    rayIs = rayCount + thrIndex;
    copy ray on position
    rayId to local variables;
  }

  syncthreads();
  if(thrId < countActive)
  {
    trace ray;
  }
  syncthreads();
}
```

V prvním kroku první vlákno ukousne kus ray bufferu o velikosi bloku a určí kolik bude altivních vláken (pokud není dost paprsků pro všechna vlákna, některé nebudou aktivní). V druhém kroku si vlákna načtou do lokálních proměnných parametry paprsku. Ve třetím kroku, když už vlákna skutečně nebudou potřebovat paprsky z bufferu (a jejich pozice mohou být přepisovány) se provede samotné zpracování paprsku.

Významnou změnou je v paralelní verzi také to, že vlákna nejdou v rekurzi příliš hluboko. Stínové paprsky jsou zpracovány okamžitě (hloubka 1), ale ty se nezanořují dále, refracted paprsky jsou přidány do fronty paprsků a zpracují se v dalším kole(hloubka 0). Jediné paprsky, které se zpracují okamžitě a mohou se zanořovat jsou reflected paprsky. ke každému paprsku existuje nejvýše jeden reflected paprsek, který se vytváří na samém konci zpracování.

Pro zpracování reflected paprsků jsem se rozhodl upravit zpracování tak, že původní paprsek je přepsán reflected paprskem a metoda vrátí příznak říkající, zda došlo k přepsání a tudíž je nutné nový paprsek zpracovat.

Toto řešení jsem zvolil z jednoduchého důvodu. Velikost paměti je omezená a tudíž je omezená velikost bufferu paprsků. Pokud by se všecny paprsky přidávaly do bufferu paprsků, mohlo by v extrémních případech dojít k tomu, že by byla paměť vyčerpána. Při rekurzivním zpracování paprsků docházelo v praxi k problémům při zpracování. Pokud se paprsky mohly odrazit více než $3x$, vlákna byla ukončena a vykreslení scény skončilo neúspěšně. Zde ovšem může docházet k problému, že každé vlákno půjde do jiné hloubky, čímž některá vlákna půjdou stále hlouběji a některá vlákna nebudou mít co dělat a čím hloubější někeré vlákno půjde, tím více vláken bude pravděpodobně nic nedělat.

IV. Implementační detaily – CUDA

Implementace v CUDA je velmi podobná té na CPU, jelikož už CPU varianta byla psána s ohledem na CUDA. Při zpracování vláken také nedochází k využití stejných proměnných a pokud ano, tak v nepředvídatelném způsobu, takže není příliš způsobů, jak by šlo využít sdílenou paměť nebo cache.

Významnou úpravou oproti CPU verzi je větší využívání lokalních proměnných. Všechna data z globální paměti, která jsou využívána alespoň 2x jsou uložena do lokálních proměnných, aby se předešlo zbytečnému čtení z globální paměti.

V. Testování

V rámci testování jsem připravil 5 různých scén, které všechny budou renderovány jak na CPU, tak na GPU. Kromě tohojak na GPU tak na CPU bude renderování probíhat bez KD stromu, s KD stromem a základním procházením a s KD stromem a pokročilým procházením s použitím sousedů.

A. Scény

*a) "2*2*4":* Tato scéna je jedna z jednodušších scén. Obsahuje 16 koulí uspořádaných v hranolu o rozměru 2*2*4.

b) "Insane": Tato scéna je vlastně modifikací klasické Cornell Box. Všechny objekty jsou značně reflektivní, což v kombinaci s až 15 odrazy nejspíš dobře vysvětluje důvod onoho názvu.

c) "Purple Wall": Scéna obsahuje čtvercovou desku složenou z 32*32 malých čtverců, kde každý čtverec je tvořen dvěmi trojúhelníky (modrým a červeným).

d) "Lens": Velmi jednoduchá scéna skládaící se ze 3 koulí (modrá, červená a průsvitná s vyšším indexem lomu)

e) "Ball-o-calypse": Scéna obsahující velmi reflektivní koule v krychlovém patternu o rozměrech 20*20*20

B. Očekávání

*f) "2*2*4":* Scéna je poměrně malá, ale použití KD stromů by se na ní mělo znatelně projevit. Koule jsou rozprostřeny tak, že by mělo dojít k dobrému rozdělení do KD stromu. U GPU varianty lze očekávat, že zde bude vidět nějaký overhead spojený s kopírováním dat

g) "Insane": Kamera je zcela uzavřena ve velmi reflektivní krychli. U této scény očekávám, že KD strom nezpůsobí veliké zrychlení, pokud vůbec k nějakému dojde. Vzhledem k velikému množství odrazů a k tomu, že všechny paprsky se budou odrážet až do maximální hloubky (nemohou uniknout), lze očekávat výrazně rychlejší renderování na GPU než na CPU

h) "Purple Wall": V této scéně by mělo docházet k velmi velikému urychlení pomocí KD stromů, neboť jednotlivé plošky lze dobře dělit. Jelikož tato scéna nemá prakticky žádnou hloubku, procházení pomocí sousedů by se nemělo lišit od obyčejného.

i) "Lens": Scéna je velmi primitivní a jde především o zjištění, jak je zhruba veliký overhead spojený s kopírováním dat.

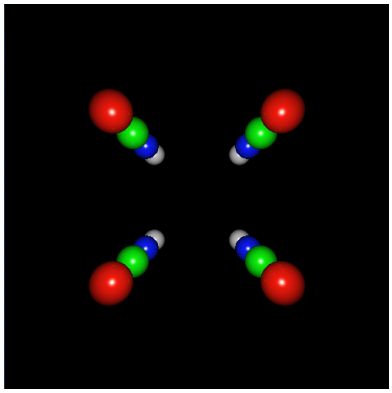
j) "Ball-o-calypse": Vše je velmi reflektivní a scéna je velmi rozsáhlá. KD stromy by ji měly značně urychlit a procházení s pomocí sousedů by zde mělo být poměrně dobře znát.

C. Parametry testů

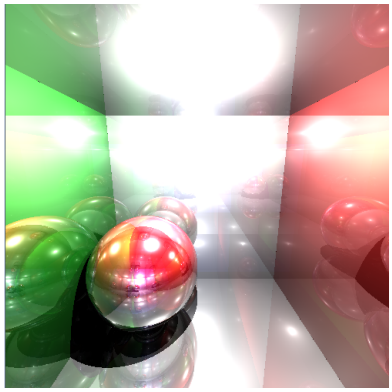
resolution	500*500
Kd tree max depth	10
ray max depth	15
grid size	1*1
block size	512*1

D. Parametry stroje

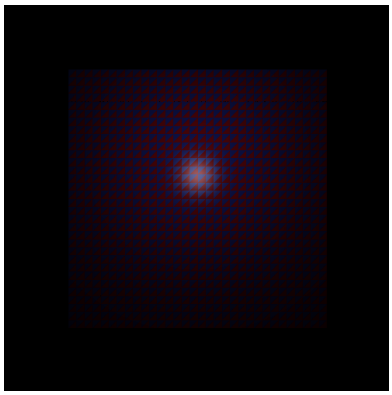
MS Windows 7 Pro 64-bit
Intel Core i7-2670QM 2.2GHz
8GB RAM
NVIDIA GeForce GT 540M
Compute Capability 2.0
CUDA 7.5



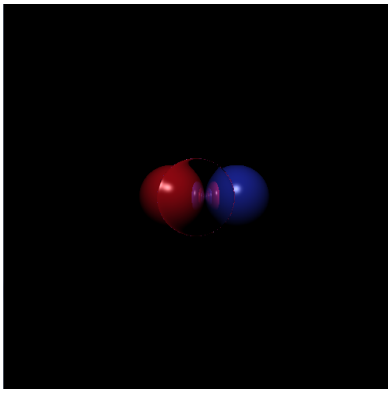
Obrázek 1. Scene 2*2*4



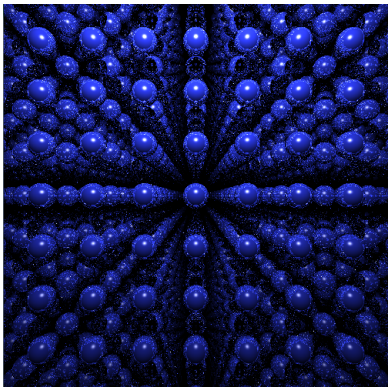
Obrázek 2. Scene Insane



Obrázek 3. Scene Purple Wall



Obrázek 4. Scene Lens



Obrázek 5. Scene Ball-o-calypse

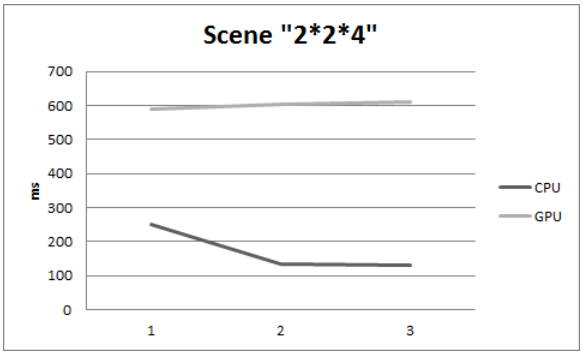
VI. Měření

A. *Parametry testů*

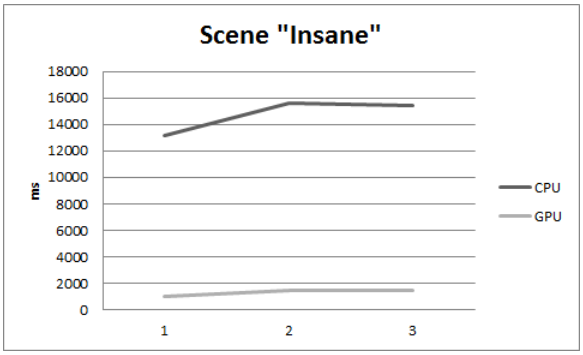
dev	trav	2*2*4	Insane	Purple	Lens	Ball-o-cal
C	NO	252	13 138	74 844	188	XXX
C	BAS	133	15 593	251	170	10 910
C	NEI	131	15 443	250	175	10 760
G	NO	590	1 010	2660	610	YYY
G	BAS	603	1 480	619	610	4 730
G	NEI	612	1 483	620	607	4 470

B. *Legenda*

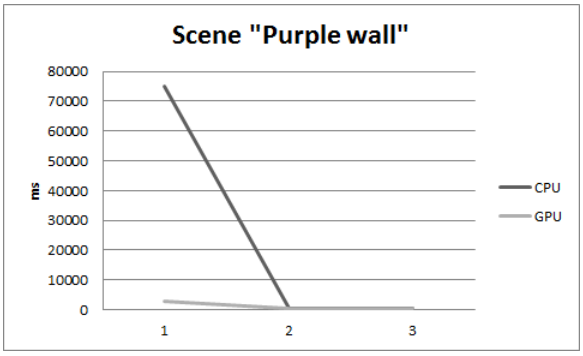
C	CPU
G	GPU
NO	no tree
BAS	basic tree traversing
NEI	neighborhood traversing
XXX	po 12 min jsem rendering ukončil
YYY	více než 15s GPU vláknům nepovolím



Obrázek 6. Graf 1
(1 - no tree, 2 - basic traversal, 3 - neighbors)



Obrázek 7. Graf 2
(1 - no tree, 2 - basic traversal, 3 - neighbors)

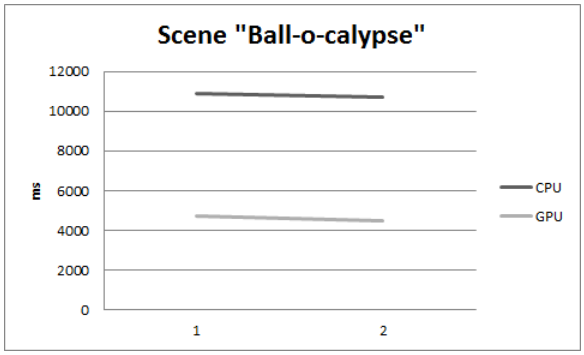


Obrázek 8. Graf 3
(1 - no tree, 2 - basic traversal, 3 - neighbors)

Výsledky některých měření byly docela překvapivé. Overhead spojený s použitím GPU, který byl vidět u malých scén, byl poměrně hodně veliký. Čekal jsem, že tam nějaký bude, ale osobně jsem si nemyslel, že i u malých scén bude dosahovat více než půl vteřiny.

Velmi mě překvapila scéna "Insane", kde použití KD stromů renderování prodloužilo a především na GPU, kde došlo až ke zpomalení o 50% času. Osobně jsem odhadoval, že se s KD stromem udrží někde poblíž naivnímu renderování, nebo budou výsledky rychlejí.

Obrázek 9. Graf 4
(1 - no tree, 2 - basic traversal, 3 - neighbors)



Obrázek 10. Graf 5
(1 - basic traversal, 2 - neighbors)

Zde se podle mě dobře projevuje fakt, že stavění KD stromu je poměrně hodně naivní a s lepším algoritmem by mohlo být dosaženo mnohem lepších výsledků.

Původně jsem očekával, že použití procházení KD stromu za použití sousedů mohlo urychlit procházení stromu více, než jak tomu je. Jelikož ale maximální hloubka KD stromu je 10 a při procházení se neprovádějí žádné příliš složité operace, je výsledek pochopitelný. V případech, kde je procházení pomocí sousedů pomalejší, dochází ke zpomalení v řádech desítek milisekund, což je ve většině případů způsobeno rozptylem časů, které vycházely.

VII. Závěr

Výsledky měření ukázaly, že použití KD stromu dokáže značně urychlit scény, které se skládají z jednotlivých objektů umístěných v prostoru.

V případech, kdy se objekty ve scéně hodně překrývají/protínají může špatné stavění KD stromu způsobit zbytečný overhead, který může výrazně

zpomalit renderování. To by ale bylo možné odstranit lepší metodou stavění Kd stromu.

Obecně lze říci, že scény renderované na GPU byly dokončeny rychleji mnohem rychleji, než při renderování na CPU. Vyjímkou jsou pouze velmi malé scény, kde overhead spojený s prací na GPU výrazně překročil čas renderování.

Vylepšením, které by bylo možné provést u GPU varianty je přidávání reflected paprsků do ray bufferu, místo jejich "rekurzivního"vypočítání. Vzhledem k známému maximálnímu množství odrazů a velikosti bloku lze odhadnout jak veliký bude muset být buffer.

Další vylepšením by mohlo být vkládání paprsků do bufferu tak, že bude využito prostorové lokality. tomto případě by nedocházelo tak často k tomu, že by některé paprsky skončily v hloubce 0 a některé šly až do max hloubky a musely na sebe čekat. (Momentálně dochází k renderování po řádcích a je zde vysoká pravděpodobnost, že alespoň část vláken čeká a nepracuje.)

Reference

[1] HAPALA, Michal; HAVRAN, Vlastimil. Review: Kd-tree Traversal Algorithms for Ray Tracing. In: Computer Graphics Forum. Blackwell Publishing Ltd, 2011. p. 199-213.

[2] POPOV, Stefan, et al. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. In: Computer Graphics Forum. Blackwell Publishing Ltd, 2007. p. 415-424.

[3] FOLEY, Tim; SUGERMAN, Jeremy. KD-tree acceleration structures for a GPU raytracer. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. ACM, 2005. p. 15-22.