Hardware pro počítačovou grafiku NPGR019

Praktický realtime raytracing

Jan Horáček

http://cgg.mff.cuni.cz/
MFF UK Praha

2012

Obsah

1. Úvod
   - Raytracing
2. OptiX
   - Přehled systému OptiX
   - OptiX programy
   - Akcelerační struktury
   - Struktura OptiX dat
   - Příklady
3. CPU raytracing
4. Literatura

Raytracing

- metoda zobrazování syntetických scén
- **1968** - **Arthur Appel** poprvé navrhl zobrazování pevných těles metodou sledování paprsků
- **1980** - **Turner Whitted** přidal myšlenku rekurze pro refraktivní a reflektivní tělesa
- další přidávání přesnosti pomocí hloubky ostrosti (depth of field), měkké stíny, rozmazání pohybem, . . .
- základní myšlenka je simulace šíření fotonů
  - dráha fotonu ze světla
  - zpětné hledání od kamery, odkud foton vyletěl
  - obousměrné metody

Raytracing 2

- všechny metody mají společnou vlastnost: *protínání paprsku s množinou 3D povrchů*
- v místě *nárazu* do scény se rozhoduje, co se s paprskem děje dál
  - odraz, lom, absorbce, ...
  - o tom rozhodují (optické) vlastnosti zasaženého povrchu
- při použití vhodné akcelerační struktury - sublineární složitost vzhledem k velikosti scény

Porovnání s rasterizací

- výhody
  - obecný model kamery
  - odrazy a lom světla se může počítat přesně a s minimem kódu
  - jednoduchý výpočet přesných ostrých i měkkých stínů
  - může se využít jako preprocessing nebo rendering pass v kombinaci s rasterizací (lightmaps, AO, přesné odrazy jako postprocess, . . .)
  - logaritmické akcelerační struktury
- nevýhody
  - náročná stavba kvalitní akcelerační struktury
  - horší podpora dynamických scén - přestavba není úplně triviální
  - dlouho nebyla adekvátní HW podpora
  - výkonný raytracer má velmi komplikovaný kód

Obsah

1 Úvod

2 OptiX

3 CPU raytracing

4 Literatura

Raytracing na GPU

- GPU jsou výborná ve zpracování datově paralelních úloh
- raytracing je masivně paralelizovatelná úloha
- bohužel raytracing může být velmi nevyvážený
- není jednoduché plně využít výkon GPU
- nVidia uvedla systém **OptiX**, který umožňuje "běžnému uživateli" bez PhD v počítačové grafice vytvořit si vlastní *rychlý* raytracer

OptiX

- OptiX **není** raytracer
- OptiX **je** framework pro stavbu aplikací využívajících raytracing, nezávislý na konkrétní metodě
- postaven na architektuře CUDA
- velká část programovatelná
- v CG například na klasický Whitted raytracing, path tracing, photon mapping, . . .
- vhodný nejen pro CG, ale i pro detekci kolizí, zjišťování viditelnosti, simulaci šíření zvuku, odhad objemu komplikovaného objektu, atp.

OptiX - pokračování

- abstraktní model obecného raytraceru
- postaven tak, aby škáloval výkon na budoucích výkonnějších GPU
- podobný typ abstrakce, jaký poskytuje OpenGL a DirectX pro rasterizační vykreslování
- poskytuje mechanismy (kde to je možné) pro spouštění uživatelského CUDA C kódu
    - shading včetně rekurzivnich paprsků
    - model kamery, generování paprsků
    - reprezentace informací v datové struktuře na paprsku
    - protnutí s libovolným tělesem (např. přesná koule bez teselace)
    - ...

Programy

- 8 druhů programovatelných komponent, jsou spouštěny na GPU
- **Ray Generation** - vstupní bod každého raytraceru, spuštěn paralelně pro každý pixel/vzorek, *střílí* primární paprsek(paprsky) a ukládá výsledek do výstupního bufferu
- **Exception** - vyvolán při přetečení zásobníku a dalších chybách
- **Closest Hit** - pokaždé, když se zjistí **nejbližší** průsečík se scénou, primárně využíván na shading
- **Any Hit** - volán při **každém** potenciálním nejbližším průsečíku, využitelné např. na stínovací paprsek
- **Intersection** - implementuje výpočet průsečíku paprsku s primitivem scény, volán při průchodu akcel. strukturou

Programy 2

- **Bounding Box** - počítá obalové těleso primitiva, volán při stavbě akcel. struktury
- **Miss** - pokud paprsek mine všechny objekty ve scéně - např. vrací barvu pozadí nebo vzorek z environment mapy
- **Visit** - při průchodu Selector node na zjištění, kterým potomkem se paprsek vydá
- vstupním jazykem je PTX
- OptiX SDK poskytuje sadu nástrojů a tříd pro využití CUDA C a nVidia C Compiler (nvcc)
  - API k dispozici v C i C++

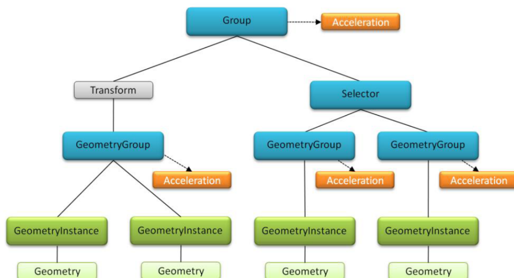
Proměnné

- OptiX obsahuje systém proměnných pro předávání dat do programů
- program referencuje proměnnou přes přesně definovanou sadu pravidel
    - proměnná barva může být navázána na *Material*
    - ovšem specifické instance geometrie tuto proměnnou mohou předefinovat

Spouštění

- všechny potřebné informace a data musí být v pořádku zkombinovány do kontextu
- určí se dimenze a velikost spuštění
- spustí se daný program pro generování paprsků, jednou pro každý element (pixel)
- výsledky se uloží do výstupního bufferu

Struktura OptiX dat



Typy akceleračních struktur

- **Bvh**
  - klasický bounding volume hierarchy
  - podporuje refitting pro rychý inkrementální update
  - vhodný pro akcelerační struktury nad skupinami
- **Sbvh**
  - Split-BVH je vysoce kvalitní BVH
  - více paměti a delší časy na stavbu
  - ideální pro statické scény
- **MedianBvh**
  - rychlá stavba
  - nižší kvalita
  - ideální pro dynamické scény

Typy akceleračních struktur 2

- **Lbvh**
  - **velmi** rychlá stavba
  - typicky rychlejší stavba než MedianBVH, ale nižší výkon pro raytracing
- **TriangleKdTree**
  - vysoce kvalitní kd-tree
  - podobný výkon jako Sbvh, ale může být náročnější na stavbu
- **NoAccel**
  - lineární průchod všemi prvky
  - výkonnější než ostatní jen na velmi jednoduchých scénách
  - hodí se na jednoduché skupiny scény

Příklad 1 - normal shader

- nejběžnějším programem **closest hit** program
- zavolán při nalezení nejbližšího protnutí paprsku primitivem

Normal shader

```
RT_PROGRAM void closest_hit_radiance0()
{
   prd_radiance.result = normalize(rtTransformNormal(
                                    RT_OBJECT_TO_WORLD,
                                    shading_normal))
                                    * 0.5f + 0.5f;
}
```

Příklad 1 - proměnné

- proměnná shading_normal musí být deklarována a její hodnota je zapsána v intersect programu

Deklarace proměnné

```
rtDeclareVariable(float3, shading_normal,
                  attribute shading_normal, );
```

- výsledek zapsán do uživatelsky def. struktury

Per Ray Data

```
struct PerRayData_radiance {
   float3 result;
   float  importance;
   int    depth;
};
rtDeclareVariable(PerRayData_radiance, prd_radiance, rtPayload, );
```

Příklad 1 - miss program

- pokud paprsek neprotne žádné těleso, volá se speciální program zvaný **miss** program
- barva pozadí nebo environment mapa, atd.
- v tomto případě vrací hodnotu `bg_color`, která je nastavena hostitelským programem

Miss program

```
rtDeclareVariable(float3, bg_color, , );

RT_PROGRAM void miss()
{
   prd_radiance.result = bg_color;
}
```

Příklad 1 - ray generation program

Ray generation program

```
RT_PROGRAM void pinhole_camera() {
   uint2 screen = output_buffer.size();
   float2 d = make_float2(launch_index) / make_float2(screen)
             * 2.f - 1.f;
   float3 ray_origin = eye;
   float3 ray_direction = normalize(d.x*U + d.y*V + W);
   optix::Ray ray(ray_origin, ray_direction,
                  radiance_ray_type, scene_epsilon);
   PerRayData_radiance prd;
   prd.importance = 1.f;
   prd.depth = 0;

   rtTrace(top_object, ray, prd);
   output_buffer[launch_index] = make_color(prd.result);
}
```

Příklad 1 - miss program

- nejdůležitejší je volání `rtTrace`, parametry:
  - `top_object` - kořen hierarchie scény, vytvořeno před spuštěním raytraceru
  - `ray` - paprsek pro výpočet daného pixelu
  - `prd` - reference na strukturu dat pro výpočet paprsku
- ve chvíli, kdy je paprsek spočítán (*hit* nebo *miss* program), vrací se zpět do ray generation a výsledek se ukládá do výstupního bufferu
- *pozn.*: protože se provádí rekurze, je třeba nastavit hloubku zásobníku, pokud přeteče, volá se *exception program*

Příklad 1

Normal shader example

Příklad 2 - diffuse shader

- změna pouze v *closest hit* programu
- geometric a shading normal se mohou lišit (např. kvůli bumpmappingu), obě získány z *intersection* programu
- výpočet ambientního a difuzního světla

Diffuse shader - část 1

```
RT_PROGRAM void closest_hit_radiance1()
{
   float3 world_geo_normal = normalize(
                            rtTransformNormal(
                            RT_OBJECT_TO_WORLD,
                            geometric_normal));
   float3 world_shade_normal = normalize(
                              rtTransformNormal(
                              RT_OBJECT_TO_WORLD,
                              shading_normal)) ;
```

Příklad 2 - diffuse shader 2

Diffuse shader - část 2

```
   float3 ffnormal = faceforward(world_shade_normal,
                                -ray.direction,
                                world_geo_normal);
   float3 color = Ka * ambient_light_color;
   float3 hit_point = ray.origin + t_hit * ray.direction;
   for(int i = 0; i < lights.size(); ++i) {
      BasicLight light = lights[i];
      float3 L = normalize(light.pos - hit_point);
      float nDl = dot(ffnormal, L);
      if( nDl > 0 )
         color += Kd * nDl * light.color;
   }
   prd_radiance.result = color;
}
```

Příklad 2 - světlo

- informace o světlech uloženy v jednorozměrném vstupním bufferu
- uživatelsky definovaná struktura
- data nastavena hostitelskou aplikací před spuštěním raytraceru
- difuzní odrazivost tělesa v proměnné Kd (nastavovaná například per material)

Diffuse shader - světla

```
struct BasicLight
{
   float3 pos;
   float3 color;
   int    casts_shadow;
}
rtBuffer<BasicLight> lights;
```

Příklad 2

Diffuse shader

Příklad 3 - Phongův světelný model

- k Lambertovskému stínování přidáme **odlesk**
- použijeme metodu Jima **Blinn**a pro výpočet **halfway** vektoru

Phong shader

```
...
   if( nDl > 0) {
      float Lc = light.color;
      color += Kd * nDl * Lc;

      float3 H = normalize(L - ray.direction);
      float nDh = dot( ffnormal, H );
      if( nDh > 0)
         color += Ks * Lc * pow(nDh, phong_exp);
   }
...
```

Příklad 3

Phong shader

Příklad 4 - stíny

- výpočet přesných vržených stínů při bodovém zdroji světla
- je třeba zkonstruovat paprsek mezi aktuálním bodem a zdrojem světla a zjistit, jestli protíná geometrii scény

Stíny

```
...
   if( nDl > 0 ) {
      PerRayData_shadow shadow_prd;
      shadow_prd.attenuation = 1.0f;
      float Ldist = length(light.pos - hit_point);
      optix::Ray shadow_ray( hit_point, L, shadow_ray_type,
                                       scene_epsilon, Ldist );
      rtTrace(top_shadower, shadow_ray, shadow_prd);
      float light_attenuation = shadow_prd.attenuation;
```

Příklad 4 - stíny 2

Shadows

```
      if( light_attenuation > 0.0f ) {
         float Lc = light.color * light_attenuation;
         color += Kd * nDl * Lc;

         float3 H = normalize(L - ray.direction);
         float nDh = dot( ffnormal, H );
         if( nDh > 0)
            color += Ks * Lc * pow(nDh, phong_exp);
      }
   }
...
```

Příklad 4 - stíny

- nepotřebujeme nejbližší objekt, stačí jakýkoliv, tzn. použijeme **any hit** program
- **any hit** je zavolán pro *každý* průsečík paprsku s primitivem, není garantováno pořadí
- nám stačí testování zastavit při jakémkoliv úspěšném testu
- pro stínový paprsek se využívá jiný **typ** paprsku - to znamená pouze jiné **ID** a jinak nastavené programy

Stínový paprsek

```
RT_PROGRAM void any_hit_shadow()
{
   prd_shadow.attenuation = 0.0f;
   rtTerminateRay();
}
```

Příklad 4

Přesný stín bodového zdroje světla

Příklad 5 - odrazy

- zrcadlový odraz velmi jednoduchý
- **closest hit** se změní tak, aby vyslal jeden odražený paprsek
- jeho barva se zkombinuje s barvou povrchu

Zrcadlový odraz

```
RT_PROGRAM void floor_closest_hit_radiance4() {
   // Phong shading
   ...
   float3 R = reflect( ray.direction, ffnormal );
   optix::Ray refl_ray( hit_point, R, radiance_ray_type,
                        scene_epsilon );
   rtTrace(top_object, refl_ray, refl_prd);
   color += reflectivity * refl_prd.result;
   prd_radiance.result = color;
}
```

Příklad 5 - odrazy s kontrolou hloubky rekurze

- předchozí kód snadno nechá přetéct zásobník
- je třeba přidat kontrolu hloubky
- další možnost kontroly je sledovat minimální *příspěvek* paprsku pro finální barvu

Zrcadlový odraz

```
...
if( prd_radiance.depth < max_depth ) {
   refl_prd.depth = prd.radiance.depth + 1;
   float3 R = reflect( ray.direction, ffnormal );
   optix::Ray refl_ray( hit_point, R, radiance_ray_type,
                        scene_epsilon );
   rtTrace(top_object, refl_ray, refl_prd);
   color += reflectivity * refl_prd.result;
}
prd_radiance.result = color;
...
```

Příklad 5

Zrcadlové odrazy

Příklad 6 - environment mapping

- přidáme jednoduchý efekt jiného pozadí než jednoduchá barva
- načítání barvy z textury
- změny kódu tentokrát v **miss** programu

Environment mapping

```
rtTextureSampler<float4, 2> envmap;
RT_PROGRAM void envmap_miss() {
   float theta = atan2f( ray.direction.x, ray.direction.z );
   float phi   = M_PIf * 0.5f - acosf( ray.direction.y );
   float u     = ( theta + M_PIf ) * ( 0.5f * M_1_PIf );
   float v     = 0.5f * ( 1.0f + sin(phi) );
   prd_radiance.result = make_float3( tex2D( envmap, u, v ) );
}
```

Příklad 6

Environment mapping

Příklad 7 - procedurální geometrie

- OptiX je navržen tak, aby se jednoduše daly přidávat nová primitiva
  - například koule, konvexní obal, . . .
- **intersect** program počítá průsečík s paprskem

Konvexní obal - 1.část

```
rtBuffer<float4> planes;
RT_PROGRAM void convhull_intersect(int primIdx) {
   int n = planes.size();
   float t0 = -FLT_MAX;
   float t1 = FLT_MAX;
   float3 t0_normal = make_float3(0);
   float3 t1_normal = make_float3(0);
   for( int i = 0; i < n && t0 < t1; ++i ) {
      float4 plane = planes[i];
      float3 n = make_float3(plane);
      float  d = plane.w;
```

Příklad 7 - pokračování konvexního obalu

Konvexní obal = 2.část

```
      float denom = dot( n, ray.direction );
      float t = -( d + dot( n, ray.origin )) / denom;
      if( denom < 0 ) { // enter
         if( t > t0 ) {
            t0 = t;
            t0_normal = n;
         }
      } else { // exit
         if( t < t1 ) {
            t1 = t;
            t1_normal = n;
         }
      }
   }
}
```

Příklad 7 - dokončení konvexního obalu

Konvexní obal = 3.část

```
if( t0 > t1 )
   return;
if( rtPotentialIntersection(t0) ) {
   shading_normal = geometric_normal = t0_normal;
   rtReportIntersection(0);
} else if( rtPotentialIntersection(t1) ) {
   shading_normal = geometric_normal = t1_normal;
   rtReportIntersection(0);
}
```

Příklad 7 - AABB

- je třeba dodat i vlastní výpočet **bounding box**
  - např. pro stavbu akcelerační struktury
- pro zjednodušení je zde počítá hostitelský program a pouze pošle v proměnné výsledný AABB

Výpočet AABB

```
rtDeclareVariable(float3, convhull_bbmin, , );
rtDeclareVariable(float3, convhull_bbmax, , );
RT_PROGRAM void convhull_bounds(int primIdx, float result[6])
{
   optix::Aabb *aabb = (optix::Aabb*) result;
   aabb->m_min = convhull_bbmin;
   aabb->m_max = convhull_bbmax;
}
```

Příklad 7

Skleněný konvexní obal

Obsah

1 Úvod

2 OptiX

3 CPU raytracing

4 Literatura

CPU raytracing

- ukazuje se, že GPU má sice velmi velkou výhodu v masivně paralelním zpracování paprsků, ale u složitějších scén a nekoherentních paprscích trpí více, než CPU implementace
- vývoj probíhá i na CPU
- moderní CPU raytracery schopny dosahovat interaktivních rychlostí
- například: **RTfact**
  - raytracer vyvíjený primárně na univerzitě v Saarbrückenu
  - heavily-templated SIMD kód, **ray packeting**
  - co jde se počítá již při kompilaci
  - v současnosti provádíme měření rychlosti pro porovnání s GPU implementací OptiX

Intel Many Integrated Core (MIC) Architecture

- **Larabee**
  - první pokus Intelu o mnoho procesorů na jednom čipu
  - nakonec oficiálně zrušeno
- **Knights Ferry**
  - pokračování vývoje mnoha CPU jader na jednom čipu
  - předveden model se 32 x86 jádry, každé zvládne až 4 vlákna pomocí HT
  - **standardní x86 programování**
- **Knights Corner**
  - finální verze
  - měla by být postavena na 22nm architektuře
  - více než 50 jader na čipu

Knights Ferry

- 32 jader x86
- 32KB L1 instrukční cache, 32KB L1 datové cache a 256KB L2 cache
- vektorová jednotka
  - v principu velmi široká (512bit) SSE jednotka
  - 16 single-precision operací v jedné instrukci
- multithreading - 4 vlákna na procesor, střídají se během L1 cache miss
- implementováno na PCI kartě s vlastní pamětí
- oficiálně *Intel Co-Processor Architecture*
- živě demonstrováno na raytracingu hry Wolfenstein - výpočet v cloudu, streamováno na notebook

Obsah

1 Úvod

2 OptiX

3 CPU raytracing

4 Literatura

Literatura

- nVidia Corporation: **Optix Programming Guide**
- nVidia Corporation: **Optix Quickstart Guide**
- http://developer.nvidia.com