Welcome!

Lecture 11 - “GPU Ray Tracing (1)”

\[ I(x, x') = g(x, x') \left[ \epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right] \]
Today’s Agenda:

- Exam Questions: Sampler
- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective
Exam Questions

We use the Surface Area Heuristic to determine a good position for a split plane during BVH construction.

a) One version of the SAH looks as follows:

\[ C_{\text{split}} = C_T + A_{\text{left}} N_{\text{left}} C_I + A_{\text{right}} N_{\text{right}} C_I \]

What are \( C_T \) and \( C_I \) for?

How would you modify this formula if your BVH supports spheres and tori?

b) Explain why we use surface area (rather than e.g. bounding box volume) in the cost function.

c) The Surface Area Heuristic is a ‘greedy’ heuristic. What is the meaning of ‘greedy’ in this context?

d) What is the algorithmic complexity of the greedy SAH-guided BVH construction algorithm (without binning), and what would be the algorithmic complexity of the non-greedy version?
Exam Questions

Behold the Rendering Equation:

\[ L_o(x, \omega_o) = L_E(x, \omega_o) + \int_{\Omega} f_r(x, \omega_o, \omega_i) L_i(x, \omega_i) \cos \theta_i \ d\omega_i \]

a) What does \( \cos \theta_i \) do?

b) Why is \( \cos \theta_i \) not included in the BRDF?

c) Is the above formulation missing the visibility factor?

d) Another formulation of the RE is the three-point formulation:

\[ L(s \leftarrow x) = L_E(s \leftarrow x) + \int_A f_r(s \leftarrow x \leftarrow x') L(x \leftarrow x') G(x \leftrightarrow x') dA(x') \]

What is \( A \) in this equation?

Write out \( G(x \leftrightarrow x') \).
A scene is illuminated by a single double-sided square light source.

Two algorithms are used to sample the light source: the first picks a random point on a random side of the light source, while the second algorithm only picks random points on the side of the light source facing point $p$.

a) Write down the Monte-Carlo integrator that estimates the illumination on point $p$ using the first algorithm.

b) Write down the Monte-Carlo integrator that estimates the illumination on point $p$ using the second algorithm.

Note: both methods should obviously produce the same answer, on average.
Today's Agenda:

- Exam Questions: Sampler
- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective
Introduction

Transferring Ray Tracing to the GPU

Platform characteristics:

- Massively parallel
- SIMT
- High bandwidth
- Massive compute potential
- Slow connection to host

Consequences:

- Thread state must be small
- Efficiency requires coherent control flow
Introduction

Transferring Ray Tracing to the GPU

Survey

- Understand evolution of graphics hardware
- Understand characteristics of modern GPUs
- Investigate algorithms designed with these characteristics in mind
Today’s Agenda:

- Exam Questions: Sampler
- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective
Ray Tracing on Programmable Graphics Hardware*

Graphics hardware in 2002:

- Vertex and fragment shaders only
- Simple instruction sets
- Integer-only (fixed-point) fragment shaders
- Limited number of instructions per program
- Limited number of inputs and outputs
- No loops, no conditional branching

Expectations:

- Floating point fragment shaders
- Improved instruction sets
- Multiple outputs per fragment shader

*: Ray tracing on programmable graphics hardware, Purcell et al., 2002.
Survey

Ray Tracing on Programmable Graphics Hardware

Challenge: to map ray tracing to stream computing.

Stage 1: Produce a stream of primary rays.

Stage 2: For each ray in the stream, find a voxel containing geometry.

Stage 3: For each voxel in the stream, intersect the ray with the primitives in the voxel.

Stage 4: For each intersection point in the stream, apply shading and produce a new ray.
Ray Tracing on Programmable Graphics Hardware

Stream computing without flow control:

**Assign a state to each ray.**

1. Traversing;
2. intersecting;
3. shading;
4. done.

Now, for each program render a quad using a stencil based on the state; this enables the program only for rays in that state*.

Survey

Ray Tracing on Programmable Graphics Hardware

Stream computing without flow control:

- Generate Eye Rays
- Traverse Accstruc
- Intersect Prims
- Shade and Generate Shadow Rays

Use stencil to select functionality

Render two triangles, shader performs ray tracing
Ray Tracing on Programmable Graphics Hardware

Acceleration structure (grid) traversal:

1. setup traversal;
2. one step using 3D-DDA*.

Note that *each step* through the grid requires one pass.
Survey

2002

Ray Tracing on Programmable Graphics Hardware

Results

passes  efficiency

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2443</td>
<td>0.009</td>
<td>1198</td>
<td>0.061</td>
<td>1999</td>
</tr>
<tr>
<td>2835</td>
<td>0.062</td>
<td>1085</td>
<td>0.105</td>
<td></td>
</tr>
</tbody>
</table>
Ray Tracing on Programmable Graphics Hardware

Conclusions

- Ray tracing can be done on a GPU
- GPU outperforms CPU by a factor 3x (for triangle intersection only)
- Flow control is needed to make the full ray tracer efficient.
KD-Tree Acceleration Structures for a GPU Raytracer*

Observations on previous work:

- Grid only: doesn't adapt to local scene complexity
- kD-tree traversal can be done on the GPU, but the stack is a problem.

Goal:

- Implement kD-tree traversal without stack.

---

* KD-Tree Acceleration Structures for a GPU Raytracer, Foley & Sugerman, 2005

Survey 2005

KD-Tree Acceleration Structures for a GPU Raytracer

Observations on previous work:

- Grid only: doesn't adapt to local scene complexity
- kD-tree traversal can be done on the GPU, but the stack is a problem.

Goal:

- Implement kD-tree traversal without stack.

*: KD-Tree Acceleration Structures for a GPU Raytracer, Foley & Sugerman, 2005
KD-Tree Acceleration Structures for a GPU Raytracer

Recall standard kD-tree traversal:

Setup:
1. $t_{\text{max}}, t_{\text{min}} = \text{intersect}(\text{ray, root bounds})$

Root node:
2. Find intersection $t$ with split plane
3. If $t_{\text{min}} \leq t \leq t_{\text{max}}$:
   - Process near child with segment $(t_{\text{min}}, t)$
   - **Process far child with segment $(t, t_{\text{max}})$**
4. else if $t > t_{\text{max}}$:
   - Process left child with segment $(t_{\text{min}},t_{\text{max}})$
5. else
   - Process right child with segment $(t_{\text{min}},t_{\text{max}})$
KD-Tree Acceleration Structures for a GPU Raytracer

Recall standard kD-tree traversal:

Setup:
1. $t_{\text{max}}, t_{\text{min}} = \text{intersect}(\text{ray}, \text{root bounds})$;

Root node:
2. Find intersection $t$ with split plane
3. If $t_{\text{min}} \leq t \leq t_{\text{max}}$:
   - **Push far child**
   - **Continue with near child**
4. else if $t > t_{\text{max}}$:
   - Process left child with segment ($t_{\text{min}}, t_{\text{max}}$)
5. else
   - Process right child with segment ($t_{\text{min}}, t_{\text{max}}$)
KD-Tree Acceleration Structures for a GPU Raytracer

Traversing the tree without a stack:

*If we always pick the nearest child, the only value that will change is \( \text{tmax} \).*

Setup:

1. \( \text{tmax}, \text{tmin} = \text{intersect( ray, root bounds )}; \)
2. Always pick the nearest child.
3. Once we have processed a leaf, restart with:
   - \( \text{tmin}=\text{tmax} \)
   - \( \text{tmax}=\text{intersect( ray, root bounds )} \)

This algorithm is referred to as *kd-restart.*

Note that the average ray intersects only a small number of leaves. Since restart only happens for each intersected leaf that didn't yield an intersection point, the expected cost is still \( O(\log n) \).
KD-Tree Acceleration Structures for a GPU Raytracer

We can reduce the cost of a restart by storing node bounds and a parent pointer with each node.

Instead of restarting at the root, we now restart at the first ancestor that has a non-empty intersection with \((t_{\text{min}}, t_{\text{max}})\).

This algorithm is referred to as \textit{kd-backtrack}.
KD-Tree Acceleration Structures for a GPU Raytracer

Implementation: each ray is assigned a state:

1. Initialize: finds $t_{\text{min}}, t_{\text{max}}$ for each ray in the input stream
2. Down: traverses each ray down by one step
3. Leaf: handles ray/leaf intersection for each ray
4. Intersect: performs actual ray/triangle intersection
5. Continue: decides whether each ray is done or needs to restart / backtrack
6. Up: performs one backtrack step for each ray in the input stream.

As before, the state is used to mask rays in the input stream when executing each of the 6 programs.
Survey

2005

KD-Tree Acceleration Structures for a GPU Raytracer

Results (ms*):

<table>
<thead>
<tr>
<th>Method</th>
<th>512x512</th>
<th>1024x1024</th>
<th>2048x2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>23</td>
<td>4620</td>
<td>4770</td>
</tr>
<tr>
<td>grid</td>
<td>63</td>
<td>357</td>
<td>8344</td>
</tr>
<tr>
<td>kd-restart</td>
<td>80</td>
<td>701</td>
<td>968</td>
</tr>
<tr>
<td>kd-backtrack</td>
<td>84</td>
<td>690</td>
<td>946</td>
</tr>
<tr>
<td>kd-backtrack</td>
<td>84</td>
<td>690</td>
<td>946</td>
</tr>
</tbody>
</table>

*: Hardware: 256MB ATI X800 XT PE (2004), rendering @ 512x512, time in milliseconds.
Survey

2007

Interactive k-d tree GPU raytracing*
Stackless KD-tree traversal for high performance GPU ray tracing**

Observations on previous work:

- GPU ray tracing performance can’t keep up with CPU
- Kd-restart requires substantially more node visits
- Kd-backtrack increases data storage and bandwidth
- Looping and branching wasn’t available, but is now.

*: Interactive k-d tree GPU raytracing, Horn et al., 2007
**: Stackless KD-tree traversal for high performance GPU ray tracing, Popov et al., 2007
Interactive k-d tree GPU raytracing
Stackless KD-tree traversal for high performance GPU ray tracing

Ray tracing with a short stack:

By keeping a fixed-size stack we can prevent a restart in almost all cases.
2007

kD-tree Traversal using Ropes*

“The main goal of any traversal algorithm is the efficient front-to-back enumeration of all leaf nodes pierced by a ray. From that point of view, any traversal of inner nodes of the tree (...) can be considered overhead that is only necessary to locate leafs quickly.”

Algorithm:

1. Traverse to a leaf;
2. If no intersection found:
   - Follow rope;
   - Goto 1.

*; Ray tracing with rope trees, Havran et al., 1998
Survey

Interactive k-d tree GPU ray tracing
Stackless KD-tree traversal for high performance GPU ray tracing

Ray tracing with flow control:

- 25x performance of the previous paper
- 1.65x – 2.3x from algorithmic improvements
- 3.75x from hardware advances

⇒ 2.9x from switching from multi-pass to single-pass.
Survey

Interactive k-d tree GPU raytracing
Stackless KD-tree traversal for high performance GPU ray tracing

Results*:

<table>
<thead>
<tr>
<th>GPU</th>
<th>12.7</th>
<th>10.6</th>
<th>36.0</th>
<th>16.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (1 core)</td>
<td>-</td>
<td>3.6</td>
<td>6.6</td>
<td>3.9</td>
</tr>
</tbody>
</table>

*: Hardware: GeForce 8800 GTX / Opteron @ 2.6 Ghz, performance in fps @ 1024x1024.
Interactive k-d tree GPU raytracing
Stackless KD-tree traversal for high performance GPU ray tracing

Conclusions

- Compared to kd-restart, approx. 1/3rd of the nodes is visited;
- The GPU now outperforms a quad-core CPU;
- NVidia GTX 8800 does 160 GFLOPS; cost per ray is 10.000 cycles...
Advanced Graphics – GPU Ray Tracing (1)

Survey

2007

Realtime Ray Tracing on GPU with BVH-based Packet Traversal*

Observations on previous work:

- kD-trees limit rendering to static scenes
- kD-trees with ropes are inefficient storage wise
- Popov et al.’s tracer achieves only 33% utilization due to register pressure
- Existing GPU ray tracers do not realize GPU potential
- Existing GPU ray tracers suffer from execution divergence.

Solution:

Use BVH instead of kD-tree.
Realtime Ray Tracing on GPU with BVH-based Packet Traversal

To achieve maximum utilization of a G80 GPU, we need 768 threads per multiprocessor (24 warps).

Each multiprocessor has 16Kb shared memory and 32Kb register space. ➡️ For 24 warps we have 5 words plus 10 registers per thread available.

An important difference between kD-tree packet traversal and BVH packet traversal is that kD-tree traversal requires a stack for the packet plus \((t_{\text{min}}, t_{\text{max}})\) per ray, while the BVH packet only requires a stack.
Realtime Ray Tracing on GPU with BVH-based Packet Traversal

GPU packet traversal for BVH:

1. A packet consists of 8x4 rays, handled by a single warp
2. The packet traverses the BVH using masked traversal (where t is used as mask)
3. Storage:
   1. Per ray: O, D, t (7 floats)
   2. Per packet: stack

Survey 2007
Observations:

This is hardly a packet traversal scheme; we are essentially traversing 32 independent rays.

However:

the rays in the packet *do* share a single stack.

Question:

will rays ever visit a node they didn’t have to visit? (i.e., do they visit a node they would not have visited using a stack per ray?)

Answer: yes they will. The weakness of this algorithm is in determining the near and far child. This is based on ‘the majority of rays’, and therefore an individual ray may visit nodes in a sub-optimal order. The paper does not address this issue.
Realtime Ray Tracing on GPU with BVH-based Packet Traversal

Results*:

<table>
<thead>
<tr>
<th></th>
<th>primary</th>
<th>shadow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>19.0</td>
<td>6.1</td>
</tr>
<tr>
<td></td>
<td>16.2</td>
<td>5.7</td>
</tr>
<tr>
<td></td>
<td>6.4</td>
<td>2.9</td>
</tr>
</tbody>
</table>

*: Hardware: GeForce 8800 GTX, rendering at 1024x1024, performance in fps.
Digest

Challenges in GPU ray tracing:

- Utilizing GPU compute potential (getting it to work ➔ beating CPU ➔ efficient)
- Mapping an embarrassingly parallel algorithm to a streaming processor
- Tiny per-thread state (balancing utilization / algorithmic efficiency)
- Freedom in the choice of acceleration structure
- Tracing divergent rays
Today's Agenda:

- Exam Questions: Sampler
- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective
5 Faces

Advanced Graphics

– GPU Ray Tracing (1)
Pragmatic GPU Ray Tracing*

Context:

- Real-time demo
- 50-100k triangles
- Fully dynamic scene
- Fully dynamic camera (no time to converge)
- Must “look good” (as opposed to “be correct”)

- Rasterize primary hit
- No BVH / kD-tree

- Use a grid (or better: sparse voxel octree / brickmap).

*: Real-time Ray Tracing Part 2 – Smash / Fairlight, Revision 2013
https://directtovideo.wordpress.com/2013/05/08/real-time-ray-tracing-part-2
2013

Pragmatic GPU Ray Tracing

Grid traversal: 3D-DDA

Brickmap traversal:

- build in linear time
- locate ray origins in constant time
- skip some open space
- little flow divergence in shader
- simple thread state
Pragmatic GPU Ray Tracing

Filling the grid: using rasterization hardware.
→ Determine which voxels a triangle overlaps.

Algorithm:

1. Determine for which plane \((xy, yz, xz)\) the triangle has the greatest projected area.
2. Rasterize to that face; use interpolated \(x, y\) and depth to determine voxel coordinate.
3. Use conservative rasterization*, **.

Pragmatic GPU Ray Tracing

In this case, we are not building a voxel set, but a grid with pointers to the original triangles.

- Add each triangle to a preallocated list per node.

From grid to brickmap:

- each brick consists of a small grid, e.g. 4x4x4.
- repeat the rasterization process at the higher resolution
- assign each triangle to cells in the fine grid.

Note that voxelization can be part of a rasterization-based rendering pipeline; it can e.g. be fed with triangles of a skinned mesh or even procedurally generated meshes.
Pragmatic GPU Ray Tracing

Pragmatic traversal:

- ‘Trace’ primary ray using rasterization
- Determine secondary ray origin from G-buffer

After this:

- Put a maximum on the number of traversal steps, regardless of bounce depth.
Pragmatic GPU Ray Tracing

Pragmatic diffraction:

Each ray represents 3 ‘wavelengths’, and each results in a different refracted direction. However, only the direction of the first ray is actually used to find the next intersection for the triplet.

EXCEPT: when the rays exit the scene and returns a skybox color; only then the three directions are used to fetch 3 skybox colors which are then blended.
Pragmatic depth of field:

Since primary rays are rasterized, the camera used is a pinhole camera.

Depth of field with bokeh is simulated using a postprocess.

See for a practical approach:

Pragmatic GPU Ray Tracing

Limitations:

- Doesn't work well for 'teapot in a stadium'
- Not suitable for very large scenes (area)
- Manual parameter tweaking

=> The method is not good for a general purpose ray tracer, but really clever for a special purpose renderer.

=> Performance is very good, although hard to estimate:
Demo runs @ 60fps on a high-end GPU;
Traces ~1M primary rays;
Most rays make several bounces (very divergent!);
Guestimate: ~250M rays per second for a fully dynamic scene.
5 Faces

Other Real-time Ray Tracing Demos

For a brief history, see these links:


Also check here: http://mpierce.pie2k.com/pages/108.php
Today's Agenda:

- Exam Questions: Sampler
- Introduction
- Survey: GPU Ray Tracing
- Practical Perspective
Next Time

Coming Soon in Advanced Graphics

GPU Ray Tracing Part 2:

- State of the art BVH traversal by Aila and Laine;
- Wavefront Path Tracing
- Heterogeneous Path Tracing: Brigade.
INFOMAGR – Advanced Graphics
Jacco Bikker - November 2017 - February 2018

END of “GPU Ray Tracing (1)”

next lecture: “GPU Path Tracing (2)”