Implementing a practical rendering system using GLSL

Toshiya Hachisuka

University of Tokyo

Tokyo Demo Fest 2015
Aim of this seminar

Sharing my experience of writing a practical rendering system on a GPU only with GLSL

Approx. 100M photon paths in 1 min @ GeForce GTX 680
Disclaimer

Not all of my comments in this seminar are fully validated by scientific experiments.

Take them with a grain of salt!
Why GLSL?

• Cross-platform (both OS and GPU)
• Battle-tested
• Easy to write
• Automatic support for multiple GPUs
Why GLSL?

• Cross-platform (both OS and GPU)
• Battle-tested
• Easy to write
• Automatic support for multiple GPUs
• **Fun**
Key features

• Bounding volume hierarchy (BVH)
  • Efficient ray tracing of lots of objects
  • Triangles only
• Stochastic progressive photon mapping (SPPM)
  • Physically accurate global illumination
  • Textures and basic materials
Bounding volume hierarchy

• In a practical system, we have lots of triangles

• Data structure to avoid touching every triangle
Bounding volume hierarchy

- In a practical system, we have lots of triangles
- Data structure to avoid touching every triangle
Bounding volume hierarchy

- In a practical system, we have **lots** of triangles
- Data structure to avoid touching every triangle
Bounding volume hierarchy

- In a practical system, we have **lots** of triangles
- Data structure to avoid touching every triangle
Bounding volume hierarchy

• In a practical system, we have **lots** of triangles

• Data structure to avoid touching every triangle
Stochastic PPM

- Global illumination algorithm developed by myself
- Consists of three steps
  - Photon tracing
  - Eye ray tracing
  - Density estimation

"Stochastic Progressive Photon Mapping" T. Hachisuka and H. W. Jensen
ACM Transactions on Graphics (SIGGRAPH Asia 2009)
Stochastic PPM
Stochastic PPM
Stochastic PPM

Photon tracing
Stochastic PPM
Stochastic PPM

Eye ray tracing
Stochastic PPM
Stochastic PPM

Density estimation
- Build BVH
- Photon tracing
- Eye ray tracing
- Density estimation

CPU

GPU
Design principles

• Make all the tasks in rendering to
  • Have a high degree of parallelism
  • Have a uniform workload distribution
  • Use no local memory

... so that they run efficiently on GPUs

• I did not aim for a “production-quality” system
Bounding Volume Hierarchy
Challenges

• Standard BVH traversal uses stack

• Stack is implemented via local memory on GPUs

• Contradicts with the design principles!
Challenges

• Standard BVH traversal uses stack
• Stack is implemented in local memory on GPUs

We want **stackless** traversal!

• Contradicts with the design principles!
Why stackless?

- Modern GPUs can do stack-based traversal [Aila 09]
  - Straightforward to implement
  - Efficient (due to dynamic traversal order)

Why bother implementing stackless traversal?
Why stackless?

Size of local memory can limit the parallelism

- Modern GPUs have around 32kB local memory
- Stack-based traversal consumes around 512 B

\[
\frac{32kB}{512 B} = 64 \text{ rays in parallel}
\]
Threaded BVH

- Precompute “hit” and “miss” links
  - Also known as skip pointers [Smits 98]
- Allows stackless traversal
- Order of traversal of child nodes is fixed
Threaded BVH
Threaded BVH

Hit links
Threaded BVH

Miss links

Terminal
Threaded BVH
Threaded BVH

Terminal
Threaded BVH

Don’t need to store the original tree!
Hit and miss links

• Hit links
  • Always the next node in the array

• Miss links
  • Internal, left: sibling node
  • Internal, right: parent’s sibling node (until it exists)
  • Leaf: same as hit links
Traversal

• Extremely **simple**! (no stack, no bitwise ops.)

```java
node = root;
while (node != null) {
    if (intersect(node.bonding, ray)) {
        if (node.leaf) {
            hit_point = intersect(node.triangles, ray);
        }
        node = node.hit;
    } else {
        node = node.miss;
    }
}
```
Challenges

• Traversal order is fixed

• Want to visit the closest node first
Challenges

- Traversal order is fixed
- Want to visit the closest node first
Challenges

• Traversal order is fixed

• Want to visit the closest node first
Challenges

- Traversal order is fixed
- Want to visit the closest node first
Challenges

• Traversal order is fixed

• Want to visit the closest node first
Challenges

• Traversal order is fixed

• Want to visit the closest node first
Challenges

• Traversal order is fixed

• Want to visit the closest node first
Multiple-threaded BVH (MTBVH)

- Prepare threaded BVHs for six major directions
  - $+X$ $-X$ $+Y$ $-Y$ $+Z$ $-Z$
- Need to add only “hit” and “miss” links
- Bounding boxes data is shared
- Classify ray directions via 1x1 cube maps
- Unpublished novel idea as far as I know :->
\[ \begin{array}{c}
\text{X} = 6.0 \\
1 \\
2 \\
3 \\
\text{X} = 2.5 \\
4 \\
5 \\
6 \\
\end{array} \]
six different directions
Terminal

six different directions
Data layout

- Put all six sets of hit and miss links into one texture
Data layout

- Threading (\(\text{vec4} \times 1\))
  - \(\text{vec4}(\text{hit.uv}, \text{miss.uv})\)
  - Store -1.0 to indicate the terminal

- AABB (\(\text{vec4} \times 2\))
  - \(\text{vec4}(\text{min.xyz}, \text{triangle.u}), \text{vec4}(\text{max.xyz}, \text{triangle.v})\)
  - Store -1.0 for w to indicate internal nodes
MTBVH traversal

- Still extremely **simple** (only one change)!

```cpp
node = cubemap(root_tex, ray.direction);
while (node != null) {
    if (intersect(node.bounding, ray)) {
        if (node.leaf) {
            hit_point = intersect(node.triangles, ray);
        }
        node = node.hit;
    } else {
        node = node.miss;
    }
}
```
Ray-triangle intersection

• There are many different approaches

• Best algorithm for CPUs is not the best for GPUs
  • Different computation/data transfer ratio and cost of conditional branches
  • Some “optimisation” can backfire!
  • Modified Möller-Trumbore algorithm works well
Ray-triangle intersection

```
vec3 p0 = V0;
vec3 e0 = V1 - V0;
vec3 e1 = V2 - V0;
vec3 pv = cross(ray.direction, e1);
float det = dot(e0, pv);
vec3 tv = ray.origin - p0;
vec3 qv = cross(tv, e0);

vec4 uvt;
uvt.x = dot(tv, pv);
uvt.y = dot(ray.direction, qv);
uvt.z = dot(e1, qv);
uvt.xyz = uvt.xyz / det;
uvt.w = 1.0 - uvt.x - uvt.y;
if (all(greaterThanEqual(uvt, vec4(0.0))) && (uvt.z < hit.a)) {
    hit = vec4(triangle_id.uv, material_id, uvt.z);
}
```
Packing vertex data

- Each vertex is packed into two vec4 data
  - Normal can be reconstructed via sign(z)
  - Material id is redundantly copied three times

<table>
<thead>
<tr>
<th>vec4_0</th>
<th>position.x</th>
<th>position.y</th>
<th>position.z</th>
<th>texcoord.u</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec4_1</td>
<td>normal.x</td>
<td>normal.y</td>
<td>sign(normal.z) * material_id</td>
<td>texcoord.v</td>
</tr>
</tbody>
</table>
Performance

M rays/sec @ GeForce GT 630
Performance

- 2.5 ~ 3.0 times faster than threaded BVH
- Roughly 0.5 of highly optimized SVBH traversal kernel for NVIDIA GPUs [Aila 09]
  - Not too bad for cross-platform code in my opinion
  - Threading (x 6 times) is very fast
- Can use SBVH with this algorithm as well
  - Potentially fill the rest of the performance gap
Memory overhead

- Original threaded BVH
  - Triangle: 8 floats $\times$ 3 vertex
  - Bounding box: 4 floats $\times$ 2 (min & max)
  - Hit/miss links: 4 floats
- Total: 36 floats
Memory overhead

• **Multiple-threaded BVH**

  • Triangle: 8 floats $\times$ 3 vertex

  • Bounding box: 4 floats $\times$ 2 (min & max)

  • Hit/miss links: 4 floats $\times$ 6 directions

• Total: **56** floats
Memory overhead

- Multiple-threaded BVH
  - Triangle: 8 floats × 3 vertex
  - Bounding box: 4 floats × 2 (min & max)
  - Hit/miss links: 4 floats × 6 directions

(only) 1.5 times of the original

- Total: 56 floats
Other stackless traversals

- There are many different approaches
  - Bitwise operation [Barringer13, Afra13…]
  - Restarting [Foley05, Laine10, Hapala11…]
- Multiple-threaded BVH seems faster in my tests
  - Traversal algorithm is extremely simple
  - 1.5 times memory overhead is acceptable IMHO
Dynamic scenes

• Threaded BVH can be constructed entirely on GPUs
  • Just like linear BVH (sorting + indexing)
  • Hit/miss links can be constructed on the fly, too
CPU
Build BVH

GPU
Photon tracing
Eye ray tracing
Density estimation
Photon tracing
Photon tracing on GPUs

- One pixel = one photon path
Photon tracing on GPUs

- One pixel = one photon path
Photon tracing on GPUs

- One pixel = one photon path
Photon tracing on GPUs

- One pixel = one photon path
Photon tracing on GPUs

- One pixel = one photon path
Challenges

• The number of bounces can vary a lot
  • Don’t want to wait until long ones terminate
  • Need make a list of photons
• Need high quality random numbers in parallel
  • Only with floating-point number operations
  • “Noise” function won’t work
Photon tracing on GPUs
Photon tracing on GPUs

Ray tracing from a light source
Photon tracing on GPUs

Hit points = photons
Photon tracing on GPUs
Photon tracing on GPUs
Photon tracing on GPUs
Photon tracing on GPUs

Multiple photons per pixel
Asynchronous path regeneration

- Each photon pass = only one bounce

- Photon paths are asynchronously regenerated
  - As soon as it’s terminated, sample a new one
  - Count the number of photon paths via reduction

- Similar to the idea by Novak et al. [2010]
1st pass

# of bounces

0
0
0
0
1st pass

0th bounce = gen. a new path

# of bounces
1st pass

0th bounce = gen. a new path

# of bounces

# of zeros = 4
1st pass

# of bounces

Terminated

# of zeros = 4
2nd pass

New path

# of bounces

1
1
1
0

# of zeros = 5
2nd pass

# of bounces

# of zeros = 5
2nd pass

# of bounces

# of zeros = 5
3rd pass

# of bounces

# of zeros = 8
Performance

![Graph showing Performance](image)

- **Number of bounces**: 5, 6, 7, 8, 9, 10
- **M paths/sec**: Changes from 1.5 to 0.7
  - **Regeneration**: Yellow line
  - **Multiple bounces**: Blue line

Inset image: A small image showing a room with a light fixture.
Random number generator

Famous fract(sin(...)) PRNG

Good PRNG
Random number generator

- Photon mapping is a statistical computation
  - “Noise function” is not random enough
  - Low quality random numbers = artifacts
- Legacy GLSL does not support integer operations
  - Existing good PRNGs use integer operations
- Need lots of good random numbers in parallel
Random number generator

• Modification of PRNG of unknown origin (post on an old GPGPU forum), but works surprisingly well and very fast

```cpp
float GPURnd(inout vec4 state) {
    const vec4 q = vec4(1225.0, 1585.0, 2457.0, 2098.0);
    const vec4 r = vec4(1112.0, 367.0, 92.0, 265.0);
    const vec4 a = vec4(3423.0, 2646.0, 1707.0, 1999.0);
    const vec4 m = vec4(4194287.0, 4194277.0, 4194191.0, 4194167.0);

    vec4 beta = floor(state / q);
    vec4 p = a * (state - beta * q) - beta * r;
    beta = (sign(-p) + vec4(1.0)) * vec4(0.5) * m;
    state = (p + beta);

    return fract(dot(state / m, vec4(1.0, -1.0, 1.0, -1.0)));
}
```
Other PRNGs

• LCG: works fine only if you do some simple stuff
• Crypto-hash: works well, but somewhat slower
• (GPU) Mersenne twister: works well, but too slow
• xorshift: not very suitable for parallel PRNGs

• My choices: crypto-hash or the one in prev. slide
CPU

Build BVH

Photon tracing

GPU

Eye ray tracing
Density estimation
Eye ray tracing
Eye ray tracing

• Almost the same as photon but one difference
  • Trace a path until it hits a “non-specular” surface
  • Single pass = multiple bounces
  • No asynchronous path regeneration (run it once per multiple photon passes to balance the loads)
• Store the result for the density estimation step
Eye ray tracing
Density estimation

• Find nearby photons around the eye ray hit point
Challenges

• Brute-force search is too slow (O(N) for N photons)
• Photons are newly generated at each pass
  • Cannot use a fixed data structure like BVHs
• More nearby photons = more computation
  • Highly non-uniform workload distribution
Spatial hashing

- Multidimensional extension of regular hashing
- Two phase
  - Construct a hash table
  - Query the hash table
Construction
Construction
Construction
Construction
Construction
Random writes using points

- Drawing one vertex per pixel
- Write into a specific pixel, not the same pixel
Hash function

• Utilize the PRNG (works fairly well)

```cpp
vec4 n = vec4(idx, (idx.x + idx.y + idx.z) / 3.0) * 4194304.0;
float hash = GPURnd(n);
```

• S-box via textures (works very well, but slow)

• Some standard integer hash functions (they can fail for spatial hashing - be careful)
Query
Query

Hash
Query

Hash
Problems

- Need to make a list when hash collision occurs
- Not GPU friendly data structure
- Some hashed lists can contain lots of photons
- Very non-uniform workload distribution
Stochastic spatial hashing

Randomly keep only one photon

"Parallel Progressive Photon Mapping on GPUs” T. Hachisuka and H. W. Jensen
SIGGRAPH Asia 2010 Technical Sketches
Construction
Construction
Construction
Construction
Construction
Construction
Construction
Construction

Hash
Query

Hash
Implementation

- Extremely simple!

For all photons in parallel
HashIndex = Hash(Photon.Position)
Table[HashIndex] = Photon
AtomicInc(Count[HashIndex])
Comparison with spatial hashing

- Red: Alcantara et al.: Construction
- Green: Alcantara et al.: Retrieval
- Blue: Our Method: Construction
- Pink: Our Method: Retrieval

Milliseconds

Key-value pairs (millions)
Comparison with tree

- Photon Tracing
- Photon Map Construction
- Gathering & Rendering

- Fabianowski09: Ring
- Zou08: Glass
- Stochastic Hashing: Cognac

×3 ~ ×10 faster
Comparison with CPU

OptiX sample (CPU & GPU)

Photon Tracing
Photon Map Construction
Gathering & Rendering

Stochastic Hashing (GPU)

Construction alone: ×30
Total: ×5
Additional noise

1:64 table  |  1:1 table  |  Full list
Stochastic spatial hashing

- Fundamentally avoids the two issues
  - No list construction is necessary
  - Hashed entry contains only one photon at most
- Added bonus - very easy to implement
Other Tips
Texturing

- You don’t want to have a separate GL texture for each
- Slow & the number of textures is limited
- Store multiple textures as one *volume texture*

\[ \text{texture3D(textures, vec3(hit.texcoord.uv, hit.mat_id)).rgb} \]
Data structure for materials

- “Über shader” fits well with the current system

- Three options to store material data
  - Texture - generally the slowest
  - Uniform - faster than texture, but limited
  - Embedded - need to compile shaders
Lowering CPU usage

- Naive implementation causes 100% CPU usage.
- Due to the way OpenGL waits for next command.
- GPU renderer uses 100% CPU sounds stupid!
Lowering CPU usage

• Use asynchronous occlusion query

• Wait until we get the number of pixels drawn back

• Use non-busy sleep (e.g., usleep)

```c
int a = 0;
glBeginQuery(GL_TIME_ELAPSED_EXT, OcclusionQuery);
    // draw quad
glEndQuery(GL_TIME_ELAPSED_EXT);
do {
    glGetQueryObjectiv(OcclusionQuery, GL_QUERY_RESULT_AVAILABLE, &a);
    sleep(1);
} while(!a);
```
16bits vs 32bits

• GLSL can easily use 16bits floats
• Surprising(?) fact: 16bits is often times enough!
  • As long as you convert everything into 16bits
  • Perhaps not true for very large scenes
• Usually slightly faster than 32bits
Cross-platform issues

- OpenCL and GLSL are cross-platform, in theory
  - This is the reason to use “legacy” GLSL
  - Battle-tested GLSL versions are stable enough
  - My code works on Intel’s, NVIDIA’s, and AMD’s
- Some annoyance only in rare cases
  - “mod” produces wrong results (use floor and arithmetics)
  - conditional while loop does not work (use break instead)
Live demo

Approx. 100M photon paths in 1 min @ GeForce GTX 680
Conclusions

- Fully functional rendering system using GLSL
  - Multiple-threaded BVH
  - Asynchronous path generation
  - PRNG using only floating-point numbers
  - Stochastic hashing