Welcome!

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

- Lecture 8 – Loose Ends
- State of the Art
- Wavefront Path Tracing
- Random Numbers
Incoming direct light

\[ \begin{align*}
-\frac{1}{2}\pi & \quad A \quad B \\
\int_{\Omega} L_d(x, \omega_i) \cos \theta_i \, d\omega_i \\
\approx & \frac{2\pi}{N} \sum_{i=1}^{N} L_d(p, \Omega_i) \cos \theta_i \\
= & \int_{A..B} L_d(x, \omega_i) \cos \theta_i \, d\omega_i + \int_{C..D} L_d(x, \omega_i) \cos \theta_i \, d\omega_i
\end{align*} \]
Next Event Estimation

Per surface interaction, we trace two random rays.

- Ray A returns (via point $x$) the energy reflected by $y$ (estimates indirect light for $x$).
- Ray B returns the direct illumination on point $x$ (estimates direct light on $x$).
- Ray C returns the direct illumination on point $y$, which will reach the sensor via ray A.
- Ray D leaves the scene.
Next Event Estimation

```cpp
Color Sample( Ray ray )
{
    // trace ray
    I, N, material = Trace( ray );
    BRDF = material.albedo / PI;
    // terminate if ray left the scene
    if (ray.NOHIT) return BLACK;
    // terminate if we hit a light source
    if (material.isLight) return BLACK;
    // sample a random light source
    L, Nl, dist, A = RandomPointOnLight();
    Ray lr( I, L, dist );
    if (N∙L > 0 && Nl∙-L > 0) if (!Trace( lr ))
    {
        solidAngle = ((Nl∙-L) * A) / dist^2;
        Ld = lightColor * solidAngle * BRDF * N∙L;
    }
    // continue random walk
    R = DiffuseReflection( N );
    Ray r( I, R );
    Ei = Sample( r ) * (N∙R);
    return PI * 2.0f * BRDF * Ei + Ld;
}
```
Next Event Estimation

Some vertices require special attention:

- If the first vertex after the camera is emissive, its energy can’t be reflected to the camera.
- For specular surfaces, the BRDF to a light is always 0.

Since a light ray doesn’t make sense for specular vertices, we will include emission from a vertex directly following a specular vertex.

The same goes for the first vertex after the camera: if this is emissive, we will also include this.

This means we need to keep track of the type of the previous vertex during the random walk.
Color Sample( Ray ray, bool lastSpecular )
{
    // trace ray
    I, N, material = Trace( ray );
    BRDF = material.albedo / PI;
    // terminate if ray left the scene
    if (ray.NOHIT) return BLACK;
    // terminate if we hit a light source
    if (material.isLight)
    {
        if (lastSpecular) return material.emissive;
        else return BLACK;
    }
    // sample a random light source
    L, NL, dist, A = RandomPointOnLight();
    Ray lr( I, L, dist );
    if ((N·L > 0 && NL·-L > 0) if (!Trace( lr ))
    {
        solidAngle = ((NL·-L) * A) / dist^2;
        Ld = lightColor * solidAngle * BRDF * N·L;
    }
    // continue random walk
    R = DiffuseReflection( N );
    Ray r( I, R );
    Ei = Sample( r, false ) * (N·R);
    return PI * 2.0f * BRDF * Ei + Ld;
}
Today’s Agenda:

- Lecture 8 – Loose Ends
- State of the Art
- Wavefront Path Tracing
- Random Numbers
Previously in Advanced Graphics

**A Brief History of GPU Ray Tracing**

2002: Purcell et al., multi-pass shaders with stencil, grid, low efficiency
2005: Foley & Sugerman, kD-tree, stack-less traversal with kdrestart
2007: Horn et al., kD-tree with short stack, single pass with flow control
2007: Popov et al., kD-tree with ropes
2007: Günther et al., BVH with packets.

- The use of BVHs allowed for complex scenes on the GPU (millions of triangles);
- CPU is now outperformed by the GPU;
- GPU compute potential is not realized;
- Aspects that affect efficiency are poorly understood.
Understanding the Efficiency of Ray Traversal on GPUs

Observations on BVH traversal:

Ray/scene intersection consists of an unpredictable sequence of node traversal and primitive intersection operations. This is a major cause of inefficiency on the GPU.

Random access of the scene leads to high bandwidth requirement of ray tracing.

BVH packet traversal as proposed by Gunther et al. should alleviate bandwidth strain and yield near-optimal performance.

Packet traversal doesn’t yield near-optimal performance. Why not?

*: Understanding the Efficiency of Ray Tracing on GPUs, Aila & Laine, 2009.

Understanding the Efficiency of Ray Traversal on GPUs

Simulator:

1. Dump sequence of traversal, leaf and triangle intersection operations required for each ray.
2. Use generated GPU assembly code to obtain a sequence of instructions that need to be executed for each ray.
3. Execute this sequence assuming ideal circumstances:
   - Execute two instructions in parallel;
   - Make memory access ‘free’.

The simulator reports on estimated execution speed and SIMD efficiency.

➔ The same program running on an actual GPU can never do better;
➔ The simulator provides an upper bound on performance.
Understanding the Efficiency of Ray Traversal on GPUs

Test setup

Scene: “Conference”, 282K tris, 164K nodes

Ray distributions:

1. Primary: coherent rays
2. AO: short divergent rays
3. Diffuse: long divergent rays

Hardware: NVidia GTX285.
**Understanding the Efficiency of Ray Traversal on GPUs**

Simulator, results, in MRays/s:

Packet traversal as proposed by Gunther et al. is a factor 1.7-2.4 off from simulated performance:

<table>
<thead>
<tr>
<th></th>
<th>Simulated</th>
<th>Actual</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>149.2</td>
<td>63.6</td>
<td>43</td>
</tr>
<tr>
<td>AO</td>
<td>100.7</td>
<td>39.4</td>
<td>39</td>
</tr>
<tr>
<td>Diffuse</td>
<td>36.7</td>
<td>16.6</td>
<td>45</td>
</tr>
</tbody>
</table>

*(this does not take into account algorithmic inefficiencies)*

Hardware: NVidia GTX285.
Simulating Alternative Traversal Loops

Variant 1: ‘while-while’

while ray not terminated
  while node is interior node
    traverse to the next node
  while node contains untested primitives
    perform ray/prim intersection

Results:

<table>
<thead>
<tr>
<th></th>
<th>Simulated</th>
<th>Actual</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>149.2</td>
<td>166.7</td>
<td>63.6</td>
</tr>
<tr>
<td>AO</td>
<td>100.7</td>
<td>160.7</td>
<td>39.4</td>
</tr>
<tr>
<td>Diffuse</td>
<td>36.7</td>
<td>81.4</td>
<td>16.6</td>
</tr>
</tbody>
</table>

Here, every ray has its own stack; This is simply a GPU implementation of typical CPU BVH traversal. Compared to packet traversal, memory access is less coherent. One would expect a larger gap between simulated and actual performance. However, this is not the case (not even for divergent rays).

Conclusion: **bandwidth is not the problem**.

Hardware: NVidia GTX285.
Simulating Alternative Traversal Loops

Variant 2: ‘if-if’

while ray not terminated
  if node is interior node
    traverse to the next node
  if node contains untested primitives
    perform a ray/prim intersection

Results:

<table>
<thead>
<tr>
<th></th>
<th>Simulated</th>
<th>Actual</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>166.7</td>
<td>129.3</td>
<td>88.0</td>
</tr>
<tr>
<td>AO</td>
<td>160.7</td>
<td>131.6</td>
<td>86.3</td>
</tr>
<tr>
<td>Diffuse</td>
<td>81.4</td>
<td>70.5</td>
<td>44.5</td>
</tr>
</tbody>
</table>

This time, each loop iteration either executes a traversal step or a primitive intersection.

Memory access is even less coherent in this case.

Nevertheless, it is faster than while-while. Why?

While-while leads to a small number of long-running warps. Some threads stall while others are still traversing, after which they stall again while others are still intersecting.

Hardware: NVidia GTX285.
Simulating Alternative Traversal Loops

Variant 3: ‘*persistent while-while*’

Idea: rather than spawning a thread per ray, we spawn the ideal number of threads for the hardware.

Each thread increases an atomic counter to fetch a ray from a pool, until the pool is depleted*.

Benefit: we bypass the hardware thread scheduler.

Results:

<table>
<thead>
<tr>
<th></th>
<th>Simulated</th>
<th>Actual</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>129.3</td>
<td>166.7</td>
<td>90.1</td>
</tr>
<tr>
<td>AO</td>
<td>131.6</td>
<td>160.7</td>
<td>88.8</td>
</tr>
<tr>
<td>Diffuse</td>
<td>70.5</td>
<td>81.4</td>
<td>45.3</td>
</tr>
</tbody>
</table>

This test shows what the limiting factor was: thread scheduling. By handling this explicitly, we get much closer to theoretical optimal performance.

*: In practice, this is done per warp: the first thread in the warp increases the counter by 32. This reduces the number of atomic operations.

Hardware: NVidia GTX285.
Simulating Alternative Traversal Loops

Variant 4: ‘speculative traversal’

Idea: while some threads traverse, threads that want to intersect prior to (potentially) continuing traversal may just as well traverse anyway – the alternative is idling.

Drawback: these threads now fetch nodes that they may not need to fetch*. However, we noticed before that bandwidth is not the issue.

Results for persistent speculative while-while:

<table>
<thead>
<tr>
<th></th>
<th>Simulated</th>
<th>Actual</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>166.7</td>
<td>165.7</td>
<td>135.6</td>
</tr>
<tr>
<td></td>
<td>135.6</td>
<td>142.2</td>
<td>81</td>
</tr>
<tr>
<td>AO</td>
<td>160.7</td>
<td>169.1</td>
<td>130.7</td>
</tr>
<tr>
<td></td>
<td>130.7</td>
<td>134.5</td>
<td>81</td>
</tr>
<tr>
<td>Diffuse</td>
<td>81.4</td>
<td>92.9</td>
<td>62.4</td>
</tr>
<tr>
<td></td>
<td>62.4</td>
<td>60.9</td>
<td>77</td>
</tr>
</tbody>
</table>

For diffuse rays, performance starts to differ significantly from simulated performance. This suggests that we now start to suffer from limited memory bandwidth.

*: On a SIMT machine, we do not get redundant calculations using this scheme. We do however increase implementation complexity, which may affect performance.

Hardware: NVidia GTX285.
### Understanding the Efficiency of Ray Traversal on GPUs

- Three years later** -

In 2009, NVidia’s Tesla architecture was used (GTX285).

#### Results on Tesla (GTX285), Fermi (GTX480) and Kepler (GTX680):

<table>
<thead>
<tr>
<th></th>
<th>Tesla</th>
<th>Fermi</th>
<th>Kepler</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primary</strong></td>
<td>142.2</td>
<td>272.1</td>
<td>432.6</td>
</tr>
<tr>
<td><strong>AO</strong></td>
<td>134.5</td>
<td>284.1</td>
<td>518.2</td>
</tr>
<tr>
<td><strong>Diffuse</strong></td>
<td>60.9</td>
<td>126.1</td>
<td>245.4</td>
</tr>
</tbody>
</table>

**: Aila et al., 2012. Understanding the efficiency of ray traversal on GPUs - Kepler and Fermi Addendum.

---

#### Code Snippet

```c

// ... (code snippet)

if (refl && !diffuse) {
    return 0.0;
}
```

---

*Note: The code snippet is part of a larger context and is not fully shown here for brevity.*
The graph confirms: GPU ray tracing is compute-bound. On newer hardware, it scales with FLOPS, not GB/s.
Latency Considerations of Depth-first GPU Ray Tracing*

A study of GPU ray tracing performance in the spirit of Aila & Laine has been published in 2014 by Guthe. Three optimizations are proposed:

1. Using a shallower hierarchy;
2. Loop unrolling for the while loops;
3. Loading data at once rather than scattered over the code.

<table>
<thead>
<tr>
<th>Primary</th>
<th>Titan (AL'09)</th>
<th>Titan (Guthe)</th>
<th>+%</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO</td>
<td>605.7</td>
<td>688.6</td>
<td>13.7</td>
</tr>
<tr>
<td>Diffuse</td>
<td>527.2</td>
<td>613.3</td>
<td>16.3</td>
</tr>
<tr>
<td></td>
<td>216.4</td>
<td>254.4</td>
<td>17.6</td>
</tr>
</tbody>
</table>

*: Latency Considerations of Depth-first GPU Ray Tracing, Guthe, 2014
Shallow Bounding Volume Hierarchies*

Idea:

*We can cut the number of traversal steps in half if our BVH nodes have 4 instead of 2 child nodes.*

Additional benefits:

- A proper layout allows for SIMD intersection of all four child AABBs;
- We increase the arithmetic density of a single traversal step.

*: Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays, Dammertz et al., 2008
Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-branching BVHs, Wald et al., 2008
Building the MBVH

**Collapsing a regular BVH**

For each node $n$: iterate over the children $c_i$:

1. See if we can ‘adopt’ the children of $c_i$: 
   \[ N_n - 1 + N_{c_i} \leq 4; \]
2. Select the child with the greatest area;
3. Replace node $c_i$ with its children;
4. Repeat until no merge is possible.

Repeat this process for the children of $n$.

*Note that for this tree, the end result has one interior node with only 2 children, and one with only 3 children.*
Building the MBVH

Data structure:

```c
struct SIMD_BVH_Node {
    __m128 bminx4, bmaxx4;
    __m128 bminy4, bmaxy4;
    __m128 bminz4, bmaxz4;
    int child[4], count[4];
};
```

To traverse a regular BVH front-to-back, we can use a single comparison to find the nearest child. For an MBVH, this is not as trivial.

**Pragmatic solution:**

1. Obtain the four intersection distances in t4;
2. Overwrite the lowest bits of each float in t4 with binary 00, 01, 10 and 11;
3. Use a small sorting network to sort t4;
4. Extract the lowest bits to obtain the correct order in which the nodes should be processed.
Today’s Agenda:

- Lecture 8 – Loose Ends
- State of the Art
- Wavefront Path Tracing
- Random Numbers
Wavefront

Mapping Path Tracing to the GPU

The path tracing loop from lecture 8 is straight-forward to implement on the GPU.

However:

- Terminated paths become idling threads;
- A significant number of paths will not trace a shadow ray.

```c
Color Sample( Ray ray )
{
    T = ( 1, 1, 1 ), E = ( 0, 0, 0 );
    while (1)
    {
        I, N, material = Trace( ray );
        BRDF = material.albedo / PI;
        if (ray.NOHIT) break;
        if (material.isLight) break;
        // sample a random light source
        L, Nl, dist, A = RandomPointOnLight();
        Ray lr( I, L, dist );
        if (N.L > 0 && Nl.L > 0) if (!Trace( lr ))
        {
            solidAngle = ((Nl.L - L) * A) / dist^2;
            lightPDF = 1 / solidAngle;
            E += T * (N.L / lightPDF) * BRDF * lightColor;
        }
        // continue random walk
        R = DiffuseReflection( N );
        hemiPDF = 1 / (PI * 2.0f);
        ray = Ray( I, R );
        T *= (N.R / hemiPDF) * BRDF;
    }
    return E;
}
```
Megakernels Considered Harmful*

Naïve path tracer:

1. **Kernel Function**
   - Generate primary ray
   - Intersect
   - Shade
   - Trace shadow ray

   **Finalize**

   **terminate?**
   - yes
   - no

   **shadow?**
   - no
   - yes

   Resource requirements (registers) - and thus parallel slack - are determined by ‘weakest link’, i.e. the functional block that requires most registers.

Translating this to CUDA or OpenCL code yields a single kernel: individual functions are still compiled to one monolithic chunk of code.

---

*Wavefront Path Tracing on GPUs, Laine et al., 2013*
Wavefront

Megakernels Considered Harmful

Solution: *split the kernel.*

Example:

Kernel 1: Generate primary rays.
Kernel 2: Trace paths.
Kernel 3: Accumulate, gamma correct, convert to ARGB32.

Consequence:

Kernel 1 generates *all* primary rays, and stores the result. Kernel 2 takes this buffer and operates on it.

⇒ **Massive memory I/O.**
Wavefront

Megakernels Considered Harmful

Taking this further: streaming path tracing*

Kernel 1: generate primary rays.
Kernel 2: extend.
Kernel 3: shade.
Kernel 4: connect.
Kernel 5: finalize.

Here, kernel 2 traces a set of rays to find the next path vertex (the random walk).

Kernel 3 processes the results and generates new path segments and shadow rays (2 separate buffers).

Kernel 4 traces the shadow ray buffer.

Kernel 1, 2, 3 and 4 are executed in a loop until no rays remain.

*: Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU, van Antwerpen, 2011
Wavefront

Megakernels Considered Harmfull

Zooming in:

The **generate** kernel produces $N$ primary rays:

| 0, 1, ... | ... N-1 |

Buffer 1: path segments ($N$ times $O,D,t$)

The **extend** kernel traces extension rays and produces intersections*.

The **shade** kernel processes intersections, and produces new extension paths as well as shadow rays:

| 0, 1, ... | ... N-1 |

Buffer 2: generated path segments ($N$ times $O,D,t$)

Buffer 3: generated shadow rays ($N$ times $O,D,t,E$)

Finally, the **connect** kernel traces shadow rays.

---

*Note: here, the loop is implemented on the host. Each block is a separate kernel invocation.

*: An intersection is at least the $t$ value, plus a primitive identifier.
Megakernels Considered Harmful

Notes:

- We do not have to generate all primary rays at once. Instead, we chose $N$ to match hardware capabilities.
- After each loop iteration, we add sufficient primary rays to fill up the extension ray buffer.
- Full buffers are not guaranteed, especially not for shadow rays. We need to inform the host about ray counts.

Also note:

- Rays are automatically sorted.
- At the start of each kernel, occupancy is 100%.
- We can also separate rays to handle each material using its own kernel.
Megakernels Considered Harmful

Digest:

Streaming path tracing introduces seemingly costly operations:

- Repeated I/O to/from large buffers;
- A significant number of kernel invocations per frame;
- Communication with the host.

The Wavefront paper claims that this is beneficial for complex shaders. In practice, this also works for (very) simple shaders.

Also note that the megakernel paper (2013) presents an idea already presented by Dietger van Antwerpen (2011).
Today’s Agenda:

- Lecture 8 – Loose Ends
- State of the Art
- Wavefront Path Tracing
- Random Numbers
Generating Random Numbers on the GPU

Random numbers are simulated using pseudo random number generators (PNRGs).

Basic concept:

- keep a state (e.g., a single 32-bit unsigned integer);
- modify this state for each query, so that it appears to be random.

Example:

- start with a prime;
- multiply this prime by a large 32-bit prime for each query;
- integer overflow ensures that successive numbers appear random.
Generating Random Numbers on the GPU

Good RNGs have the following properties:

- Must produce uniformly distributed numbers;
- Must not repeat the same sequence;
- Must not exhibit correlation between successive numbers.

An excellent PRNG is the Mersenne Twister.

For path tracing, we need a pretty decent PRNG – our entire algorithm is based on randomness. Question is: how good does it have to be?
Consider the following PRNG:

```c
float Xor32( uint& seed )
{
    seed ^= seed << 13;
    seed ^= seed >> 17;
    seed ^= seed << 5;
    return seed * 2.3283064365387e-10f;
}
```

Complexity: 6 (cheap) operations.

In practice, we get away with this in a path tracer.

*: Marsaglia, Xorshift RNGs, 2003.
Seeding the PRNG

When running thousands of threads, we must be careful to avoid correlation between pixels. This requires careful selection of the seed for the PRNG.

On top of this, we do not want to keep the state of the RNG from frame to frame; it must be seeded for each invocation.

- A thread is uniquely identified by its thread ID.
- Combining this with the frame number ensures different sequences over time.

Initializing the seed:

```
uint seed = (threadID + frameID * largePrime1) * largePrime2;
```

For a list of 9-digit primes (will fit in 32-bit) see: [http://www.rsok.com/~jrm/9_digit_palindromic_primes.html](http://www.rsok.com/~jrm/9_digit_palindromic_primes.html)
Today's Agenda:

- Lecture 8 – Loose Ends
- State of the Art
- Wavefront Path Tracing
- Random Numbers
INFOMAGR – Advanced Graphics

Jacco Bikker - November 2018 - February 2019

END of “GPU Ray Tracing (2)”

next lecture: “Big Picture”