Welcome!

Lecture 4 - “Real-time Ray Tracing”
Today's Agenda:

- Introduction
- Ray Distributions
- The Top-level BVH
- Real-time Ray Tracing
- Assignment 2
Introduction

Cost Breakdown for Ray Tracing:

- Pixels
- Primitives
- Light sources
- Path segments

*Mind scalability as well as constant cost.*

Example: scene consisting of 1k spheres and 4 light sources, diffuse materials, rendered to 1M pixels:

$1M \times 5 \times 1k = 5 \cdot 10^9$ ray/prim intersections.

(multiply by desired framerate for realtime)

Using the BVH:

- Pixels
- N primitives $\Rightarrow \log N$ deep tree
- Light sources
- Path segments

Example: scene consisting of 1k spheres and 4 light sources, diffuse materials, rendered to 1M pixels:

$1M \times 5 \times 10 = 5 \cdot 10^7$ ray/(prim or node) intersections.

(multiply by desired framerate for realtime)
Introduction

Advanced Graphics

– Real-time Ray Tracing
Performance is now OK, but we’re not quite ready to render a game world.
Introduction

We need to go deeper
Today's Agenda:

- Introduction
- Ray Distributions
- The Top-level BVH
- Real-time Ray Tracing
- Assignment 2
Cost of ray tracing:

- Ray data
- Node data
- Triangle data
- Material data (incl. textures)

Dominated by memory access cost.
Ray Distributions

Primary rays:
For a tile of pixels, these are organized in a narrow frustum. All rays share a common origin.
Shadow rays:
For point lights, shadow rays also tend to travel close together. When traced from the light source, they too have a common origin.
Secondary rays: Reflected and refracted rays tend to diverge significantly.
Coherence

Primary rays and shadow rays for point lights are *coherent*:
- they tend to intersect the same primitives;
- they tend to traverse the same BVH nodes.

Our problem:

*Ray tracing cost is dominated by memory latency.*

Solution:

*Amortize cost of fetching data over multiple rays.*
Ray Distributions

Coherent Ray Tracing*

SIMD: four rays for the price of one.

BVHNode::Traverse( Ray r )
{
    if (!r.Intersects( bounds )) return;
    if (isleaf())
    {
        IntersectPrimitives();
    }
    else
    {
        pool[left].Traverse( r );
        pool[left + 1].Traverse( r );
    }
}

*: Interactive Rendering with Coherent Ray Tracing, Wald et al., 2001
Coherent Ray Tracing*

SIMD: four rays for the price of one.

```cpp
BVHNode::Traverse( Ray4 r4 )
{
    if (!r4.Intersects( bounds )) return;
    if (isleaf())
    {
        IntersectPrimitives();
    }
    else
    {
        pool[left].Traverse( r4 );
        pool[left + 1].Traverse( r4 );
    }
}
```

*: Interactive Rendering with Coherent Ray Tracing, Wald et al., 2001
Coherent Ray Tracing

Ray packet traversal:
- intersect four rays with a single BVH node;
- if any ray in the packet intersects the node, we traverse it;
- if the node is a leaf node, we intersect the four rays with each primitive in the leaf.

Masking:
- We maintain an ‘active’ mask for disabling rays that do not intersect a node.
Coherent Ray Tracing*

SIMD: four rays for the price of one.

BVHNode::Traverse( Ray4 r4, bool4 mask4 )
{
    bool4 hit4 = r4.Intersects( bounds ) & mask4;
    if (none( hit4 )) return;
    if (isleaf())
    {
        IntersectPrimitives();
    }
    else
    {
        pool[left].Traverse( r4, hit4 );
        pool[left + 1].Traverse( r4, hit4 );
    }
}
Ray Distributions

Coherent Ray Tracing

Results:

- for coherent packets, memory traffic is reduced;
- overall performance is improved by ~2.3x.

Overhead:

- if only a single ray requires traversal or intersection, all four rays perform this operation.
Ray Distributions

Large Packets*

Cost of memory access can be amortized over more rays by using larger packets.

Note that a naïve approach will lead to significant overhead.

We therefore add a frustum test to rapidly reject BVH nodes: If the packet frustum does not intersect the node AABB, we discard the node. The cost of this operation is independent of the number of rays in the packet.

Likewise, a node is traversed as soon as we find that a ray intersects it. This is also independent of packet size.

*: Large Ray Packets for Real-time Whitted Ray Tracing, Overbeck et al., 2008
Ray Distributions

Large Packets

Algorithm:

1. Early hit test: test first active ray against AABB
2. Early miss test: test AABB against frustum
3. Brute force test: test all rays until a hit is found.

This step yields a new first active ray index.

```cpp
BVHNode::Traverse( RayPacket rp, int first )
{
    if (!Intersects( rp[first ] ) ) // 1
    {
        if (!Intersects( rp.frustum ) )
        {
            return;
        }
        FindFirstActive( rp, ++first ); // 3
    }
    if (first < rp.rayCount )
    {
        if (isleaf())
        {
            IntersectPrimitives( rp );
        }
        else
        {
            left.Traverse( rp, first );
            right.Traverse( rp, first );
        }
    }
}
```
Ray Distributions

Large Packets

Details:

- Constructing the frustum
- Ray order & overhead
- First / last
- Optimizations: recursion, SIMD

BVHNode::Traverse( RayPacket rp, int first )
{
    if (!Intersects( rp[first ])) // 1
    {
        if (!Intersects( rp.frustum ))
            return; // 2
        FindFirstActive( rp, ++first ); // 3
    }
    if (first < rp.rayCount)
    {
        if (isleaf())
        {
            IntersectPrimitives( rp );
        }
        else
        {
            left.Traverse( rp, first );
            right.Traverse( rp, first );
        }
    }
}
Ray Distributions

Frustum Construction

Method 1, for primary rays:

Planes are easily defined using the corner rays:

\[ N_1 = (p_0 - E) \times (p_1 - p_0), \quad d_1 = N_1 \cdot E \]
\[ N_2 = (p_1 - E) \times (p_2 - p_1), \quad d_2 = N_2 \cdot E \]
\[ N_3 = (p_2 - E) \times (p_3 - p_2), \quad d_3 = N_3 \cdot E \]
\[ N_4 = (p_3 - E) \times (p_0 - p_3), \quad d_4 = N_4 \cdot E \]

Note: for secondary rays, we will not have a common origin, nor corner rays.
Ray Distributions

Frustum Construction

Method 2, for shadow rays:

1. Determine dominant axis and direction:
   \[ \overrightarrow{D_s} = \sum_{i=0}^{N} \overrightarrow{D_i} \], chose axis \( \hat{k} \) as the largest component of \( \overrightarrow{D_s} \), \( \hat{u} \) and \( \hat{v} \) are the other axes.

2. Construct a plane \( P \) orthogonal to \( \hat{k} \)

3. Calculate intersection coordinates \( u, v \) of the rays with \( P \)

4. Determine \( u_{\text{min}}, u_{\text{max}} \) and \( v_{\text{min}}, v_{\text{max}} \)

5. Corner rays are now:
   
   \[ (u_{\text{min}}, v_{\text{min}}) \]
   
   \[ (u_{\text{max}}, v_{\text{min}}) \]
   
   \[ (u_{\text{max}}, v_{\text{max}}) \]
   
   \[ (u_{\text{min}}, v_{\text{max}}) \]

Note: this still requires a common origin.
Ray Distributions

Frustum Construction

Method 3, for generic rays:

1. Construct two planes: 
   \( P_{\text{far}} \), orthogonal to \( \hat{k} \), at location \( k_{\text{far}} \) which is obtained from the scene AABB; 
   \( P_{\text{near}} \), orthogonal to \( \hat{k} \), at location \( k_{\text{near}} \), which is obtained from the AABB over the ray origins.

2. Calculate intersection coordinates \( u_{\text{far}}, v_{\text{far}} \) and \( u_{\text{near}}, v_{\text{near}} \) of the rays with \( P_{\text{far}} \) and \( P_{\text{near}} \).

3. Determine \( u_{\min, \max}, v_{\min, \max} \) of the corner rays as
   \[
   \begin{align*}
   (u_{\min}, v_{\min}) &= (u_{\min, \max}, v_{\min, \max}) - (u_{\min, \max}, v_{\min, \max}), \\
   (u_{\max}, v_{\max}) &= (u_{\min, \max}, v_{\min, \max}) - (u_{\min, \max}, v_{\min, \max}),
   \\
   (u_{\min, \max}, v_{\min, \max}) &= (u_{\min, \max}, v_{\min, \max}) - (u_{\min, \max}, v_{\min, \max}).
   \end{align*}
   \]
Ray Order

The order of the rays in a packet is important. We keep track of the first active ray: in this case the green dot.

We thus enter the node with 61 rays, while only 12 rays actually intersect the node.

Keeping track of the last ray helps somewhat.
Ray Distributions

Ray Order

Overhead can be reduced by numbering rays in each quadrant sequentially.
Ray Distributions

Ray Order

Overhead can be reduced by numbering rays in each quadrant sequentially.

For the general case, Morton order is optimal.
Divergent Rays: Partition Traversal

```c
int PartitionRays
```

```c
for ( int i = 0; i < ia; i++ )
    if (ray[idx[i]].IntersectsAABB())
        swap( idx[ie++], idx[i] );
```

```c
return ie;
```

```c
rays
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```c
ray indices
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```c
ie
```

```c
ia
```
Divergent Rays: Partition Traversal

```c
int PartitionRays(int i = 0; i < ia; i++)
    if (ray[idx[i]].IntersectsAABB())
        swap(idx[ie++], idx[i]);

return ie;
```
Divergent Rays: Partition Traversal

```c
int PartitionRays
{
    for ( int i = 0; i < ia; i++ )
        if (ray[idx[i]].IntersectsAABB())
            swap(idx[ie++], idx[i]);
    return ie;
}
```
Divergent Rays: Partition Traversal

```c
int PartitionRays
    for ( int i = 0; i < i_a; i++ )
        if (ray[idx[i]].IntersectsAABB())
            swap( idx[i_e++], idx[i] );
return i_e;
```
Divergent Rays: Partition Traversal

```c
int PartitionRays
    for ( int i = 0; i < ia; i++ )
        if (ray[idx[i]].IntersectsAABB())
            swap( idx[ie++], idx[i] );
return ie;
```
Ray Distributions

Divergent Rays: Partition Traversal

Partition traversal gathers active rays in a continuous list.

This comes at the price of some overhead:

- one layer of indirection (ray indices ➔ rays);
- swapping of indices.

In practice, this method is suitable for ray distributions where large gaps in the ray set are to be expected.
Optimization: Recursion

The recursion can be replaced by a local stack:

```c
struct Stack {
  BVHNode* node;
  int first;
};
Stack stack[STACKSIZE];
stack[0].node = GetBVHRoot();
stack[0].first = 0;
int stackPtr = 1;
while (stackPtr > 0) {
  BVHNode* node = stack[--stackPtr].node;
  first = stack[stackPtr].first;
  ...
}
```

```c
BVHNode::Traverse( RayPacket rp, int first )
{
  if (!Intersects( rp[first ]))  // 1
    {
      if (!Intersects( rp.frustum ))
        return;  // 2
      FindFirstActive( rp, ++first ); // 3
    }
  if (first < rp.rayCount)
    {
      if (isleaf())
        {
          IntersectPrimitives( rp );
        }
      else
        {
          left.Traverse( rp, first );
          right.Traverse( rp, first );
        }
    }
}
```
Ray Distributions

Optimization: SIMD

We can still use SIMD to test four rays at once.

- The smallest primitive thus becomes the ‘QuadRay’;
- a packet of $N$ rays consists of $N/4$ QuadRays;
- ‘first’ points to the first QuadRay that has at least one active ray;
- FindFirst processes the remaining rays four at a time.

Note: for AVX, replace ‘four’ by ‘eight’.
Ray Distributions

Results

Compared to 2x2 SIMD packet traversal, ranged traversal improves primary and shadow rays by ~3.5x.

Note that ray divergence has a large impact on performance.
Today's Agenda:

- Introduction
- Ray Distributions
- The Top-level BVH
- Real-time Ray Tracing
- Assignment 2
Top-level BVH

Ray Tracing Animated Scenes

Covered so far:

- Static geometry: high-quality construction with spatial splits
- Deformations: BVH refitting (water, waving trees, ...)
- Structural changes: binned BVH construction

Not covered:

- Rigid motion
Top-level BVH
Top-level BVH
Top-level BVH

Combining BVHs
Combining BVHs

Two BVHs can be combined into a single BVH, by simply adding a new root node pointing to the two BVHs.

- This works regardless of the method used to build each BVH
- This can be applied repeatedly to combine many BVHs
Top-level BVH

Scene Graph
Top-level BVH

Scene Graph

If our application uses a scene graph, we can construct a BVH for each scene graph node.

The BVH for each node is built using an appropriate construction algorithm:

- High-quality SBVH for static scenery (offline)
- Fast binned SAH BVHs for dynamic scenery

The extra nodes used to combine these BVHs into a single BVH are known as the Top-level BVH.
Rigid Motion

Applying rigid motion to a BVH:

1. Refit the top-level BVH
2. Refit the affected BVH
Top-level BVH

Rigid Motion

Applying rigid motion to a BVH:

1. Refit the top-level BVH
2. Refit the affected BVH

or:

2. **Transform the ray, not the node**

Rigid motion is achieved by transforming the rays by the *inverse transform* upon entering the sub-BVH.

*(this obviously does not only apply to translation)*
Top-level BVH

The Top-level BVH - Construction

Input: list of axis aligned bounding boxes for transformed scene graph nodes

Algorithm:

1. Find the two elements in the list for which the AABB has the smallest surface area
2. Create a parent node
3. Replace the two elements in the list by the parent node
4. Repeat until one element remains in the list.

Note: Algorithmic complexity is $O(N^3)$. 
The Top-level BVH – Faster Construction*

Algorithm:

Node A = list.GetFirst();
Node B = list.FindBestMatch( A );
while (list.size() > 1)
{
    Node C = list.FindBestMatch( B );
    if (A == C)
    {
        list.Remove( A );
        list.Remove( B );
        A = new Node( A, B );
        list.Add( A );
        B = list.FindBestMatch( A );
    }
    else A = B, B = C;
}

*: Fast Agglomerative Clustering for Rendering, Walter et al., 2008
The Top-level BVH – Traversal

The leafs of the top-level BVH contain the sub-BVHs.

When a ray intersects such a leaf, it is transformed by the transform matrix of the sub-BVH. After this, it traverses the sub-BVH.

Once the sub-BVH has been traversed, we transform the ray again, this time by the transform matrix of the top-level BVH.

For efficiency, we store the inverted matrix with the sub-BVH root.
Today's Agenda:

- Introduction
- Ray Distributions
- The Top-level BVH
- Real-time Ray Tracing
- Assignment 2
Mount Wario
Game    Mario Kart 8
Section  Courses
Size     69.81 MB
Submitter Ray Koopa
Format   ZIP
Hits     10,137
Comments 1

Download this Model

Return to Game

https://www.models-resource.com/wii_u/mariokart8/model/10718
INFOMAGR – Advanced Graphics

Jacco Bikker - November 2018 - February 2019

END of “Real-time Ray Tracing”

next lecture: “SIMD recap”