Lecture 6 - “The Perfect BVH”

Welcome!
Today’s Agenda:

- Building Better BVHs
- Refitting
- Fast BVH Construction
- The Top-level BVH
Better BVHs

```c

void MyDLT() 
{
    // ... (code snippet)
}
```

---

Advanced Graphics – The Perfect BVH
Better BVHs

```plaintext

Better BVHs

```
What Are We Trying To Solve?

A BVH is used to reduce the number of ray/primitive intersections.

But: it introduces new intersections.

The ideal BVH minimizes:

- # of ray / primitive intersections
- # of ray / node intersections.
Better BVHs

```
First, check depth < PARAM_DEPTH:
If inside, check NT > 0.9 * NT:
If NT < 1 - 0.9 * NT, return NT := 0.9 * NT + 0.1 * NT:
If depth < depth + 1, return NT := 0.9 * NT + 0.1 * NT:

If diffuse, return true;

If (not refract) and (depth < PARAM_DEPTH):
If NT > 0.9 * NT:
Return NT := 0.9 * NT + 0.1 * NT:
Return NT := 0.9 * NT + 0.1 * NT:
Return NT := 0.9 * NT + 0.1 * NT:
Return NT := 0.9 * NT + 0.1 * NT:
Return NT := 0.9 * NT + 0.1 * NT:

If refract Diffuse:
Return true;

If refract Phi:
Return true;

WADDEPTH)
```

Survive = SurviveProbability; diffuse;
estimation = doing it properly, (1.0);
}
if radiance = SampleLight: brand, I, &L, SAMPLE_SIZE:
if x + radiance.y + radiance.z > 0) &L (0.0, 0.0, 0.0):
v = true;
if refract Diffuse = EvaluateDiffuse (L, M, i, &L, SAMPLE_SIZE):
if refract Diffuse = EvaluateDiffuse (L, M, i, &L, SAMPLE_SIZE):
if refract Diffuse = EvaluateDiffuse (L, M, i, &L, SAMPLE_SIZE):
if refract Diffuse = EvaluateDiffuse (L, M, i, &L, SAMPLE_SIZE):
if refract Diffuse = EvaluateDiffuse (L, M, i, &L, SAMPLE_SIZE):

Window walk - done properly, closely following Surface:
```
Better BVHs

**BVH versus kD-tree**

The BVH better encapsulates geometry.

➔ This reduces the chance of a ray hitting a node.

➔ This is all about probabilities!

*What is the probability of a ray hitting a random triangle?*

*What is the probability of a ray hitting a random node?*

This probability is proportional to **surface area**.
Better BVHs

Route 1: 10% up-time, $1000 fine
Route 2: 100% up-time, $100 fine
Better BVHs

Optimal Split Plane Position

The ideal split minimizes the \textit{expected cost} of a ray intersecting the resulting nodes.

This expected cost is based on:

- Number of primitives that will have to be intersected
- Probability of this happening

The cost of a split is thus:

\[ A_{\text{left}} \times N_{\text{left}} + A_{\text{right}} \times N_{\text{right}} \]
Optimal Split Plane Position

The ideal split minimizes the *expected cost* of a ray intersecting the resulting nodes.

This expected cost is based on:

- Number of primitives that will have to be intersected
- Probability of this happening

The cost of a split is thus:

\[ A_{left} \times N_{left} + A_{right} \times N_{right} \]
Better BVHs

Optimal Split Plane Position

Which positions do we consider?

Object subdivision may happen over x, y or z axis.

The cost function is constant between primitive centroids.

➔ For N primitives: $3(N - 1)$ possible locations

➔ For a 2-level tree: $(3(N - 1))^2$ configurations
Better BVHs

SAH and Termination

A split is ‘not worth it’ if it doesn’t yield a cost lower than the cost of the parent node, i.e.:

\[ A_{\text{left}} \cdot N_{\text{left}} + A_{\text{right}} \cdot N_{\text{right}} \geq A \cdot N \]

This provides us with a natural and optimal termination criterion.

(and it solves the problem of the Bad Artist)
Optimal Split Plane Position

The *surface area heuristic* (SAH) is applied in a greedy manner*.

Better BVHs

Optimal Split Plane Position

Comparing naïve versus SAH:
- SAH will cut #intersections in half;
- expect ~2x better performance.

SAH & kD-trees:
- Same scheme applies.
Better BVHs

Median Split
Surface Area Heuristic
Better BVHs
Better BVHs
Better BVHs
Today’s Agenda:

- Building Better BVHs
- Refitting
- Fast BVH Construction
- The Top-level BVH
Refitting

Summary of BVH Characteristics

A BVH provides significant freedom compared to e.g. a kD-tree:

- No need for a 1-to-1 relation between bounding boxes and primitives
- Bounding boxes may overlap
- Bounding boxes can be altered, as long as they fit in their parent box
- A BVH can be very bad but still valid

Some consequences / opportunities:

- We can rebuild part of a BVH
- We can combine two BVHs into one
- We can \textit{refit} a BVH
Refitting

Q: What happens to the BVH of a tree model, if we make it bend in the wind?

A: Likely, only bounds will change; the topology of the BVH will be the same (or at least similar) in each frame.

Refitting:

*Updating the bounding boxes stored in a BVH to match changed primitive coordinates.*
Refitting

**Upd\textit{a}ting the bounding boxes stored in a BVH to match changed primitive coordinates.**

**Algorithm:**

1. For each leaf, calculate the bounds over the primitives it represents
2. Update parent bounds
Refitting - Suitability
Refitting

Order of nodes in the node array:

*We will never find the parent of node X at a position greater than X.*

Therefore:

```c
for ( int i = N-1; i >= 0; i-- )
    nodeArray[i].AdjustBounds();
```
Today’s Agenda:

- Building Better BVHs
- Refitting
- Fast BVH Construction
- The Top-level BVH
Rapid BVH Construction

Refitting allows us to update hundreds of thousands of primitives in real-time. But what if topology changes significantly?

Rebuilding a BVH requires $3N \log N$ split plane evaluations.

Options:

1. Do not use SAH (significantly lower quality BVH)
2. Do not evaluate all 3 axes (minor degradation of BVH quality)
3. Make split plane selection independent of $N$
Binning

```
Binning
```
Binning

Binned BVH Construction*

Binned construction:

* Evaluate SAH at N discrete intervals.

Advanced Graphics – The Perfect BVH

Binning

Binned BVH Construction

Performance evaluation:

472ms 7.88M triangles (12 cores @ 2Ghz)*.

Today's Agenda:

- Building Better BVHs
- Refitting
- Fast BVH Construction
- The Top-level BVH
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
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*. 
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)*
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 for these elements
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 $O(N^3)$. 
Top-level BVH

The Top-level BVH – Faster Construction*

Algorithm:

```plaintext
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
```
Top-level BVH

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 inverted 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 sub-BVH.

For efficiency, we store the inverted matrix with the sub-BVH root.
Top-level BVH

The Top-level BVH – Summary

The top-level BVH enables complex animated scenes:

- for static objects, it contains high-quality sub-BVHs;
- for objects undergoing rigid motion, it also contains high-quality sub-BVHs, with a transform matrix and its inverse;
- for deforming objects, it contains sub-BVHs that can be refitted;
- for arbitrary animations, it contains lower quality sub-BVHs.

Combined, this allows for efficient maintenance of a global BVH.
END of “The Perfect BVH”

next lecture: “Path Tracing”