Rendering: Spatial Acceleration Structures

Bernhard Kerbl

Research Division of Computer Graphics
Institute of Visual Computing & Human-Centered Technology
TU Wien, Austria

With slides based on material by Jaakko Lehtinen, used with permission
A good image needs realistic **intensity** and **visibility**

- **Intensity** creates stimulus of optic nerve (black, white, color)
- **Visibility** makes sure that objects adhere to depth

How would you process the scene on the right to make sure the rendered output image is correct?

**(Naïve) Ray-Casting Render Loop**
- Shoot a ray through **each** pixel into the scene
- Iterate over **all** objects and test for intersection
- Record the **closest** intersection (**visibility**)
- Compute color and write to pixel (**intensity**)
void render(Camera cam) {
    for (Pixel& pix : pixels) {
        pix.Color = background;

        Intersection closest;
        closest.Distance = INFINITY;

        Ray ray = rayThroughPixel(cam, pix);

        for (Triangle& tri : triangles) {
            Intersection sect = findClosestIntersection(ray, tri);
            if (sect.Distance < closest.Distance) { closest = sect; }
        }

        if (closest.Distance != INFINITY) { pix.Color = computeColor(closest); }
    }
}
void render(Camera cam) {
    for (Pixel& pix : pixels) {
        pix.Color = background;
        Intersection closest;
        closest.Distance = INFINITY;
        Ray ray = rayThroughPixel(cam, pix);
        for (Triangle& tri : triangles) {
            Intersection sect = findClosestIntersection(ray, tri);
            if (sect.Distance < closest.Distance) { closest = sect; }
        }
        if (closest.Distance != INFINITY) {
            pix.Color = computeColor(closest);
        }
    }
}

Source: renderstuff.com/

Those are your dad’s pixels!
Instead of a single ray through each pixel, use multiple „samples“.

Pixel with sampling positions

Sampled colours

Average = displayed colour

Source: Parcly Taxel, Wikipedia “Supersampling”
void render(Camera cam) {
    for(Pixel& pix : pixels) {
        pix.Color = background;
        Intersection closest;
        closest.Distance = INFINITY;
        Ray ray = rayThroughPixel(cam, pix);
        for (Triangle& tri : triangles) {
            Intersection sect = findClosestIntersection(ray, tri);
            if(sect.Distance < closest.Distance) { closest = sect; }
        }
        if(closest.Distance != INFINITY) { pix.Color = computeColor(closest); }
    }
}
Updated Render Loop

```cpp
pix.Color = background;

Intersection closest;
closest.Distance = INFINITY;

for (int s = 0; s < NUM_SAMPLES; s++)
{
    SampleInfo sInfo = drawSample();
    Ray ray = rayThroughSample(cam, sInfo.Location);
    for (Triangle& tri : triangles)
    {
        Intersection sect = findClosestIntersection(ray, tri);
        if(sect.Distance < closest.Distance) { closest = sect; }
    }

    if(closest.Distance != INFINITY)
    {
        RGBColor sample = computeColor(closest);
        pix.Color += filter(sInfo.Filter, RGBWColor(sample, 1));
    }
}

pix.Color /= pixColor.w;

Rendering – Spatial Acceleration Structures
pix.Color = background;

Intersection closest;
closest.Distance = INFINITY;

for(int s = 0; s < NUM_SAMPLES; s++)
{
    SampleInfo sInfo = drawSample();
    Ray ray = rayThroughSample(cam, sInfo.Location);
    for (Triangle& tri : triangles)
    {
        Intersection sect = findClosestIntersection(ray, tri);
        if(sect.Distance < closest.Distance) { closest = sect; }
    }
    if(closest.Distance != INFINITY)
    {
        RGBColor sample = computeColor(closest);
        pix.Color += filter(sInfo.Filter, RGBWColor(sample, 1));
    }
}

pix.Color /= pixColor.w;
Let’s look at the basic runtime (single sample per pixel)

```c
void render(Camera cam)
{
    for (Pixel& pix : pixels)
    {
        ...  
        
        for (Triangle& tri : triangles)
        {
            ...  
        }
    }
    ...  
}
```
Let’s look at the basic runtime (single sample per pixel)

```c++
void render(Camera cam)
{
    for(Pixel& pix : pixels) \( \leftarrow N \)
    {
        ...

        for (Triangle& tri : triangles) \( \leftarrow M \)
        {
            ...
        }

        ...
    }
}
```

This is \( \mathcal{O}(N \cdot M) \), but even worse, it’s \( \Omega(N \cdot M) \)!
Is That Actually a Problem?

- Run time complexity quickly becomes a limiting factor

- High-quality scenes can have several million triangles *per object*

- Current screens and displays are moving towards 4k resolution
Amazon Lumberyard “Bistro”
3,780,244 triangles
1200x675 pixels

3 trillion ray/triangle intersection tests?

At 10M per second, one shot will take ~4 days.

Good luck with your movie!
For rendering, we will want to learn to run before we can walk

Find ways to speed up the basic loop for visibility resolution

Enter “spatial acceleration structures”

Essentially, pre-process the scene geometry into a structure that reduces expected traversal time to something more reasonable
<table>
<thead>
<tr>
<th>Structure</th>
<th>Additional Memory</th>
<th>Building Time</th>
<th>Traversal Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>none</td>
<td>abysmal</td>
</tr>
</tbody>
</table>
Consider a group of triangles

Which ones should we test?
Regular Grids

- Overlay scene with regular grid
- Sort triangles into cells
- Traverse cells and test against their contents
Regular Grids

- Overlay scene with regular grid
- Sort triangles into cells
- Traverse cells and test against their contents
- Overlay scene with regular grid
- Sort triangles into cells
- Traverse cells and test against their contents
Geometry is usually not uniform

Comes in clusters (buildings, characters, vegetation...)

Regular Grids
Geometry is usually not uniform

Comes in clusters (buildings, characters, vegetation...)

Almost all triangles in one cell! Hitting this cell will be costly!
Regular Grids

- Geometry is usually not uniform
- Comes in clusters (buildings, characters, vegetation...)
- Almost all triangles in one cell! Hitting this cell will be costly!
- Using a finer grid works
- Geometry is usually not uniform

- Comes in clusters (buildings, characters, vegetation...)

- Almost all triangles in one cell! Hitting this cell will be costly!

- Using a finer grid works, but most of its cells are unused!
## Spatial Acceleration Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Memory Consumption</th>
<th>Building Time</th>
<th>(Expected) Traversal Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>none</td>
<td>abysmal</td>
</tr>
<tr>
<td>Regular Grid</td>
<td>low – high (resolution)</td>
<td>low</td>
<td>uniform scene: ok otherwise: bad</td>
</tr>
</tbody>
</table>
Quadtrees and Octrees

- Start with scene bounds, do finer subdivisions only if needed

- Define parameters $S_{max}, N_{leaf}$

- Recursively split bounds into quadrants (2D) or octants (3D)

- Stop after $S_{max}$ subdivisions or if no cell has $> N_{leaf}$ triangles
Quad and Octrees: $N_{leaf} = 4$

- Start with scene bounds, do finer subdivisions only if needed
- Define parameters $S_{max}, N_{leaf}$
- Recursively split bounds into quadrants (2D) or octants (3D)
- Stop after $S_{max}$ subdivisions or if no cell has $> N_{leaf}$ triangles

Rendering – Spatial Acceleration Structures
Quad and Octrees: $N_{leaf} = 4$

- Start with scene bounds, do finer subdivisions only if needed

- Define parameters $S_{max}, N_{leaf}$

- Recursively split bounds into quadrants (2D) or octants (3D)

- Stop after $S_{max}$ subdivisions or if no cell has $> N_{leaf}$ triangles
Quad and Octrees: $N_{leaf} = 4$
Quad and Octrees: $N_{leaf} = 4$
Quad and Octrees: $N_{leaf} = 4$
Quad and Octrees: $N_{leaf} = 4$
Quad and Octrees: $N_{leaf} = 4$
Quad and Octrees

- Triangles may not be contained within a quadrant or octant

- Triangles must be referenced in all overlapping cells or split at the border into smaller ones
Quad and Octrees

- Triangles may not be contained within a quadrant or octant
- Triangles must be referenced in all overlapping cells or *split* at the border into smaller ones
- Can drastically increase memory consumption!
<table>
<thead>
<tr>
<th>Structure</th>
<th>Memory Consumption</th>
<th>Building Time</th>
<th>(Expected) Traversal Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>none</td>
<td>abysmal</td>
</tr>
<tr>
<td>Regular Grid</td>
<td>low – high (resolution)</td>
<td>low</td>
<td>uniform scene: ok otherwise: bad</td>
</tr>
<tr>
<td>Quadtree/Octree</td>
<td>low – high (overlap/uniformity)</td>
<td>low</td>
<td>good</td>
</tr>
</tbody>
</table>
BSP Trees & K-d Trees

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!
BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

![Diagram of BSP Trees]

Rendering – Spatial Acceleration Structures
BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

 Rendering – Spatial Acceleration Structures
BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!
BSP Trees & K-d Trees

- **Binary Space Partition Tree**
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

- **K-dimensional (K-d) Tree**
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits
BSP Trees & K-d Trees, $N_{\text{leaf}} = 4$

- Binary Space Partition Tree
  - Recursive split via \textit{hyperplanes}
  - Left/right child nodes treat objects in each \textit{half-space}
  - Splits can be arbitrary!

- K-dimensional (K-d) Tree
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits
BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

- K-dimensional (K-d) Tree
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits
BSP Trees & K-d Trees, $N_{leaf} = 4$

- Binary Space Partition Tree
  - Recursive split via *hyperplanes*
  - Left/right child nodes treat objects in each *half-space*
  - Splits can be arbitrary!

- K-dimensional (K-d) Tree
  - Every hyperplane must be perpendicular to a base axis
  - Limits search space for splits
## Spatial Acceleration Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Memory Consumption</th>
<th>Building Time</th>
<th>(Expected) Traversal Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>none</td>
<td>abysmal</td>
</tr>
<tr>
<td>Regular Grid</td>
<td>low – high (resolution)</td>
<td>low</td>
<td>uniform scene: ok otherwise: bad</td>
</tr>
<tr>
<td>Quadtree/Octree</td>
<td>low – high (overlap/uniformity)</td>
<td>low</td>
<td>good</td>
</tr>
<tr>
<td>K-d Tree</td>
<td>low – high (overlap)</td>
<td>low – high</td>
<td>good – excellent</td>
</tr>
</tbody>
</table>
Find enclosing (“conservative”) volumes that are easier to test

Ideally: tight, but easy to check for intersection with ray

Common choices:
- Bounding Spheres
- Bounding Boxes
  - Axis-aligned (AABB)
  - Oriented (OBB)

Saves on computational effort if reject
Axis-Aligned Bounding Boxes (AABBs)

- AABBs are defined by their two extrema (min/max)
  - Linear run time to compute
    - Iterate over all vertices
    - Keep min/max values for each dimension
    - Done!

\[
(x_{\text{min}}, y_{\text{min}}, z_{\text{min}}) = (\min(x_0, x_1, x_2), \min(y_0, y_1, y_2), \min(z_0, z_1, z_2))
\]

\[
(x_{\max}, y_{\max}, z_{\max}) = (\max(x_0, x_1, x_2), \max(y_0, y_1, y_2), \max(z_0, z_1, z_2))
\]
Merging AABBs

- Find the AABB that encloses multiple, smaller AABBs
- Operates only on extrema of each smaller AABB
- Merging process is commutative
Bounding Spheres

- Bounding spheres need a center \( \hat{c} \) and a radius \( r \)

- For \( \hat{c} \), can pick the mean vertex position or center of AABB

- Once center is chosen, find vertex position \( \hat{v}_{max} \) farthest from it

- \[ r = |\hat{c} - \hat{v}_{max}| \]
How to Use Bounding Volumes

- Can also be applied to entire objects
- Reject entire object if volume is not hit
- Good start, but what if...
  - scene is not partitioned into objects?
  - objects are extremely large (terrain)?
  - objects are extremely detailed (characters)?
  - there are millions of objects with ~ 2 triangles each (leaves)?
Bounding Volume Hierarchy (BVH)

- Each node of the hierarchy has its own bounding volume

- Every node can be
  - An inner node: references child nodes
  - A leaf node: references triangles

- Each node’s bounding volume is a subset of its parent’s bounding volume
  (i.e., child nodes are spatially contained by their parents)
The final hierarchy is (again) a tree structure with $N$ leaf nodes

Leaf nodes can be
- Individual triangles
- Clusters (e.g., $\leq 10\Delta$)

Total number of nodes for a binary tree: $2N - 1$
- If balanced, it takes $\sim \log N$ steps to reach a leaf from the root
- If trees have more than 2 branches, they require fewer nodes

Source: Schreiberx, Wikipedia “Bounding Volume Hierarchy”
What makes BVHs special?

- Important feature: bounding volumes can **overlap**!

- No duplicate references or split triangles necessary!

- Implicitly limits the amount of memory required
**BVH Building**

- Generating BVH and tree for input triangle geometry

- **CPU**: usually top-down
  **GPU**: usually bottom-up

- From here on out, we will consider box BVHs only
Define $N_{leaf}$ for leaves

For each node, do the following:

- Compute bounding box that fully encloses triangles & store
- Holds $\leq N_{leaf}$ triangles? Stop.
- Else, split into child groups
- Make one new node per group
- Set them as children of current
- Repeat with child nodes
BVH Building, Top-Down, \( N_{leaf} = 4 \)

Define \( N_{leaf} \) for leaves

For each node, do the following:

- Compute bounding box that fully encloses triangles & store
- Holds \( \leq N_{leaf} \) triangles? Stop.
- Else, split into child groups
- Make one new node per group
- Set them as children of current
- Repeat with child nodes
Define $N_{\text{leaf}}$ for leaves

For each node, do the following:
- Compute bounding box that fully encloses triangles & store
- Holds $\leq N_{\text{leaf}}$ triangles? Stop.
- Else, split into child groups
- Make one new node per group
- Set them as children of current
- Repeat with child nodes
BVH Building, Top-Down, $N_{leaf} = 4$
BVH Building, Top-Down, $N_{leaf} = 4$
BVH Building, Top-Down, $N_{leaf} = 4$
How to split a node?

- Which axes to consider for building bounding boxes/splitting?
  - Basis vectors $(1,0,0), (0,1,0), (0,0,1)$ only
  - Oriented basis vectors only
  - Arbitrary

- Where to split?
  - Spatial median
  - Object median
  - Something more elaborate...
How to split a node?

Which axes to consider for building bounding boxes/splitting?
- Basis vectors \((1,0,0), (0,1,0), (0,0,1)\) only
- Oriented basis vectors only
- Arbitrary

Where to split?
- Spatial median
- Object median
- Something more elaborate...

Algorithms exist (e.g. “separating axis theorem”), but usually very slow!
Splitting at spatial median

- Pick the longest axis (X/Y/Z) of current node bounds

- Find the midpoint on that axis

- Assign triangles to A/B based on which side of the midpoint each triangle’s centroid lies on

- Continue recursion with A/B
Splitting at spatial median

- Careful: can result in infinite recursion!

- All triangles are assigned again to one node, none in the other

- Can guard against it in several ways
  - Limit max. number of split attempts
  - Try other axes if one node is empty
  - Compute box over triangle centroids and split that on longest axis instead
Splitting at object median

- Pick an axis. Can try them all, don’t pick the same every time

- Sort triangles according to their centroid’s position on that axis

- Assign first half of the sorted triangles to A, the second to B

- Continue recursion with A/B

Rendering – Spatial Acceleration Structures
BVH Traversal

0. Set $t_{min} = \infty$. Start at root node, return if it doesn’t intersect ray.

1. Process node if its closest intersection with ray is closer than $t_{min}$

2. If it’s an inner node, run from 1. for child nodes that intersect ray
   - Process the closest node first
   - Keep others on stack to process further ones later (recursion works)

3. If it’s a leaf, check triangles and update $t_{min}$ in case of closer hit
1. Process node if its closest intersection with ray is closer than $t_{\text{min}}$
1. Process node if its closest intersection with ray is closer than $t_{\text{min}}$
2. If it’s an inner node, run from 1. for child nodes that intersect ray

- Process the closest node first
- Keep others on stack to process further ones later
1. Process node if its closest intersection with ray is closer than $t_{\text{min}}$
1. Process node if its closest intersection with ray is closer than $t_{min}$
3. If it’s a leaf, check triangles and update $t_{min}$ in case of closer hit.
3. If it’s a leaf, check triangles and update $t_{\text{min}}$ in case of closer hit
2. If it’s an inner node, run from 1. for child nodes that intersect ray:

- Process the closest node first
- Keep others on stack to process further ones later
1. Process node if its closest intersection with ray is closer than $t_{\text{min}}$
1. Process node if its closest intersection with ray is closer than $t_{\text{min}}$
3. If it’s a leaf, check triangles and update $t_{\text{min}}$ in case of closer hit (○)
3. If it’s a leaf, check triangles and update $t_{\text{min}}$ in case of closer hit (●)
3. If it’s a leaf, check triangles and update $t_{min}$ in case of closer hit (○)
3. If it’s a leaf, check triangles and update $t_{min}$ in case of closer hit (●).
3. If it's a leaf, check triangles and update $t_{min}$ in case of closer hit (●)
The Surface Area Heuristic [1]

- Simple, but powerful heuristic for choosing splits

- Created with traversal in mind, based on the following ideas:
  - Assume rays are uniformly distributed in space
  - Probability of a ray hitting a node is proportional to its surface area
  - Cost of traversing it depends on the number of triangles in its leaves
  - Hence, avoid large nodes with many triangles, because:
    - They have a tendency to get checked often
    - Getting a definite result (reject or closest hit) is likely to be expensive
Applying the Surface Area Heuristic

**Goal**: To split a node, find the hyperplane $b$ that minimizes

$$f(b) = LSA(b) \cdot L(b) + RSA(b) \cdot (N - L(b)),$$

where

- $LSA(b)/RSA(b)$ are the **surface area** of the nodes that enclose the triangles whose centroid is on the “left”/“right” of the split plane $b$

- $L(b)$ is the **number of primitives on the “left”** of $b$

- $N$ is the **total number of primitives** in the node
We want to constrain the search space for a good split.

Pick a set of axes to test (e.g., 3D basis vectors X/Y/Z).

When splitting a node with $N$ triangles, for each axis:
- Sort all triangles by their centroid’s position on that axis.
- Find the index $i$ that minimizes

$$f(i) = LSA(i) \cdot i + RSA(i) \cdot (N - i),$$

where
- $LSA(i)$ is the surface area of the AABB over sorted triangles $[0, i)$
- $RSA(i)$ is the surface area of the AABB over sorted triangles $[i, N)$

Select the axis and index $i$ with the best $f(i)$ for the split overall!
Importance of Optimizing Splits

- Important trade-off: building time vs. traversal time
  - Given the same tracing/traversal code, the quality of a BVH tree may have a big impact on performance!
  - Can be as high as 2x compared to naïve splitting

- Benefits depend on the parameters of your rendering scenario
  - How big is your scene and how are triangles distributed?
  - How long will your BVH be valid?
  - What are the quality requirements for your images?
Evaluation of Combined Building + Traversal [2]

Efficiency measured as a function of TOTAL WALLCLOCK TIME PER RAY, taking into account both BVH construction and actual tracing.

Check out the paper this comparison came from [https://users.aalto.fi/~ailat1/publications/karras2013hpg_paper.pdf](https://users.aalto.fi/~ailat1/publications/karras2013hpg_paper.pdf)
If you don’t have too many rays to trace, it probably pays off to construct BVH really quickly, even if tracing wasn’t as fast per ray.

Efficiency measured as a function of TOTAL WALLCLOCK TIME PER RAY, taking into account both BVH construction and actual tracing.
Evaluation of Combined Building + Traversal [2]

**After some point a faster but slower-to-build BVH’s increased tracing speed starts to pay off**

Efficiency measured as a function of TOTAL WALLCLOCK TIME PER RAY, taking into account both BVH construction and actual tracing.

Check out the paper this comparison came from [https://users.aalto.fi/~ailat1/publications/karras2013hpg_paper.pdf](https://users.aalto.fi/~ailat1/publications/karras2013hpg_paper.pdf)
<table>
<thead>
<tr>
<th>Structure</th>
<th>Memory Consumption</th>
<th>Building Time</th>
<th>(Expected) Traversal Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>none</td>
<td>abysmal</td>
</tr>
<tr>
<td>Regular Grid</td>
<td>low – high (resolution)</td>
<td>low</td>
<td>uniform scene: ok otherwise: bad</td>
</tr>
<tr>
<td>Quadtree/Octree</td>
<td>low – high (overlap/uniformity)</td>
<td>low</td>
<td>good</td>
</tr>
<tr>
<td>K-d Tree</td>
<td>low – high (overlap)</td>
<td>low – high</td>
<td>good – excellent</td>
</tr>
<tr>
<td>BVH</td>
<td>low</td>
<td>low – high</td>
<td>good – excellent</td>
</tr>
</tbody>
</table>
For each split, sort the node’s portion of the triangle list $L$ in-place.

When constructing child nodes, pass them $L$ and `start/end` indices.
For each split, sort the node’s portion of the triangle list $L$ in-place.

When constructing child nodes, pass them $L$ and *start/end* indices.
BVH Building Hints

- For each split, sort the node’s portion of the triangle list $L$ in-place.

- When constructing child nodes, pass them $L$ and `start/end` indices.
SAH Coding Hints

- Don’t loop over triangles at each $i$ to get $LSA(i)$ and $RSA(i)$!

- Precompute them once per node and axis instead
  - Create two 0-volume bounding boxes $BB_L, BB_R$
  - Allocate N+1 entries for $LSA/RSA$, set $LSA(0) = RSA(N) = 0$
  - Iterate $i$ over range $[1, N]$, for each $i$:
    - Merge $BB_L$ with the AABB of sorted triangle with index $(i - 1)$
    - Store surface area of $BB_L$ as value for $LSA(i)$
    - Merge $BB_R$ with the AABB of sorted triangle with index $(N - i)$
    - Store surface area of $BB_R$ as value for $RSA(N - i)$
Consider using `stdlib` container (e.g., vector)
- Try to avoid dynamic memory allocation
- $2N - 1$ is an upper bound for the total number of nodes you need

```cpp
std::sort(<first>, <last>, <predicate>)
```

```cpp
std::nth_element(<first>, <nth>, <last>, <predicate>)
```

- Can be used for splitting if you don’t need exact sorting
- Reorders the $N$-sized vector such that:
  - $n$ smallest elements are on the left
  - $N - n$ biggest are on the right
- Faster than sorting!
Each have their specializations, strengths and weaknesses

E.g., K-d Trees with ropes do not require a stack for traversal [5]

Which acceleration structure is the best is contentious

Currently, BVHs are extremely widespread and well-understood
State-of-the-Art Variants and Trends

- Higher child counts (>2) per node, mixed nodes (children + triangles)
- Actually DO split triangles sometimes to get maximal performance
- Build BVHs bottom-up in parallel on the GPU [3]
- In animated scenes, reuse BVHs, update those parts that change
- Actually use built-in traversal logic of GPU hardware (NVIDIA RTX!)
References and Further Reading

- Interesting topics: BVHs for animation, LBVH, SIMD/packet/stackless traversal, Turing RTX architecture


