Efficient octree traversal

for real-time path tracing systems

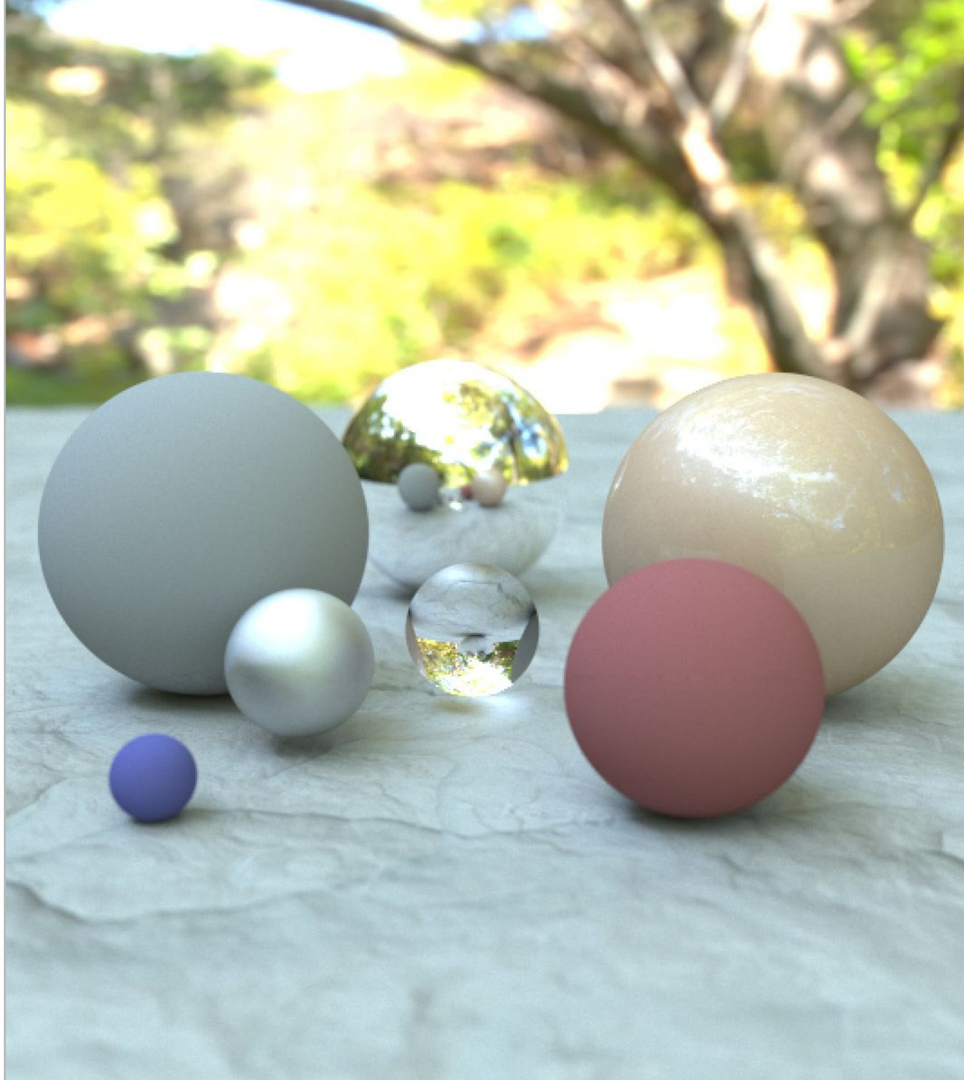
Octrees are one of my favorite spatial data structures because of their *simplicity* and *efficiency* — they're easy to understand and visualize, and can significantly improve the performance of a task.

The Final Stage path tracer uses octrees to reduce rendering time by about **100x**, so I thought it might be interesting to explore how this was achieved.

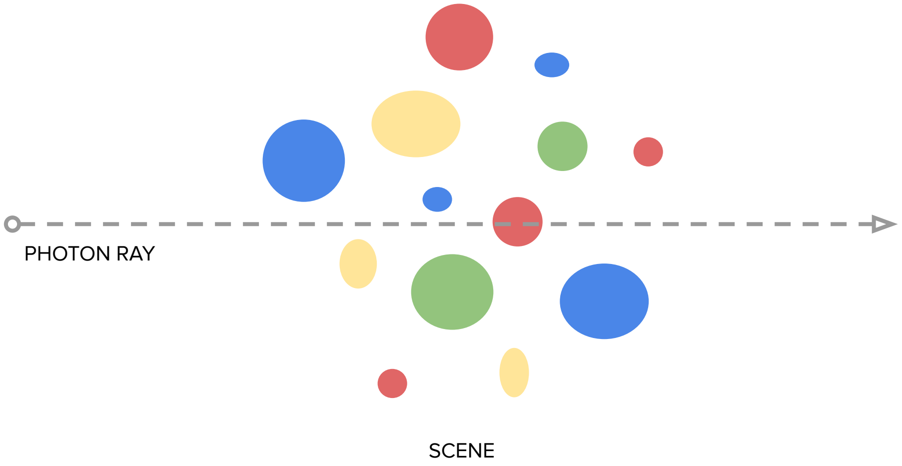
THE PROBLEM

Photorealistic rendering algorithms often require us to determine ***where a photon ray first collides with objects in our scene*** *(aka ray-scene collision detection)*.

Modern renderers perform this kind of collision detection *hundreds of billions of times* ***to generate a single frame,*** so performance is incredibly important.



Specifically, given a collection of objects that comprise our scene, and a photon travelling in a straight line (i.e. a photon ray), ***determine the first object, if any, that is struck by the photon***.



Result: the first object struck by the photon ray is a small **red sphere**.

Things we definitely *don't* want to do:

- ✘ Test our photon ray against every object in the scene
- ✘ Test our photon ray against every polygon of every object
- ✘ Rely solely on non-spatial data structures to represent our scene

Doing any of these will ***severely impact*** *performance!*

Our solution needs to scale to support scenes with an arbitrary number of objects, and an arbitrary number of polygons. Ideally, *memory* should be the limiting factor, not *processing power.*

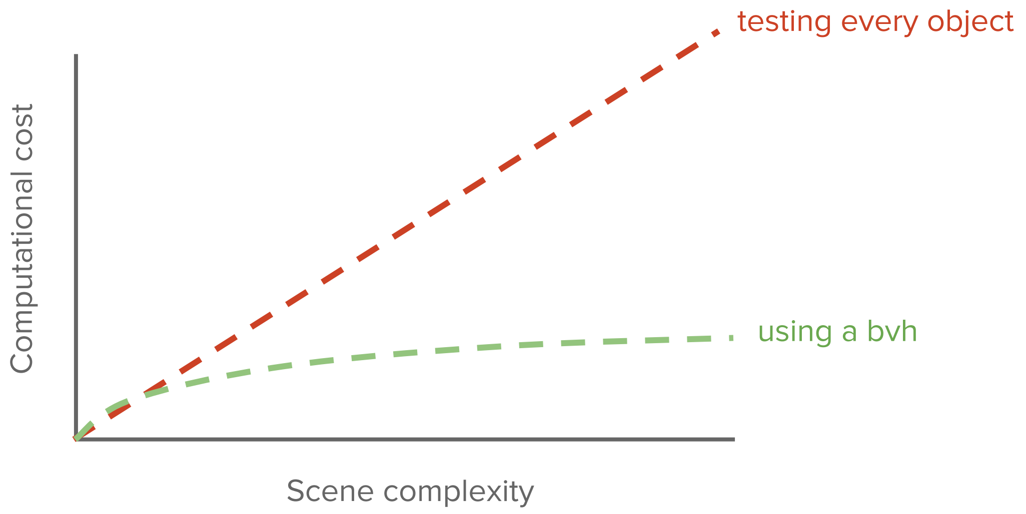
So how do we optimize our ray-scene collision detection?

Bounding volume hierarchies

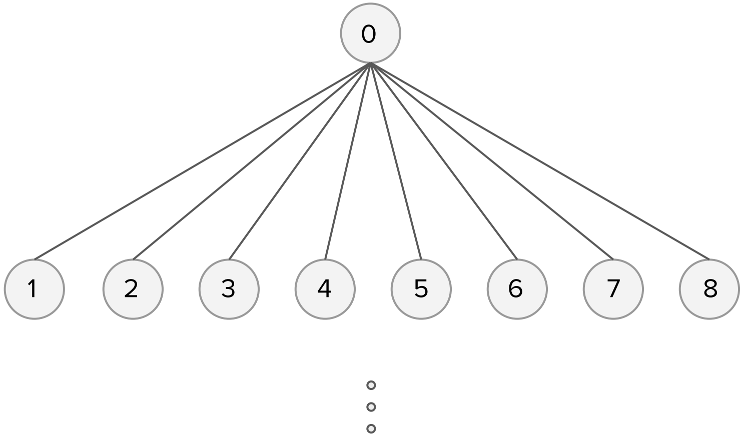
We divide the scene into a ***hierarchy of spatial regions*** that allow us to disregard large groups of objects when we know that the photon will not pass through their vicinity (aka bounding volume).

There are many different types of BVHs, including octrees, bsp trees, quadtrees, kd trees, and many more. ***We're going to focus on octrees.***

Bounding volume hierarchies give us **logarithmic computational cost** as scene complexity increases. Approaches that consider every object will be linear at best!

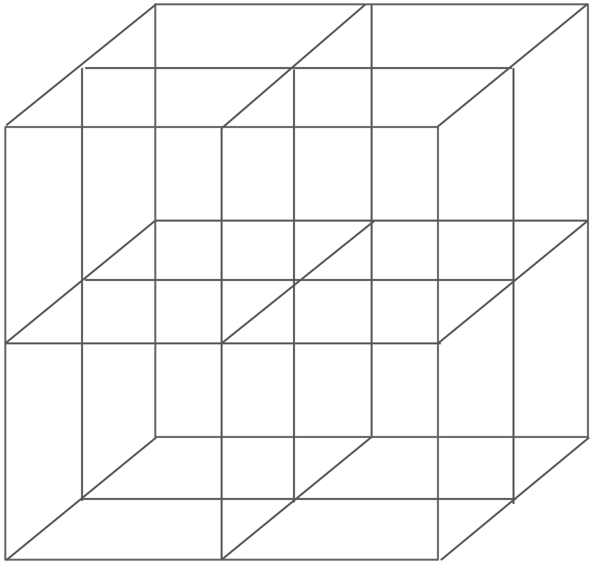


Data structure
Each node references up to 8 children



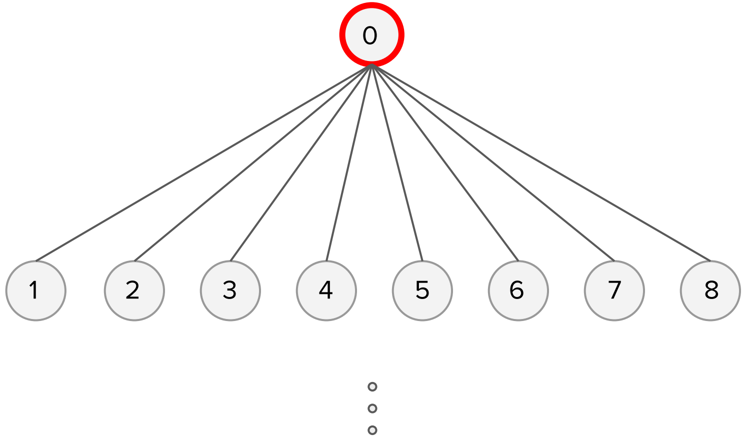
Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned
sub-volume of the parent node



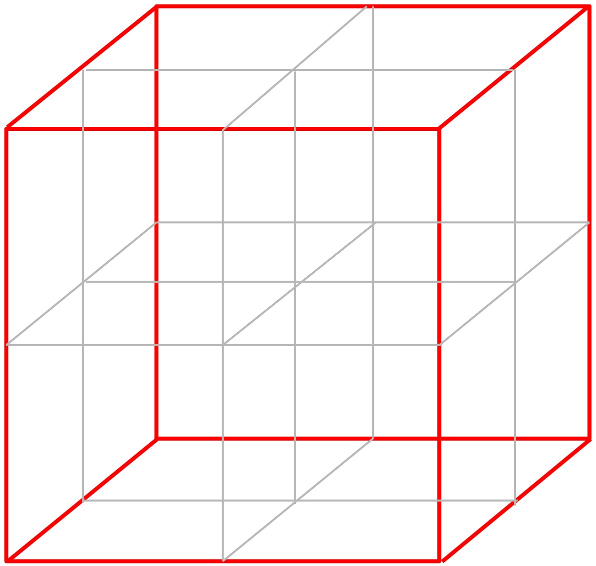
OCTREE STRUCTURE

Data structure
Each node references up to 8 children



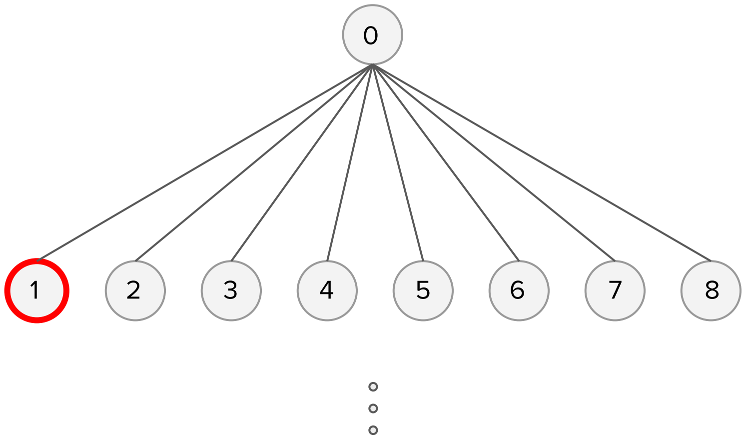
Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned
sub-volume of the parent node



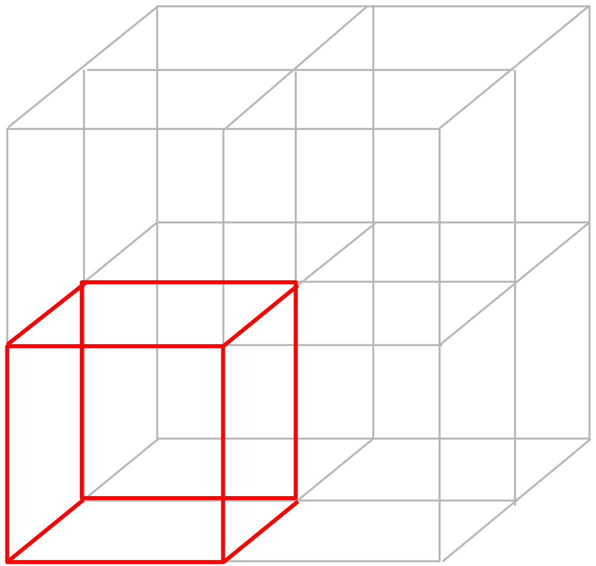
OCTREE STRUCTURE

Data structure
Each node references up to 8 children

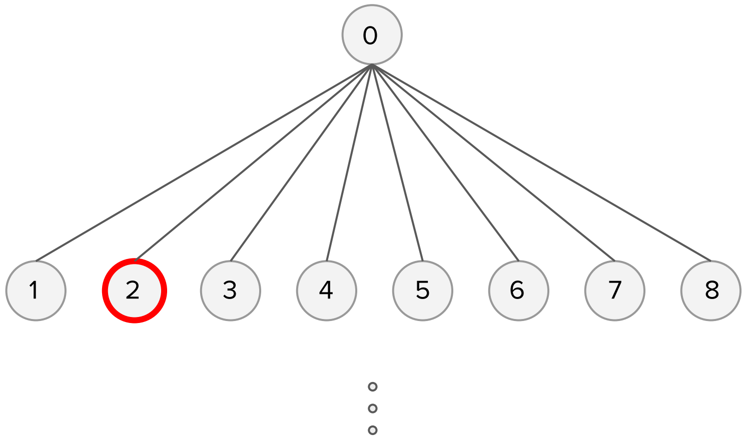


Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned
sub-volume of the parent node

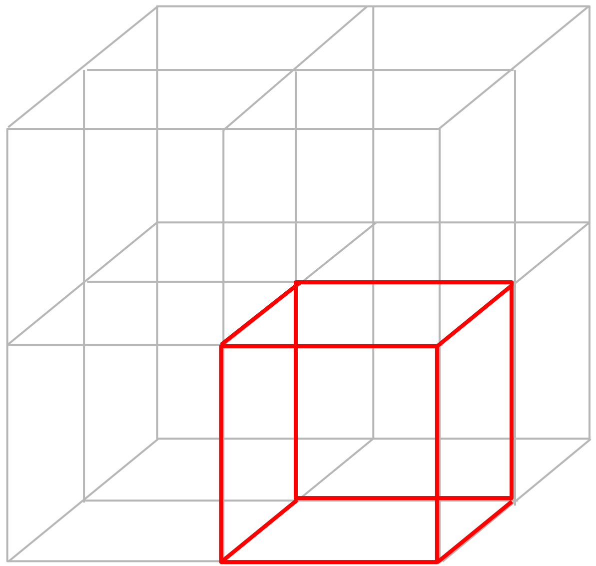


Data structure
Each node references up to 8 children



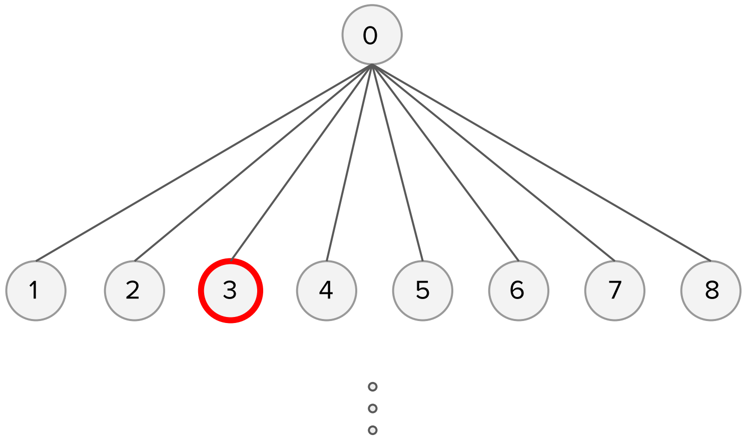
Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned sub-volume of the parent node



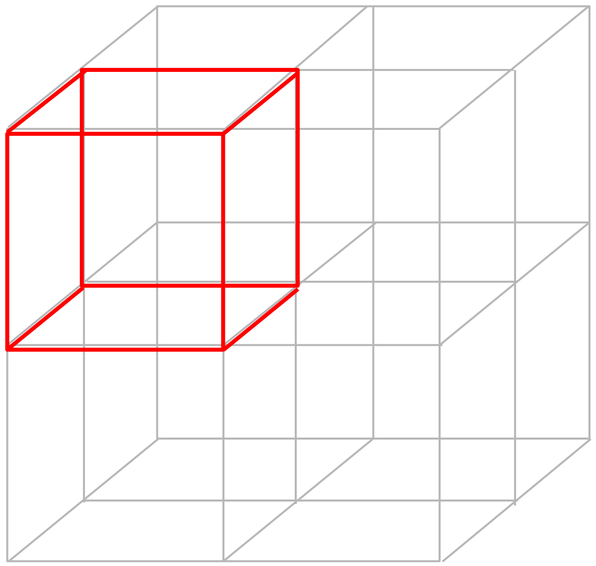
OCTREE STRUCTURE

Data structure
Each node references up to 8 children



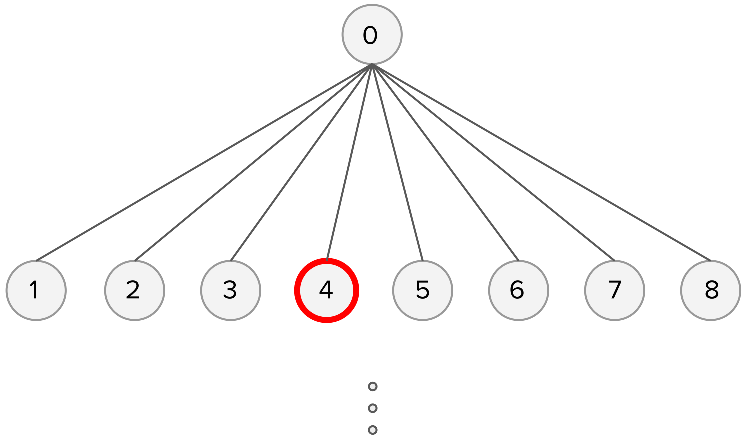
Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned
sub-volume of the parent node



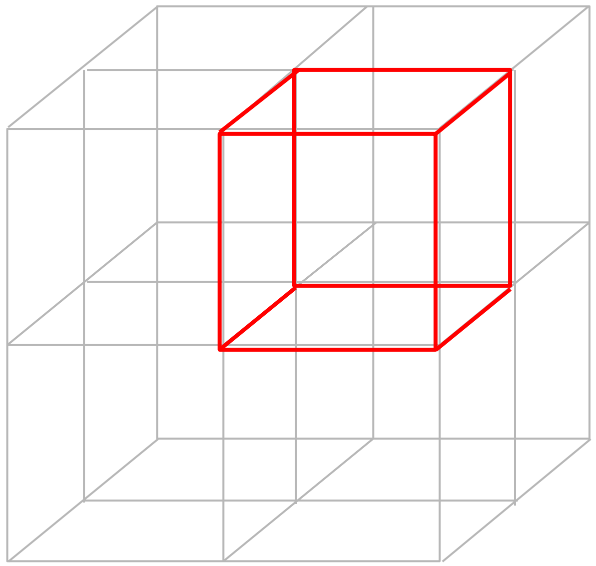
OCTREE STRUCTURE

Data structure
Each node references up to 8 children



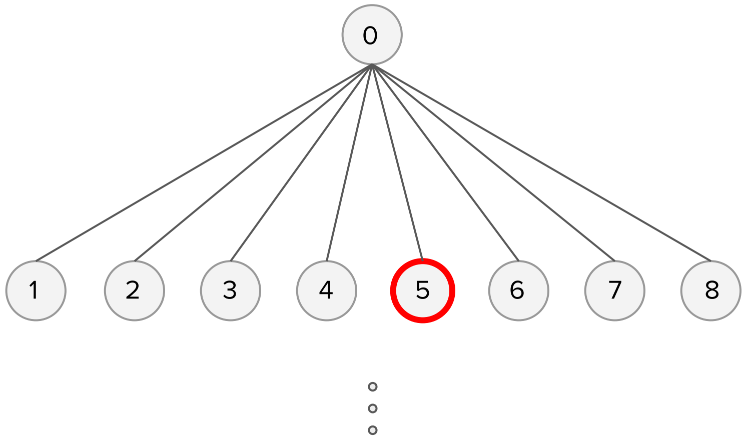
Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned sub-volume of the parent node



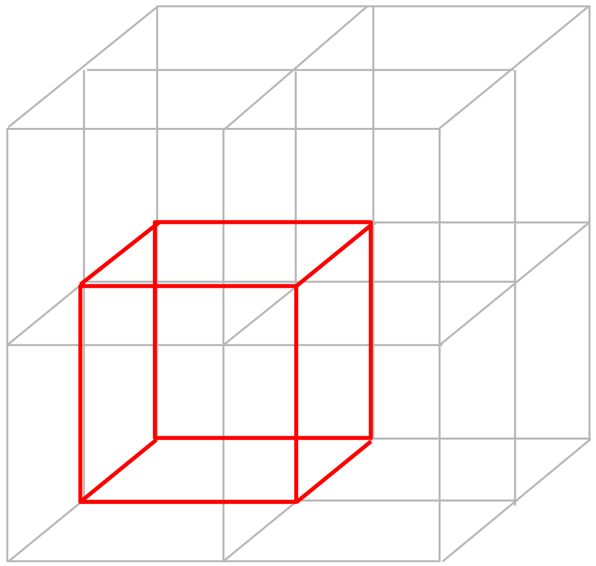
OCTREE STRUCTURE

Data structure
Each node references up to 8 children



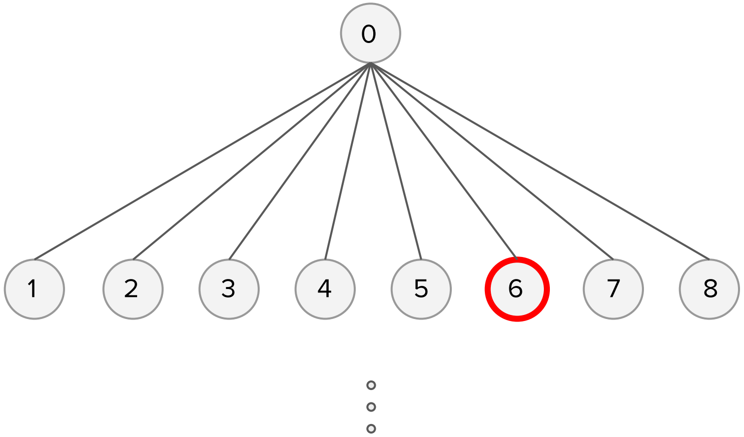
Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned
sub-volume of the parent node



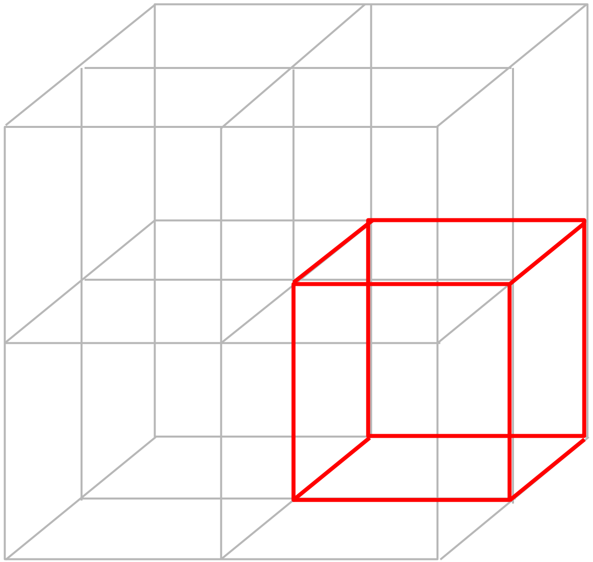
OCTREE STRUCTURE

Data structure
Each node references up to 8 children



Tree depth restricted to some maximum depth
(e.g. d=32)

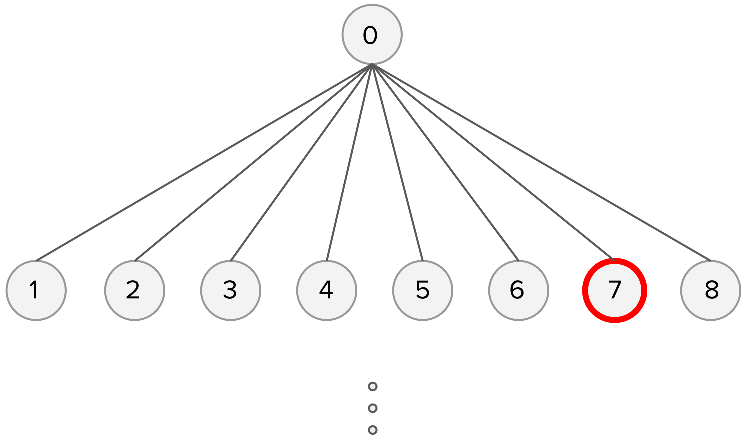
Spatial structure
Each node represents an axis aligned sub-volume of the parent node



bertolami.com

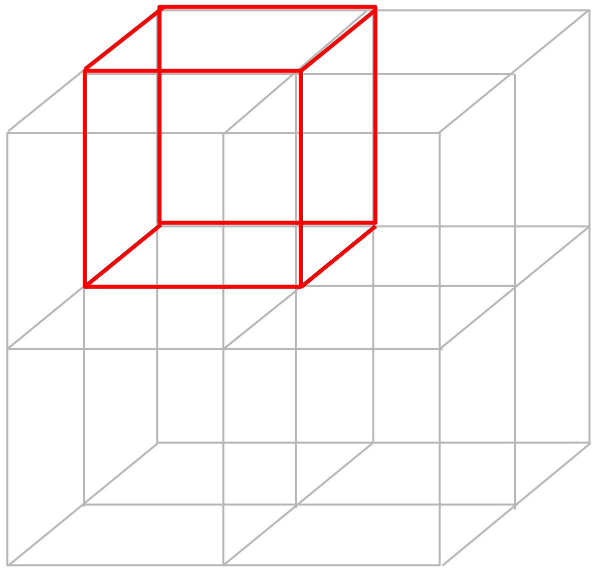
OCTREE STRUCTURE

Data structure
Each node references up to 8 children



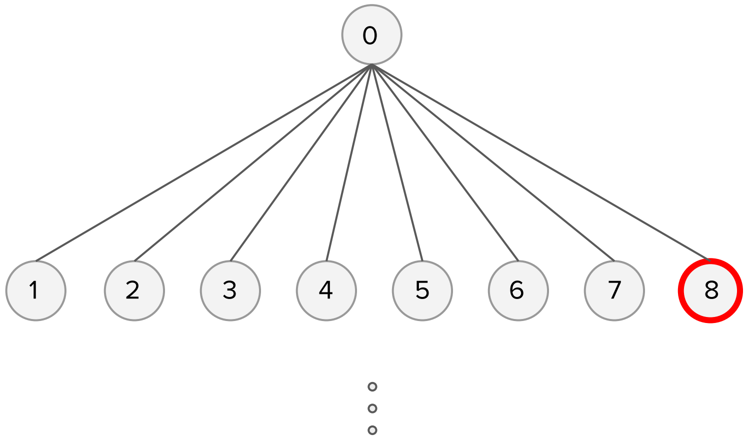
Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned
sub-volume of the parent node



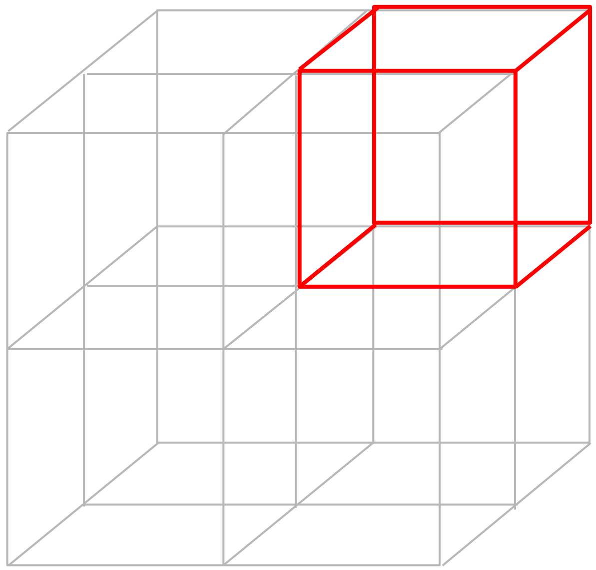
OCTREE STRUCTURE

Data structure
Each node references up to 8 children



Tree depth restricted to some maximum depth
(e.g. d=32)

Spatial structure
Each node represents an axis aligned
sub-volume of the parent node

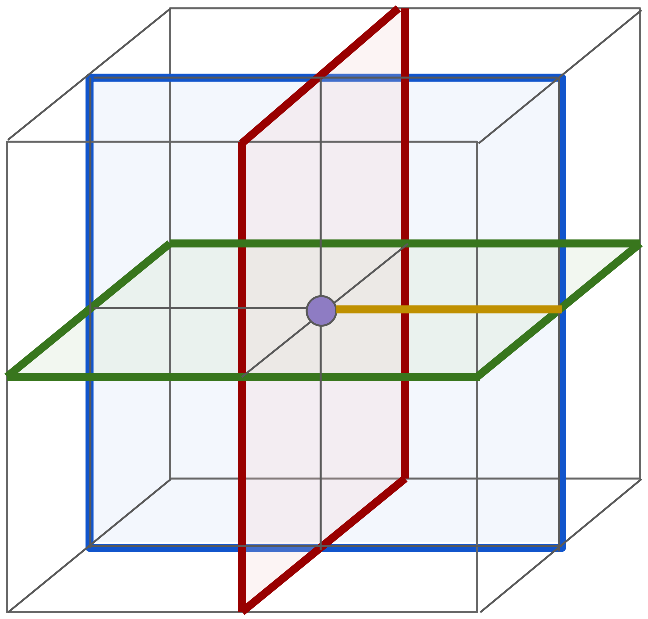


Node description

Each node boundary can be described by a ***center point*** and a ***span***.

For convenience we also define three axis aligned dividing planes (***YZ***, ***XZ***, ***XY***).

We'll use these to quickly determine intersections between our ray and the nearest child volume.



Thus, our goal is to create an Octree class that supports the following operations:

Construction: `Octree::Initialize(scene)`

Initializes our octree object with scene data. Performed once during setup, or whenever the scene changes.

Traversal: `Octree::Trace(ray, collision)`

Efficiently checks for collisions between the ray and our scene. Stores information within *collision parameter* about the closest collision found, if any. Performed many times per frame.

OCTREE CONSTRUCTION

Initialization:

Constructs the root node and then kicks off recursive subdivision to initialize children.

Subdivision:

Constructs the spatial hierarchy such that:

- Each node has exactly zero or eight children
- Parent bounds is the union of its child node bounds
- Polygons are only referenced by leaf nodes

Halt subdivision when node volume becomes too small, node manages a small number of polygons, or we reach maximum tree depth.

Optimizations:

- Avoid empty child nodes
- Split polygons along node boundaries
- Multi-threaded construction
- Be careful with memory management of polygons

```
void Octree::Initialize(scene) {
  allocate a root_node

  for each polygon in the scene:
    add the polygon to the root_node
    if any of polygon's vertices are outside of root bounds:
      expand root bounds to cover it

  call root_node.Subdivide(0)
}


void OctreeNode::Subdivide(depth) {
  if depth >= maximum allowable depth or
      bounding volume is too small or
      total polycount in node bounds < minimum polycount
    return

  for each child node index:
    create the node and set its bounding volume equal to
    the respective sub-volume of the current node.

  for each polygon in the current node:
    add it to any children that it intersects, or optionally
    split polygons along node boundaries and add pieces
    to respective child nodes

  clear all polygons out of the current node

  for each child node:
    call Subdivide on the child node
}
```

Attempt 1: naïve top/down traversal

Recursively test nodes, beginning with the root. If a collision is detected against the current node bounds, recursively test every child. Maintain a running knowledge of the closest collision found.

TOP/DOWN TRAVERSAL

Conditions:

A node is considered a leaf if it contains any polygons. Non-leaf nodes will not reference any polygons.

Process:

Start at the root and check it's children only if the ray intersects the root bounds. Recursively repeat this process for children, and keep track of the best collision (closest to the ray origin) encountered.

Issues:

- We traverse child nodes in a fixed order
- We check every child, even if the nearest collision has already been found

```
void Octree::Trace(ray, collision) {
  root_node.Trace(ray, collision)
}

void OctreeNode::Trace(ray, collision) {
  if ray does not intersect node bounds:
    return

  if a collision has already been detected and it's closer
      than the ray entry point into this node:
    return

  if node is a leaf:
    for each polygon in node:
      if ray intersects polygon:
        if intersection is closer to ray origin than collision:
          update collision with current intersection
  else:
    for each child_node:
       child_node.Trace(ray, collision)
}
```

Attempt 2: top/down with distance sorted siblings

Recursively test nodes, beginning with the root. If collision detected against current node bounds, recursively test each child using front-to-back order from the ray origin.

Halt the entire process once a collision is found.

SORTED SIBLING TRAVERSAL

Process:

Start at the root and check it’s children only if the ray intersects the root bounds.

When checking non-leaf children, begin with the child closest to the ray origin, and then test progressively farther child nodes. If a collision is detected within a leaf node, halt the entire process.

We check at most 4 children of any given node. If a collision is not detected in the nearest 4 nodes, there won’t be a collision in the remaining 4 either.

But how do we know the order to process child nodes?

```
void Octree::Trace(ray, collision) {
  root_node.Trace(ray, collision)
}

void OctreeNode::Trace(ray, collision) {
  if ray does not intersect node bounds:
    return

  if a collision has already been detected and it’s closer
      than the ray entry point into this node:
    return

  if node is a leaf:
    for each polygon in node:
      if ray intersects polygon:
        if intersection is closer to ray origin than collision:
          update collision with current intersection
  else:
    for i = 0, i < 4, ++i:
      determine nearest untested node to the ray origin
      call node.Trace(ray, collision)
      if collision detected:
        break
}
```

Two key steps for determining the proper traversal order of child nodes:

- ❑ Finding the closest child node to a given point (often the ray origin)
- ❑ Finding the next nearest untested child node

Cases to consider:

- *Does the ray originate from outside of the parent node?*
- *Is the ray pointing away from all child nodes?*
- *Does the ray graze a node boundary?*

If these are both true, the ray will not intersect any child

Finding the nearest child relative to a point:

Recall that we defined our nodes as the combination of a point, a span, and three planes. If we orient our point *(which is often the ray origin)* relative to the node center, then we can quickly determine which child node is nearest by comparing the oriented point with the basis x, y, and z planes.

In other words, for the oriented point [x', y', z']:

- If x >= 0, then it is closest to a positive x child node
- If y >= 0, then it is closest to a positive y child node
- If z >= 0, then it is closest to a positive z child node

The combination of these boolean results uniquely identifies which node must be closest to the point.

```
int OctreeNode::ClosestChild(point) {
  oriented_point = point - node_center

  x_test = oriented_point.x >= 0.0
  y_test = oriented_point.y >= 0.0
  z_test = oriented_point.z >= 0.0

  return x_test | (y_test << 1) | (z_test >= 0.0 << 2)
}
```

Finding the nearest child relative to a point:

Recall that we defined our nodes as the combination of a point, a span, and three planes. If we orient our point *(which is often the ray origin)* relative to the node center, then we can quickly determine which child node is nearest by comparing the oriented point with the basis x, y, and z planes.

In other words, for the oriented point [x', y', z']:

- If x >= 0, then it is closest to a positive x child node
- If y >= 0, then it is closest to a positive y child node
- If z >= 0, then it is closest to a positive z child node

The combination of these boolean results uniquely identifies which node must be closest to the point.

```
int OctreeNode::ClosestChild(point) {
  oriented_point = point - node_center

  x_test = oriented_point.x >= 0.0
  y_test = oriented_point.y >= 0.0
  z_test = oriented_point.z >= 0.0

  return x_test | (y_test << 1) | (z_test >= 0.0 << 2)
}
```

Packaging the response

For convenience, we combine the results into an index that matches the order that we allocated child nodes in the tree. The following table describes this mapping:

Binary value	Quadrant
000	-X, -Y, -Z
001	+X, -Y, -Z
010	-X, +Y, -Z
011	+X, +Y, -Z
...	...
111	+X, +Y, +Z

SORTED SIBLING TRAVERSAL

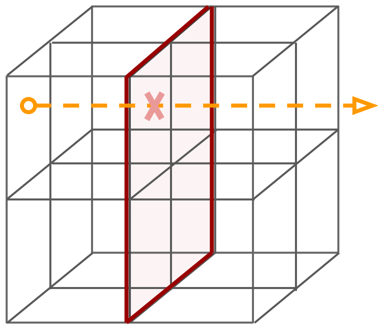
Finding the *next nearest* sibling to check

We check for collisions between the ray and each of the parent node's three axis planes. The nearest plane that is struck will indicate the entry point of the ray into the next nearest sibling.

If no plane is hit, then the ray is headed out of the node and will not hit any other children of the current parent.

Example:

A ray exiting the (-X,+Y,-Z) quadrant with a collision against the YZ plane will enter the (+X,+Y,-Z) quadrant as the next nearest sibling node in our traversal.



```
closest_node_index = ClosestChild(ray.origin)

plane_hit[0] = result of ray intersection test against YZ plane
plane_hit[1] = result of ray intersection test against XZ plane
plane_hit[2] = result of ray intersection test against XY plane

for i = 0, i < 4, ++i:
  if child_node at closest_node_index is valid:
    call child_node.Trace(ray, collision)
    if collision detected:
      break;

  plane_index = index of closest valid plane_hit collision

  if there are no valid plane collisions or
     closest plane collision point is outside parent bounds:
    break;

  closest_node_index ^= 0x1 << plane_index
  invalidate plane_hit[plane_index]
```

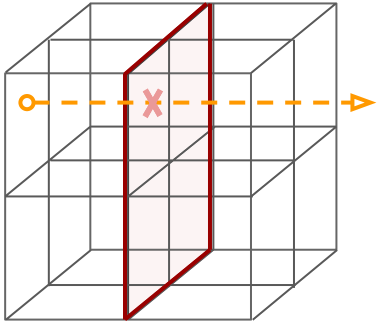
Finding the *next nearest* sibling to check

We check for collisions between the ray and each of the parent node's three axis planes. The nearest plane that is struck will indicate the entry point of the ray into the next nearest sibling.

If no plane is hit, then the ray is headed out of the node and will not hit any other children of the current parent.

Example:

A ray exiting the (-X,+Y,-Z) quadrant with a collision against the YZ plane will enter the (+X,+Y,-Z) quadrant as the next nearest sibling node in our traversal.




```
closest_node_index = ClosestChild(ray.origin)

plane_hit[0] = result of ray intersection test against YZ plane
plane_hit[1] = result of ray intersection test against XZ plane
plane_hit[2] = result of ray intersection test against XY plane

for i = 0, i < 4, ++i:
  if child_node at closest_node_index is valid:
    call child_node.Trace(ray, collision)
    if collision detected:
      break;

  plane_index = index of closest valid plane_hit collision

  if there are no valid plane collisions or
     closest plane collision point is outside parent bounds:
    break;

  closest_node_index ^= 0x1 << plane_index
  invalidate plane_hit[plane_index]
```

Configuring the next nearest child node

If we get here, it means the child_node did not have a collision. We update closest_node_index to indicate the next nearest sibling node and disqualify the current child from future consideration.

Putting it all together:

- Skip nodes that do not intersect the ray
- If the node is a leaf, test its polygons
- If the node is not a leaf recursively test up to 4 child nodes that are nearest to the ray origin
- Halt the entire process the moment a collision is detected, or the ray exits the parent bounds.

Optimizations:

- Perform ray-plane intersection tests only if a collision isn't found within the nearest child node.

```
void OctreeNode::Trace(ray, collision) {
  if ray does not intersect node bounds:
    return

  if a collision has already been detected and it's closer
      than the ray entry point into this node:
    return

  if node is a leaf:
    for each polygon in node:
      if ray intersects polygon:
        if intersection is closer to ray origin than collision:
          update collision with current intersection
  else:
    closest_node_index = ClosestChild(ray.origin)

    plane_hit[0] = ray intersection test against YZ plane
    plane_hit[1] = ray intersection test against XZ plane
    plane_hit[2] = ray intersection test against XY plane

    for i = 0, i < 4, ++i:
      if child_node at closest_node_index is valid:
        call child_node.Trace(ray, collision)
        if collision detected:
          break;

      plane_index = index of closest valid plane_hit collision

      if there are no valid plane collisions or
         closest plane collision point is outside parent bounds:
        break;

      closest_node_index ^= 0x1 << plane_index
      invalidate plane_hit[plane_index]
}
```

Final Stage 2.0 uses nearest neighbor octree traversal. This enabled a 10x improvement over naïve traversal, and a 100x improvement over an initial non-BVH solution.

	No BVH	Simple traversal	Sorted sibling traversal
Render time	1,000 ms	100 ms	10 ms

Bounding volume hierarchies can **significantly improve the performance** of spatial search operations. Commonly used in rendering, but also applicable elsewhere.

Lots of libraries from Intel, Nvidia, AMD that do the heavy lifting for you.

Thanks for listening (or reading)!

Don’t forget to check out the source for Final Stage 2.0, which demonstrates most of the concepts discussed in this talk.