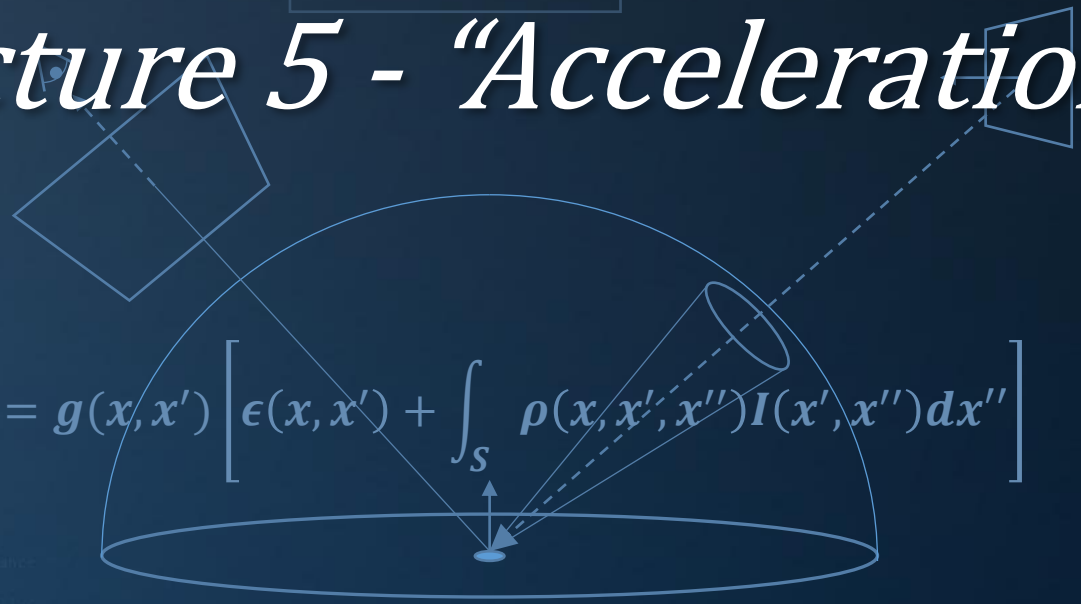
# INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

# *Lecture 5 - "Acceleration Structures"*



$$I(x,x') = g(x,x')\left[\epsilon(x,x') + \int_S \rho(x,x',x'')I(x',x'')dx''\right]$$

## Welcome!

# Today’s Agenda:

- Problem Analysis
- Early Work
- BVH Up Close

# Analysis

“Cornell Box”

Voxel game

# Analysis

Unreal 5 Tech Demo

Avengers Endgame

# Analysis

## Characteristics

### Rasterization:

- Games
- Fast
- Realistic
- Consumer hardware

### Ray Tracing:

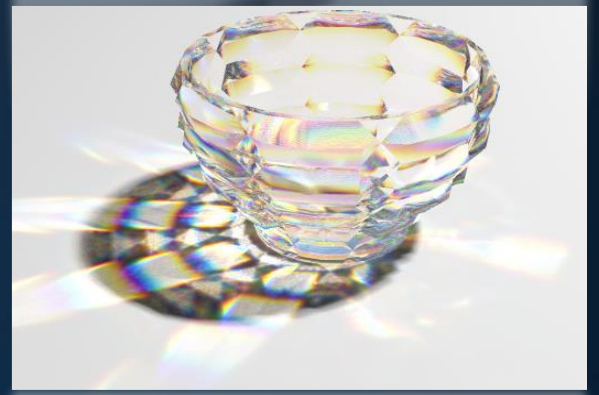
- Movies
- Slow
- Very Realistic
- Supercomputers


Heaven7, Exceed, 2000


LOTR: The Return of the King, 2003



Mirror's Edge, DICE, 2008



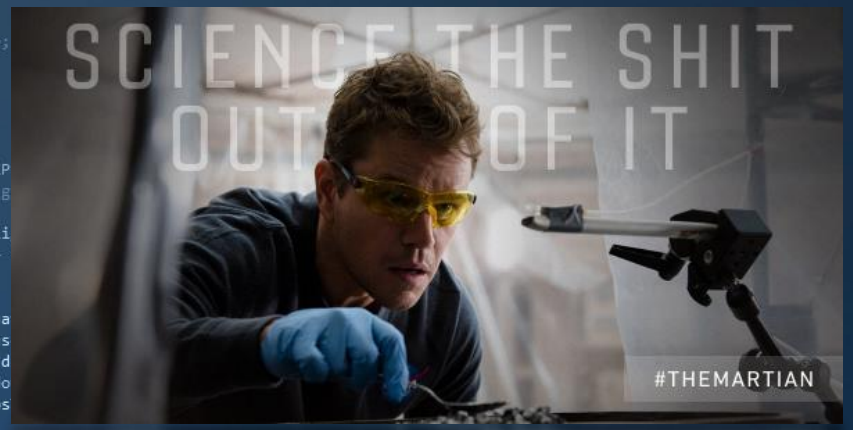

Crysis, 2007

# Analysis

## Characteristics

Reality:

- everyone has a budget
- bar must be raised
- *we need to optimize*.




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)

# Analysis

Optimizing Ray Tracing

Options:

1. Faster intersections (reduce constant cost)
2. Faster shading (reduce constant cost)
3. Use more expressive primitives (trade constant cost for algorithmic complexity)
4. Fewer of ray/primitive intersections (reduce algorithmic complexity)

Note for option 1:

At 5 billion ray/primitive intersections, we will have to bring down the cost of a single intersection to 1 cycle on a 5Ghz CPU – if we want one frame per second.



Crysis, 2007

# Today’s Agenda:

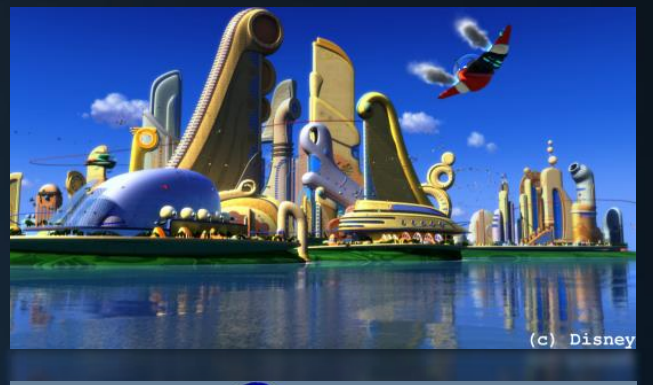
- Problem Analysis
- Early Work
- BVH Up Close

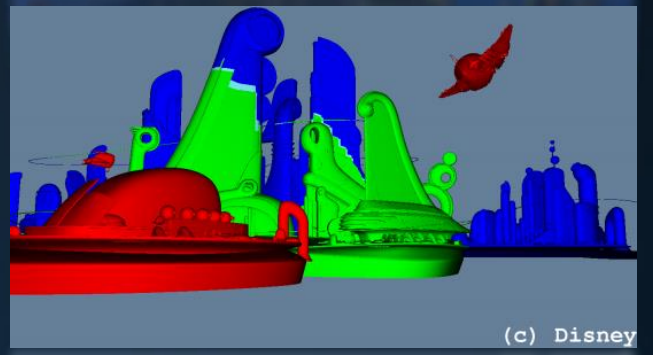
# Early Work

Complex Primitives

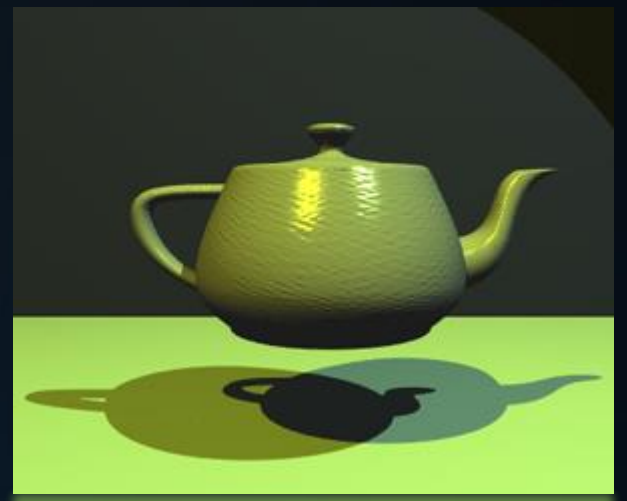
More expressive than a triangle:

- Sphere
- Torus
- Teapotahedron
- Bézier surfaces
- Subdivision surfaces*
- Implicit surfaces**
- Fractals***




Meet the Robinsons, Disney, 2007



Utah Teapot, Martin Newell, 1975

*: Benthin et al., Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces. 2007.
**: Knoll et al., Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic. RT'07 Proceedings, Pages 11-18
***: Hart et al., Ray Tracing Deterministic 3-D Fractals. In Proceedings of SIGGRAPH '89, pages 289-296.

# Early Work

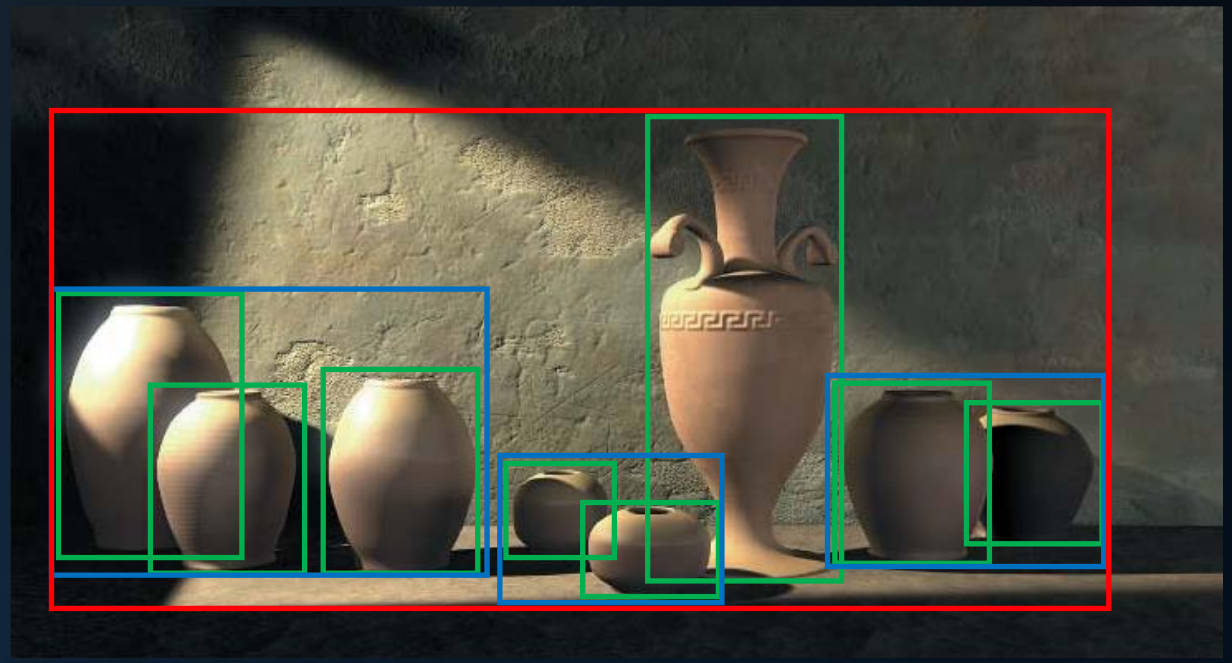
Rubin & Whitted*

*"Hierarchically Structured Subspaces"*

Proposed scheme:

- Manual construction of hierarchy
- Oriented parallelepipeds

A transformation matrix allows efficient Intersection of the skewed / rotated boxes, which can tightly enclose actual geometry.



*: S. M. Rubin and T. Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In: Proceedings of SIGGRAPH '*80*, pages 110–116.

# Early Work

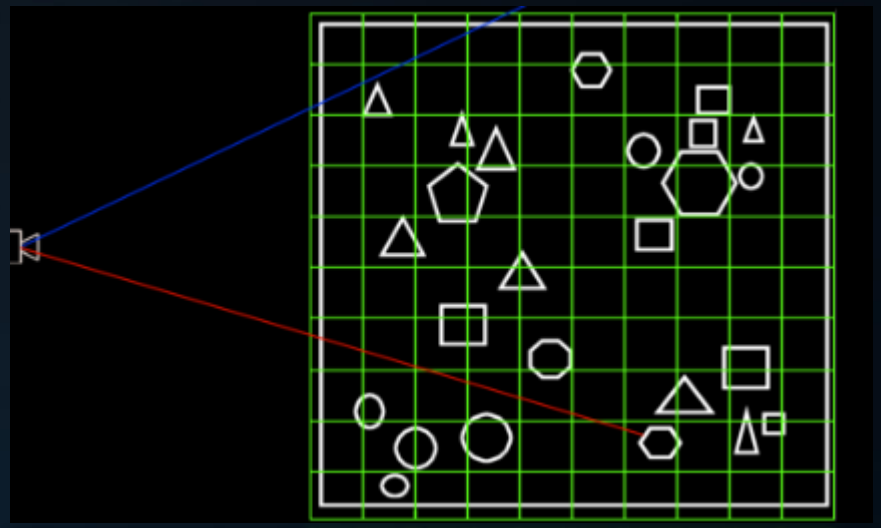
Amanatides & Woo*

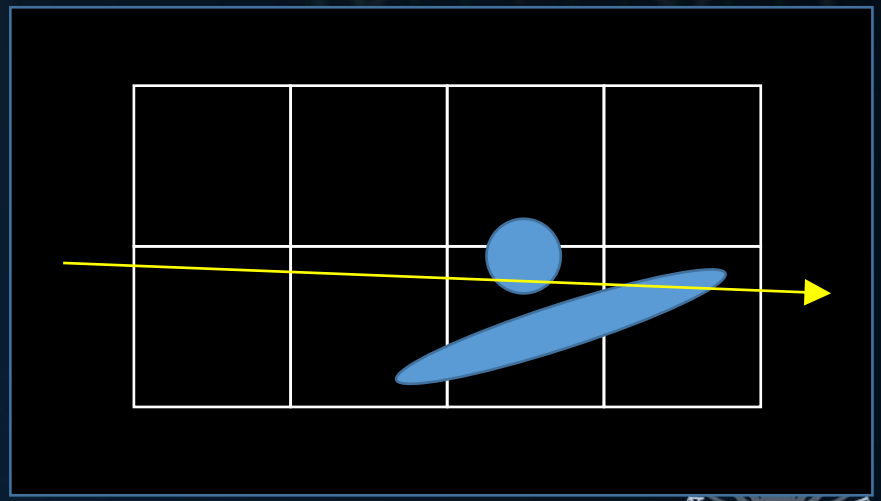
*"3DDDA of a regular grid"*

The grid can be automatically generated.

Considerations:

- Ensure that an intersection happens in the current grid cell
- Use mailboxing to prevent repeated intersection tests





*: J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In Eurographics '*87*, pages 3–10, 1987.

# Early Work

Glassner*

*"Hierarchical spatial subdivision"*

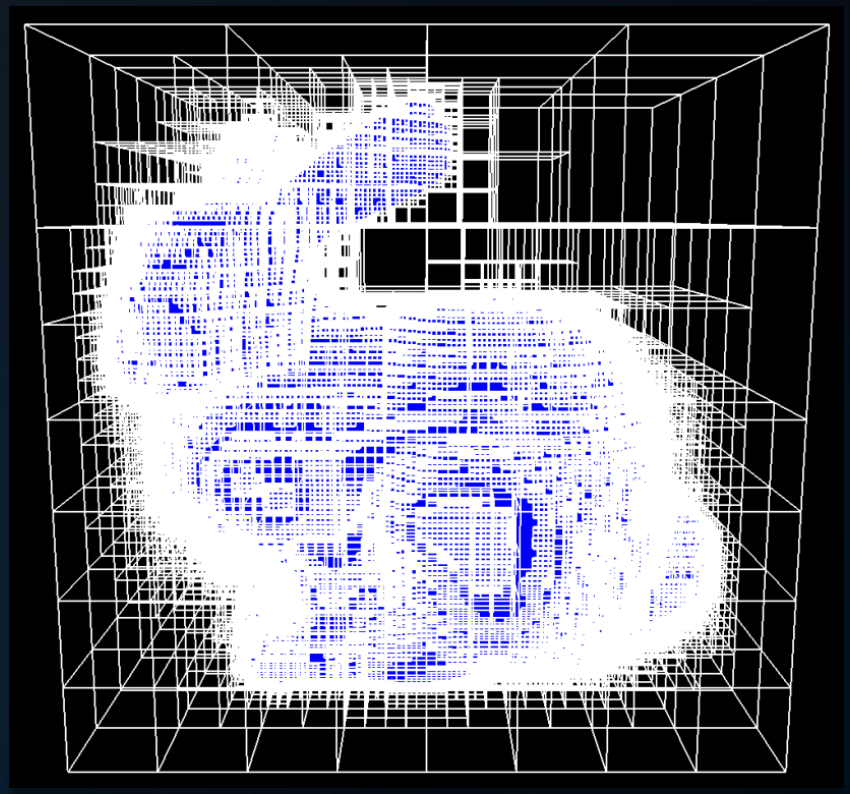
Like the grid, octrees can be automatically generated.

Advantages over grids:

- Adapts to local complexity: fewer steps
- No need to hand-tune grid resolution

Disadvantage compared to grids:

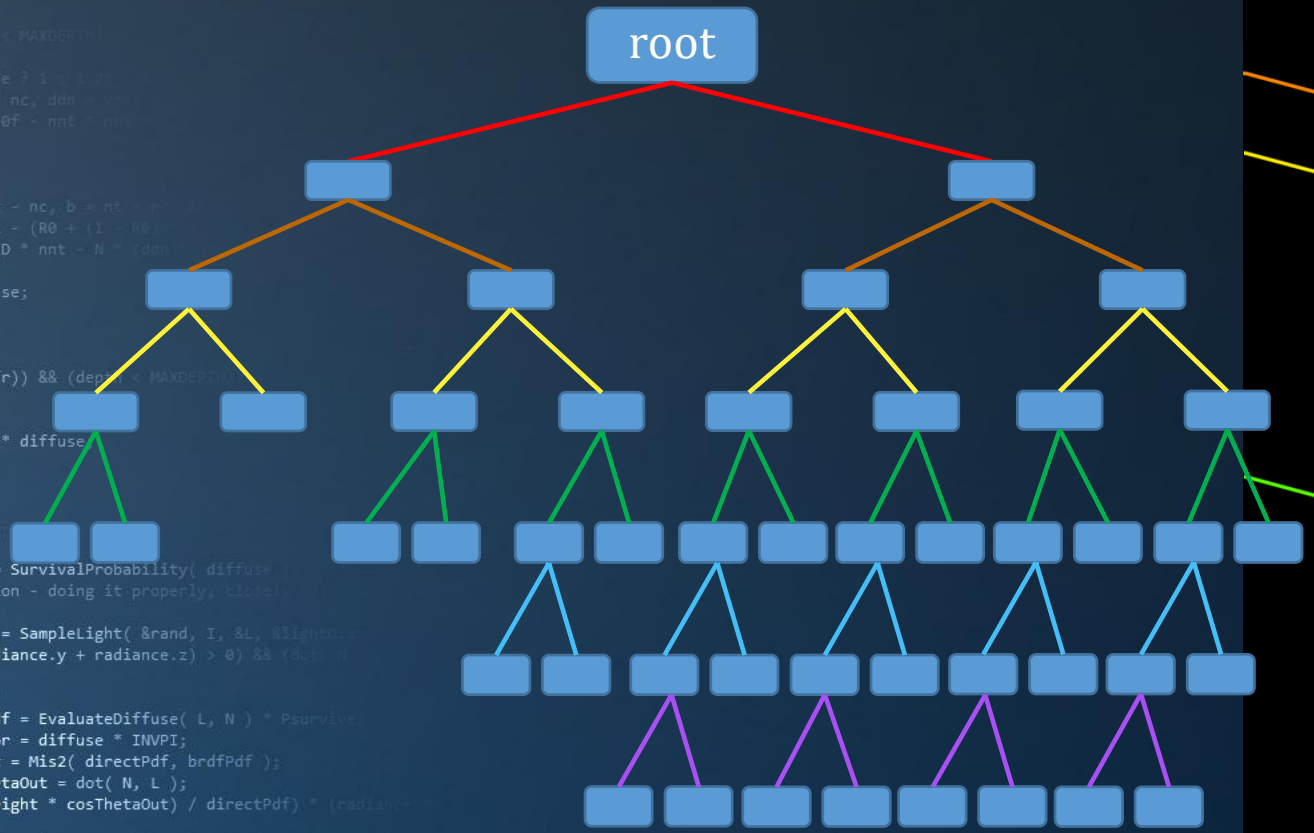
- Expensive traversal steps.

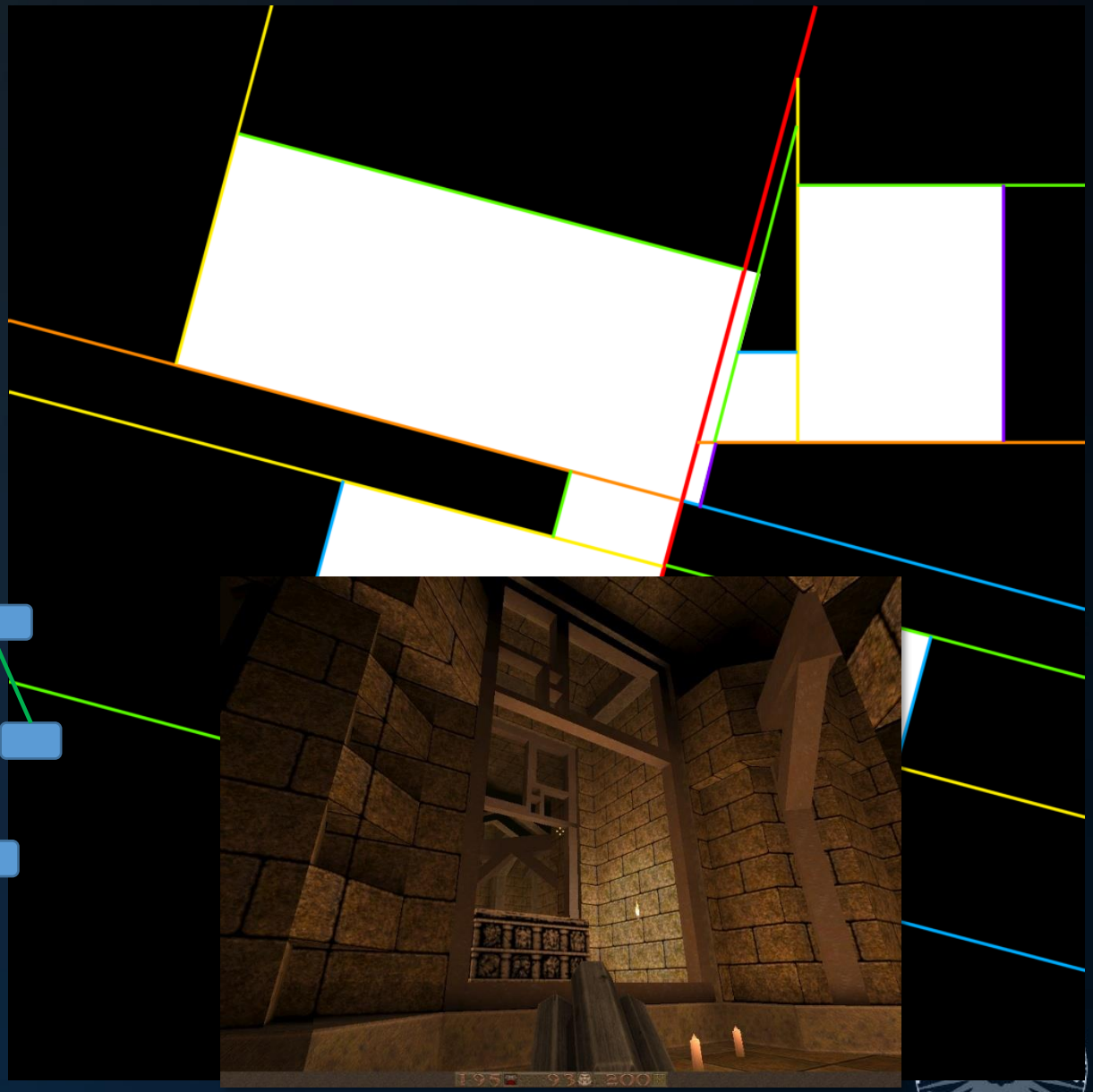


*: A. S. Glassner. Space Subdivision for Fast Ray Tracing. IEEE Computer Graphics and Applications, 4:15–22, 1984.

# Early Work

BSP Tree*



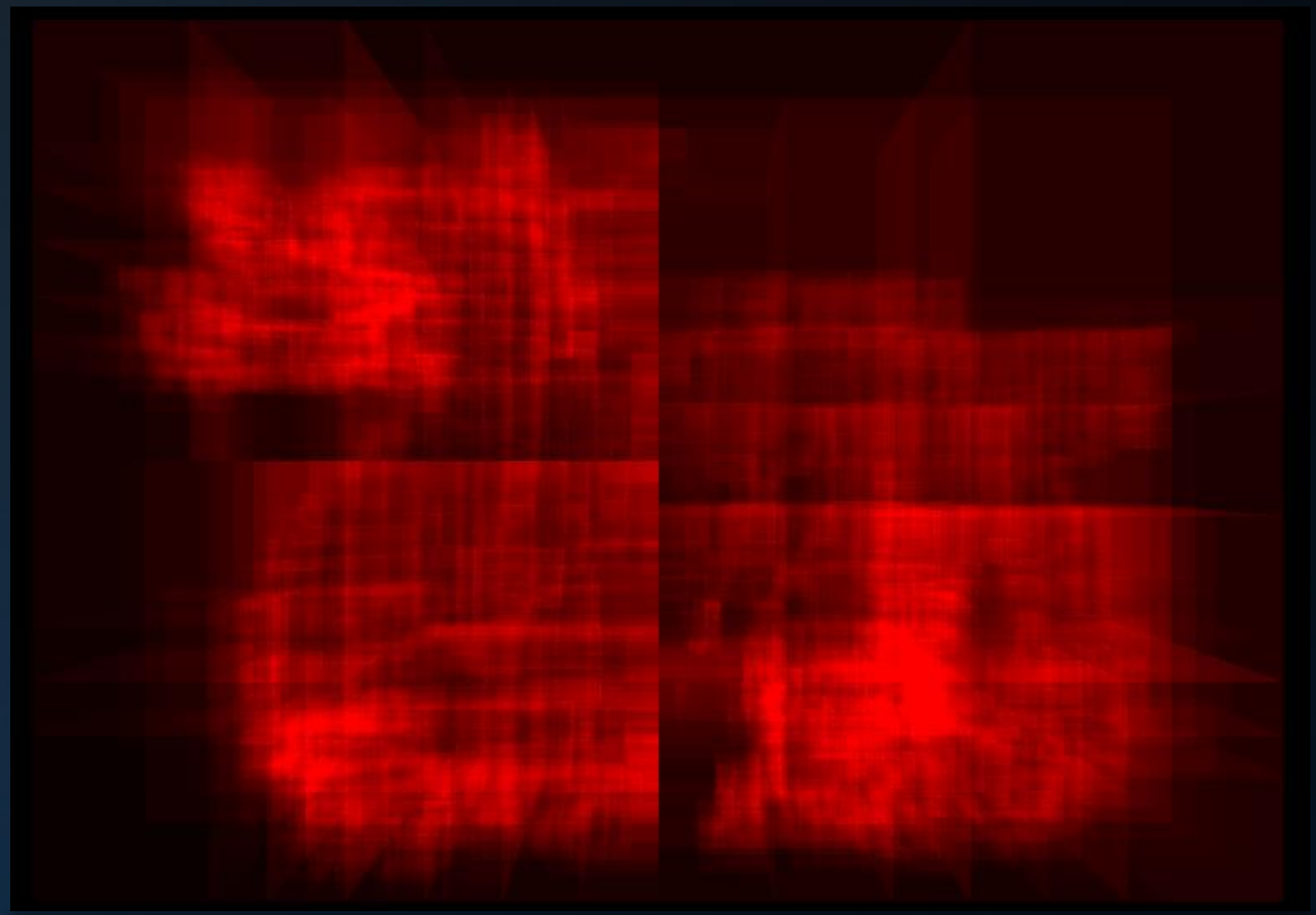



*: K. Sung, P. Shirley. Ray Tracing with the BSP Tree. In: Graphics Gems III, Pages 271-274. Academic Press, 1992

# Early Work

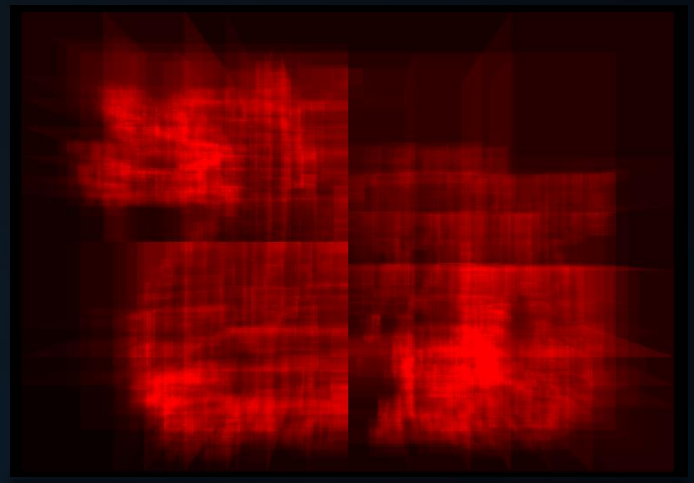
kD-Tree*

*"Axis-aligned BSP tree"*



*: V. Havran, Heuristic Ray Shooting Algorithms. PhD thesis, 2000.

# Early Work



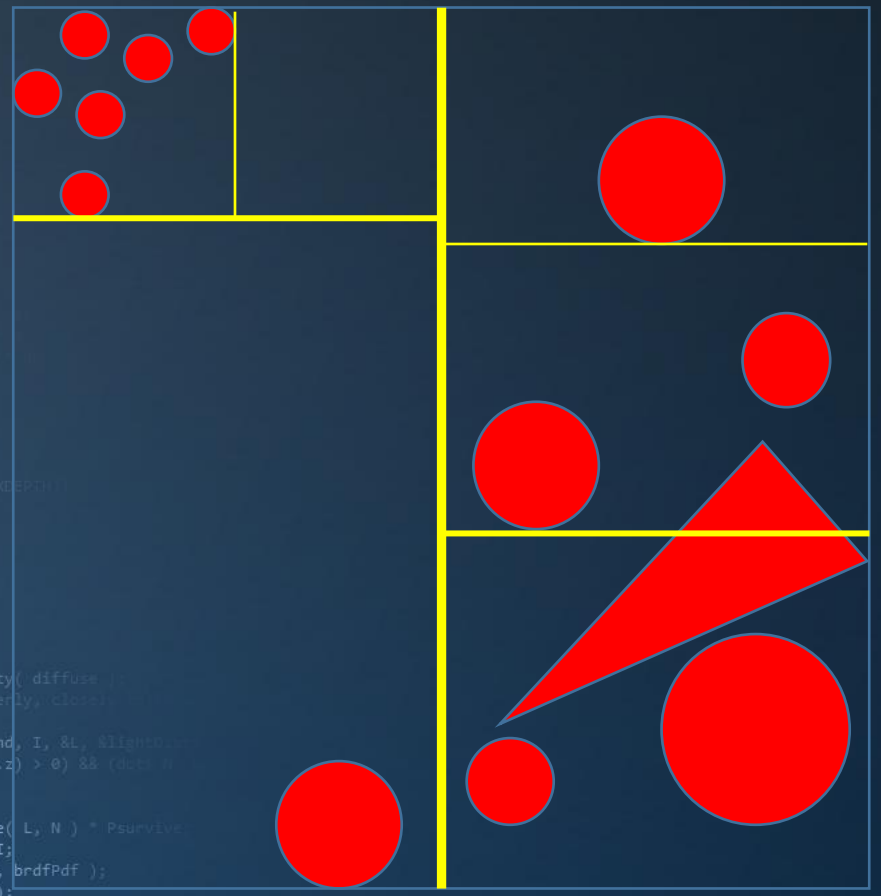
## kD-Tree Construction*

A kd-tree is a binary tree that recursively subdivides the space occupied by the scene.

- The root corresponds to the axis aligned bounding box (AABB) of the scene;
- Interior nodes represent planes that recursively subdivide space perpendicular to the coordinate axis;
- Leaf nodes store references to all the triangles overlapping the corresponding voxel.

*: On building fast kD-trees for ray tracing, and on doing that in O(N log N), Wald & Havran, 2006

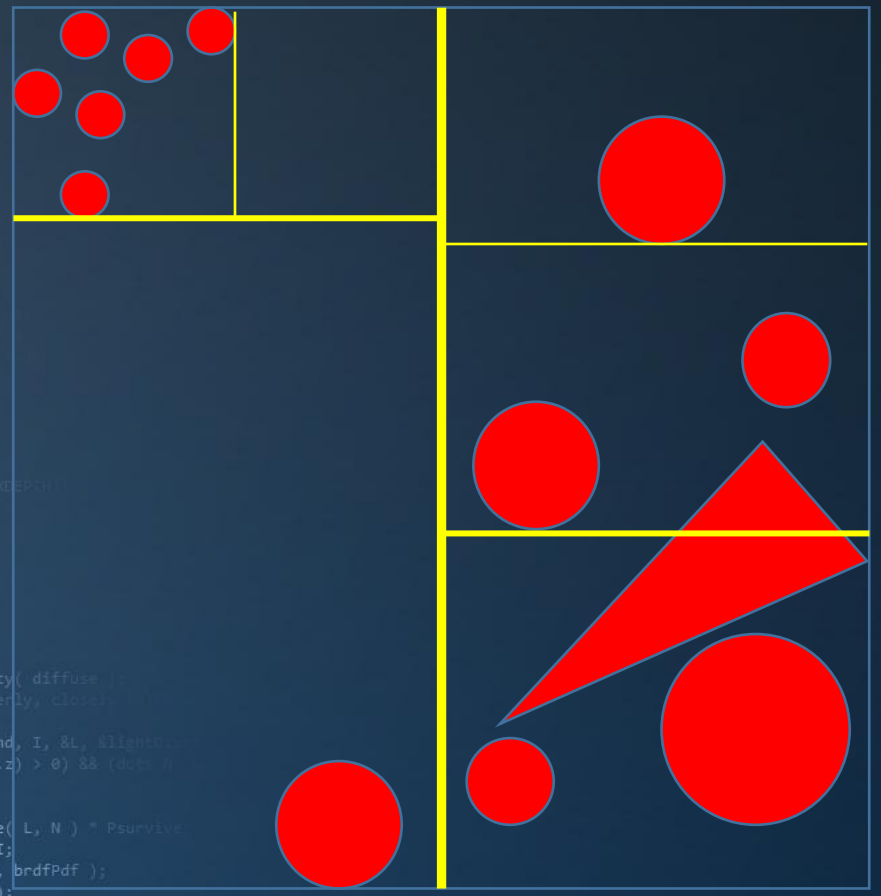
# Early Work



```
function Build( triangles T, voxel V )
{
   if (Terminate( T, V )) return new LeafNode( T )
   Plane p = FindPlane( T, V )
   Voxel V_L,V_R = Split V with p
   triangles T_L = { t ∈ T | (t ∩ V_L) ≠ 0
   triangles T_R = { t ∈ T | (t ∩ V_R) ≠ 0
   return new InteriorNode(
      p,
      Build( T_L, V_L ),
      Build( T_R, V_R )
   )
}

Function BuildKDTree( triangles T )
{
   Voxel V = bounds( T )
   return Build( T, V )
}
```

# Early Work



## Considerations

- Termination

*minimum primitive count, maximum recursion depth*

- Storage

*primitives may end up in multiple voxels: required storage hard to predict*

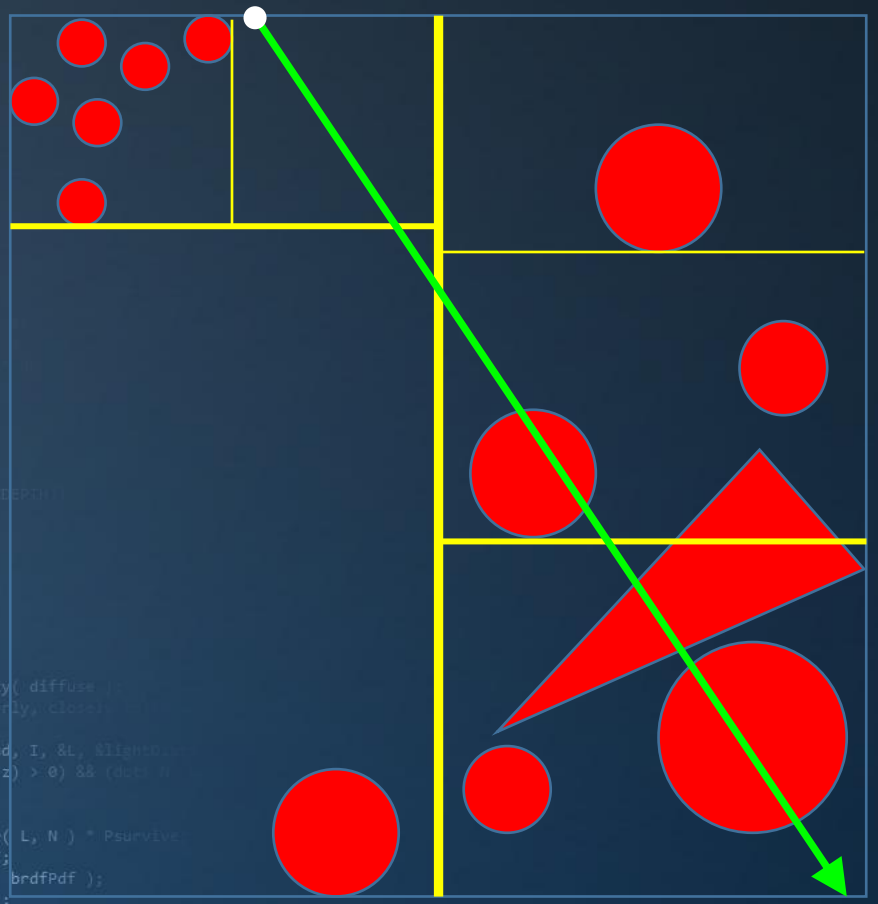
- Empty space

*empty space reduces probability of having to intersect primitives*

- Optimal split plane position / axis

*good solutions exist – will be discussed later.*

# Early Work

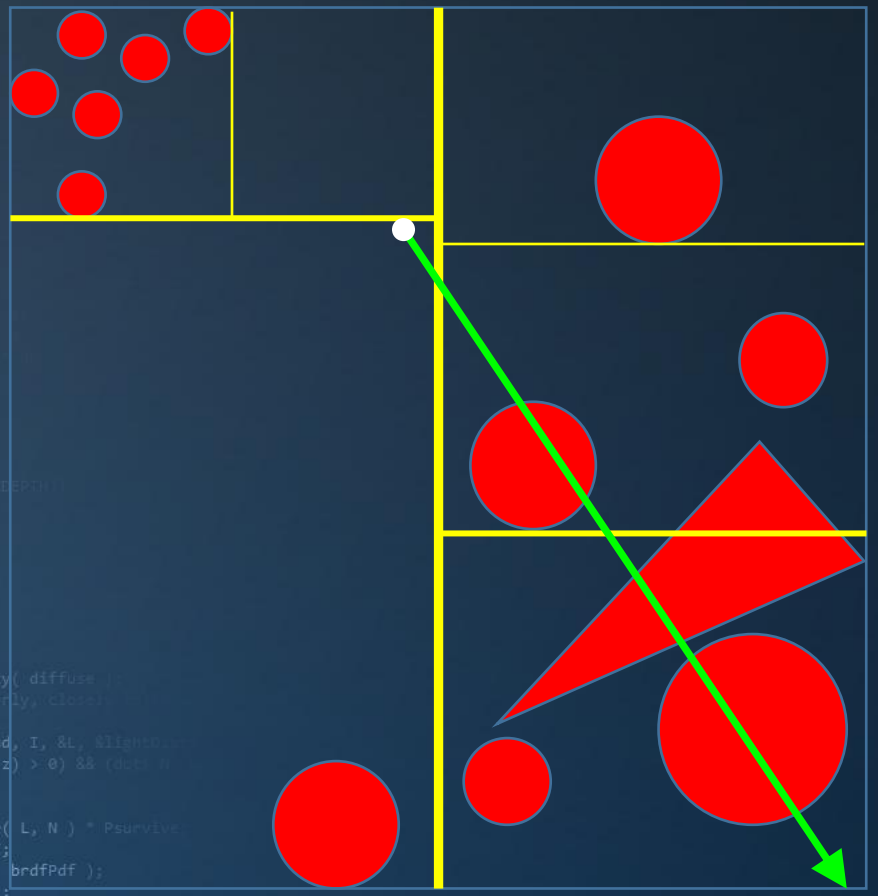


Traversal*

1. Find the point *P* where the ray enters the voxel
2. Determine which leaf node contains this point
3. Intersect the ray with the primitives in the leaf
   If intersections are found:
   - Determine the closest intersection
   - If the intersection is inside the voxel: done

*: Space-Tracing: a Constant Time Ray-Tracer, Kaplan, 1994

# Early Work



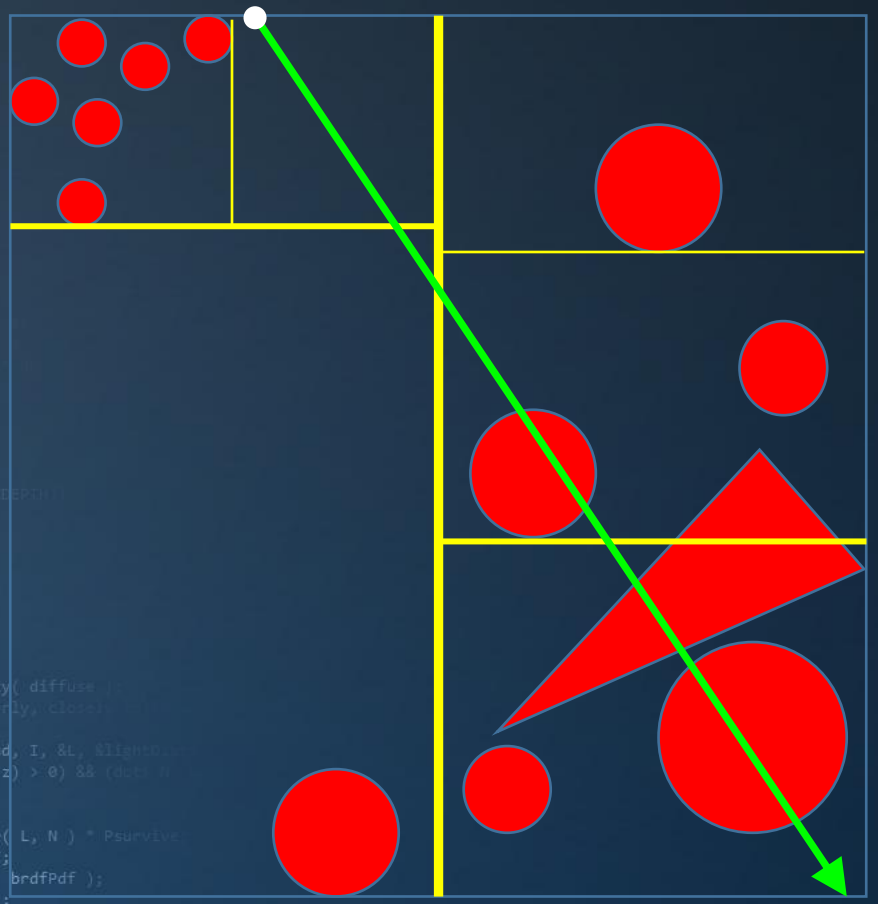
Traversal*

1. Find the point *P* where the ray enters the voxel
2. Determine which leaf node contains this point
3. Intersect the ray with the primitives in the leaf
   If intersections are found:
   - Determine the closest intersection
   - If the intersection is inside the voxel: done
4. Determine the point B where the ray leaves the voxel
5. Advance P slightly beyond B
6. Goto 1.

Note: step 2 traverses the tree repeatedly – inefficient.

*: Space-Tracing: a Constant Time Ray-Tracer, Kaplan, 1994

# Early Work



Traversal – Alternative Method*

For interior nodes:

1. Determine ‘near’ and ‘far’ child node
2. Determine if ray intersects ‘near’ and/or ‘far’

   If only one child node intersects the ray:
   - Traverse the node (goto 1)

   Else (both child nodes intersect the ray):
   - Push ‘far’ node to stack
   - Traverse ‘near’ node (goto 1)

For leaf nodes:

1. Determine the nearest intersection
2. Return if intersection is inside the voxel.

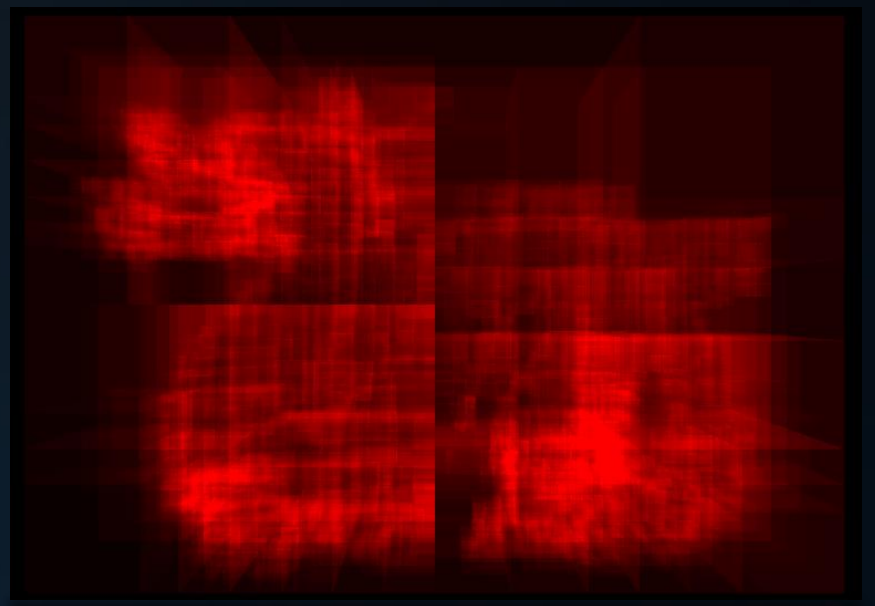
*: Data Structures for Ray Tracing, Jansen, 1986.

# Early Work

## kD-Tree Traversal

Traversing a kD-tree is done in a strict order.

Ordered traversal means we can stop as soon as we find a valid intersection.



# Early Work

## Acceleration Structures

| | Partitioning | Construction | Quality |
|---|---|---|---|
| ▪ Grid | space | O(n) | low |
| ▪ Octree | space | O(n log n) | medium |
| ▪ BSP | space | $O(n^2)$ | good |
| ▪ kD-tree | space | O(n log n) | good |
| ▪ BVH | object | O(n log n) | good |
| ▪ Tetrahedralization | space | ? | low |
| ▪ BIH | object | O(n log n) | medium |
| ▪ … | | | |

# Today’s Agenda:

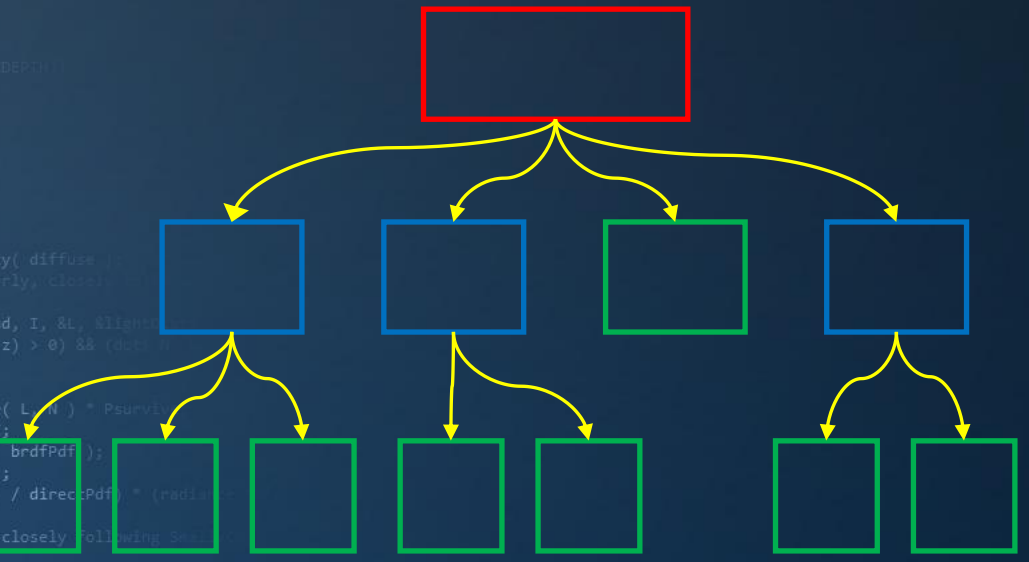
- Problem Analysis
- Early Work
- BVH Up Close

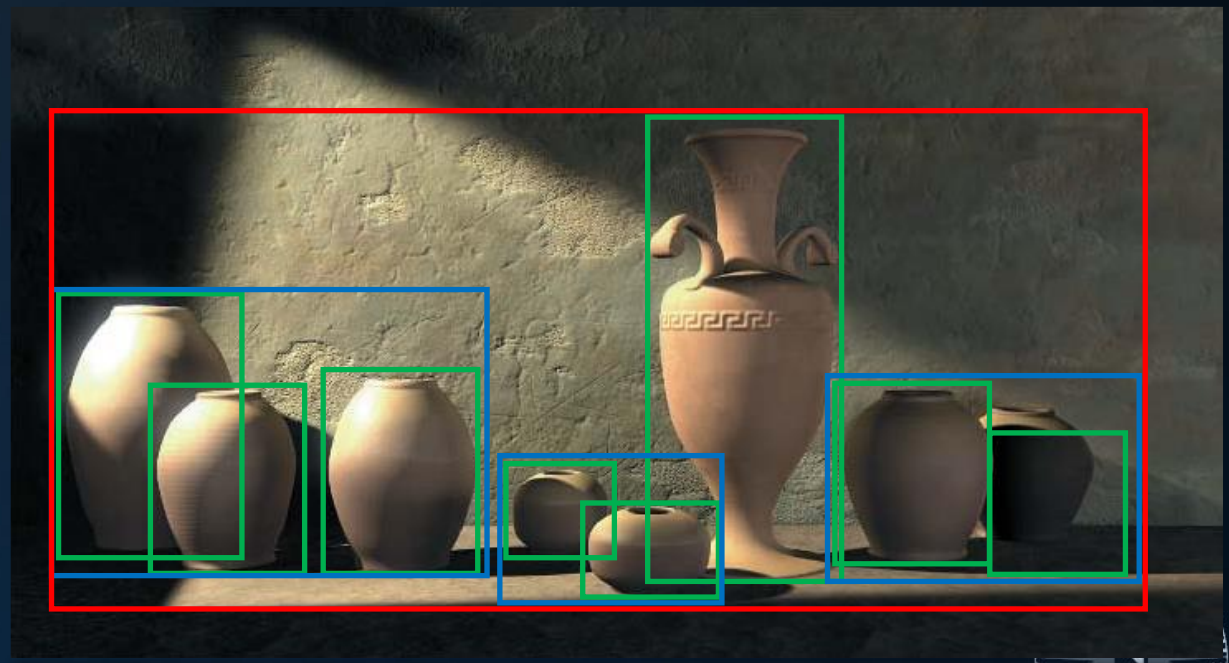
# BVH

Automatic Construction of Bounding Volume Hierarchies

BVH: tree structure, with:

- a bounding box per node
- pointers to child nodes
- geometry at the leaf nodes



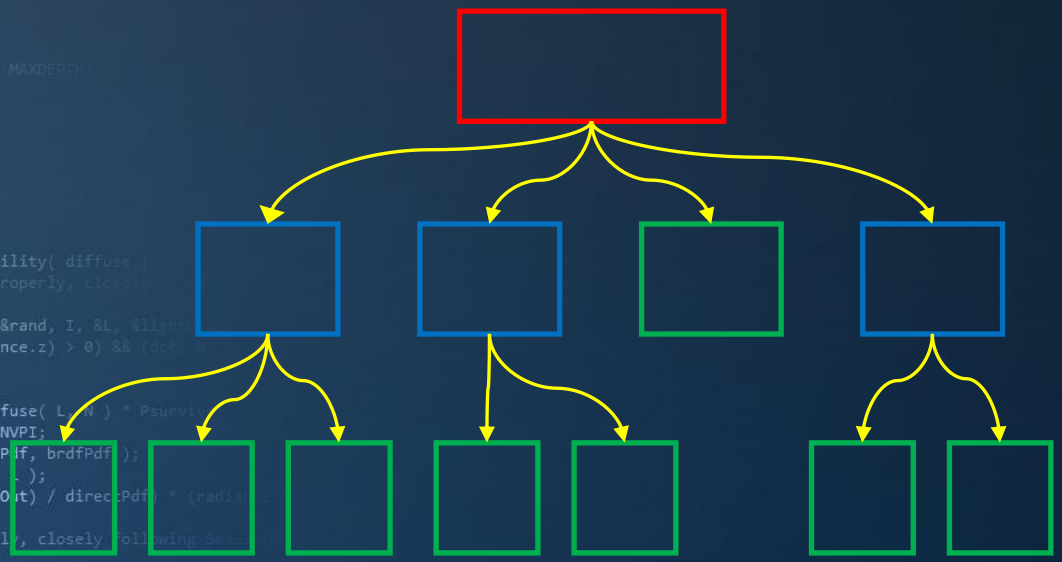


# BVH

Automatic Construction of Bounding Volume Hierarchies

BVH: tree structure, with:

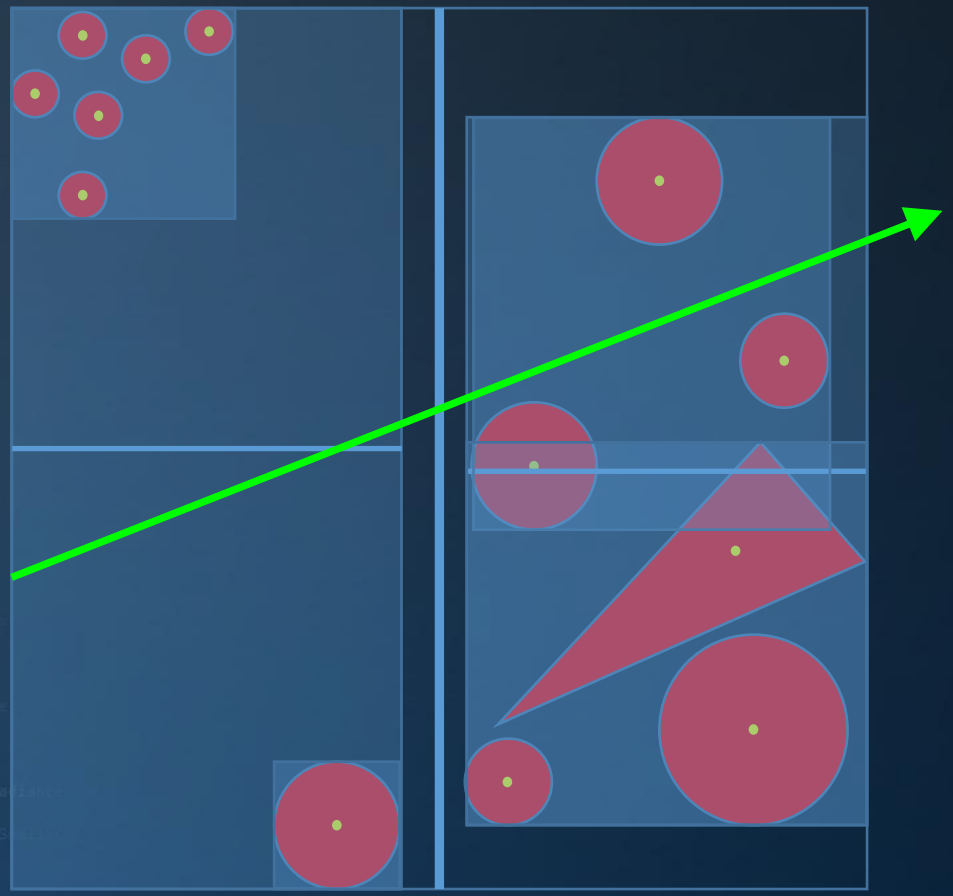
- a bounding box per node
- pointers to child nodes
- geometry at the leaf nodes

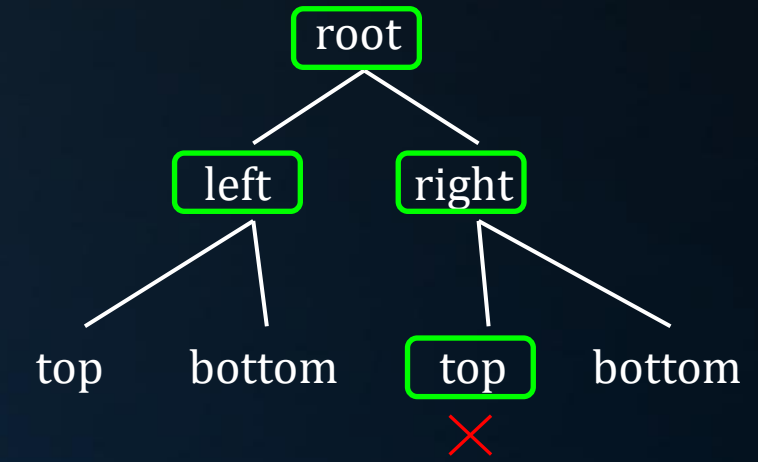


```
struct BVHNode
{
    AABB bounds;
    bool isLeaf;
    BVHNode*[] child;
    Primitive*[] primitive;
};
```

# BVH

## Automatic Construction of Bounding Volume Hierarchies





# BVH

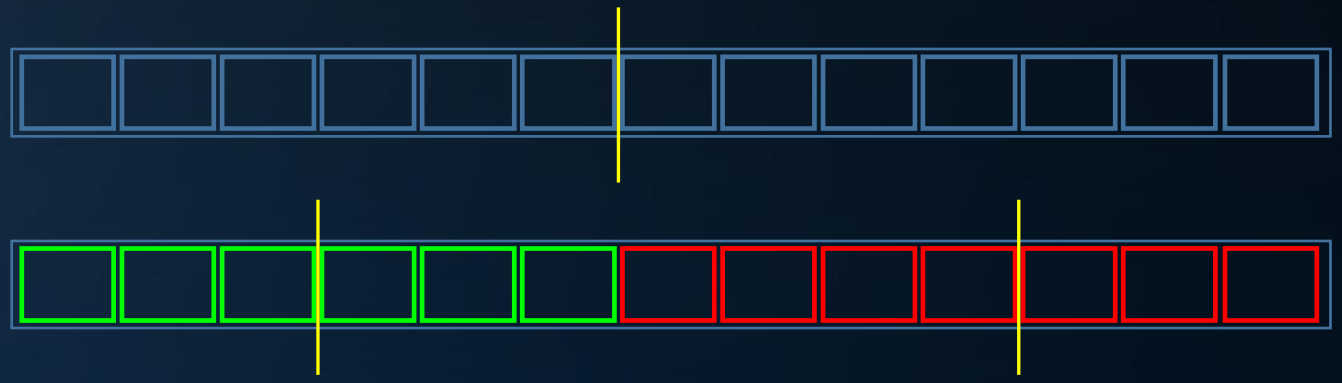
Automatic Construction of Bounding Volume Hierarchies

1. Determine AABB for primitives in array
2. Determine split axis and position
3. Partition
4. Repeat steps 1-3 for each partition

Note:

Step 3 can be done ‘in place’.

This process is identical to QuickSort: the split plane is The ‘pivot’.



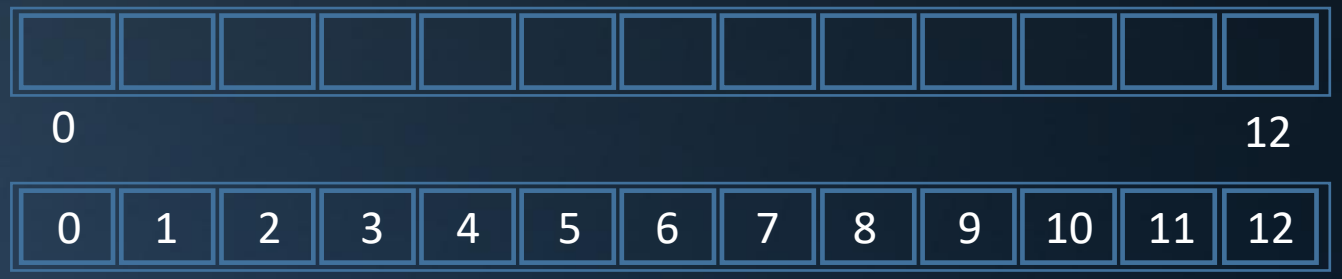
# BVH

## Automatic Construction of Bounding Volume Hierarchies


```
struct BVHNode
{
   AABB bounds;                 // 24 bytes
   bool isLeaf;                 // 4 bytes
   BVHNode* left, *right;       // 8 or 16 bytes
   Primitive** primList;        // ? bytes
};
```

# BVH

## Automatic Construction of Bounding Volume Hierarchies




```
struct BVHNode
{
    AABB bounds;                // 24 bytes
    bool isLeaf;                // 4 bytes
    BVHNode* left, *right;      // 8 or 16 bytes
    int first, count;           // 8 bytes
};
```

# BVH

## Automatic Construction of Bounding Volume Hierarchies

```
void BVH::ConstructBVH( Primitive* primitives )
{
    // create index array
    indices = new uint[N];
    for( int i = 0; i < N; i++ ) indices[i] = i;

    // allocate BVH root node
    root = new BVHNode();

    // subdivide root node
    root->first = 0;
    root->count = N;
    root->bounds = CalculateBounds( primitives, root->first, root->count );
    root->Subdivide();
}
```

```
void BVHNode::Subdivide()
{
    if (count < 3) return;
    this.left = new BVHNode();
    this.right = new BVHNode();
    Partition();
    this.left->Subdivide();
    this.right->Subdivide();
    this.isLeaf = false;
}
```

# BVH

## Automatic Construction of Bounding Volume Hierarchies

```
void BVH::ConstructBVH( Primitive* primitives )
{
   // create index array
   indices = new uint[N];
   for( int i = 0; i < N; i++ ) indices[i] = i;

   // allocate BVH root node
   pool = new BVHNode[N * 2 – 1];
   root = &pool[0];
   poolPtr = 2;

   // subdivide root node
   root->first = 0;
   root->count = N;
   root->bounds = CalculateBounds( primitives, root->first, root->count );
   root->Subdivide();
}
```

```
void BVHNode::Subdivide()
{
   if (count < 3) return;
   this.left = &pool[poolPtr++];
   this.right = &pool[poolPtr++];
   Partition();
   this.left->Subdivide();
   this.right->Subdivide();
   this.isLeaf = false;
}
```

# BVH

## Automatic Construction of Bounding Volume Hierarchies


primitives

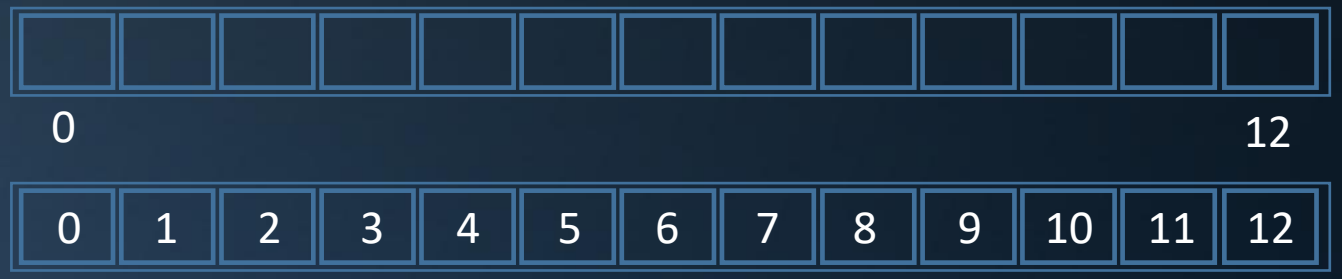

primitive indices

```
struct BVHNode
{
    AABB bounds;            // 24 bytes
    bool isLeaf;            // 4 bytes
    int left, right;        // 8 bytes
    int first, count;       // 8 bytes, total 44 bytes
};
```

BVH nodes

# BVH

## Automatic Construction of Bounding Volume Hierarchies




```
struct BVHNode
{
    AABB bounds;              // 24 bytes
    int left;                 // 4 bytes
    int first, count;         // 8 bytes, total 36
};
```


# BVH

## Automatic Construction of Bounding Volume Hierarchies


primitives


primitive indices

```
struct BVHNode
{
   AABB bounds;                 // 24 bytes
   int leftFirst;               // 4 bytes
   int count;                   // 4 bytes, total 32  ☺
};
```


BVH nodes

# BVH

Automatic Construction of Bounding Volume Hierarchies

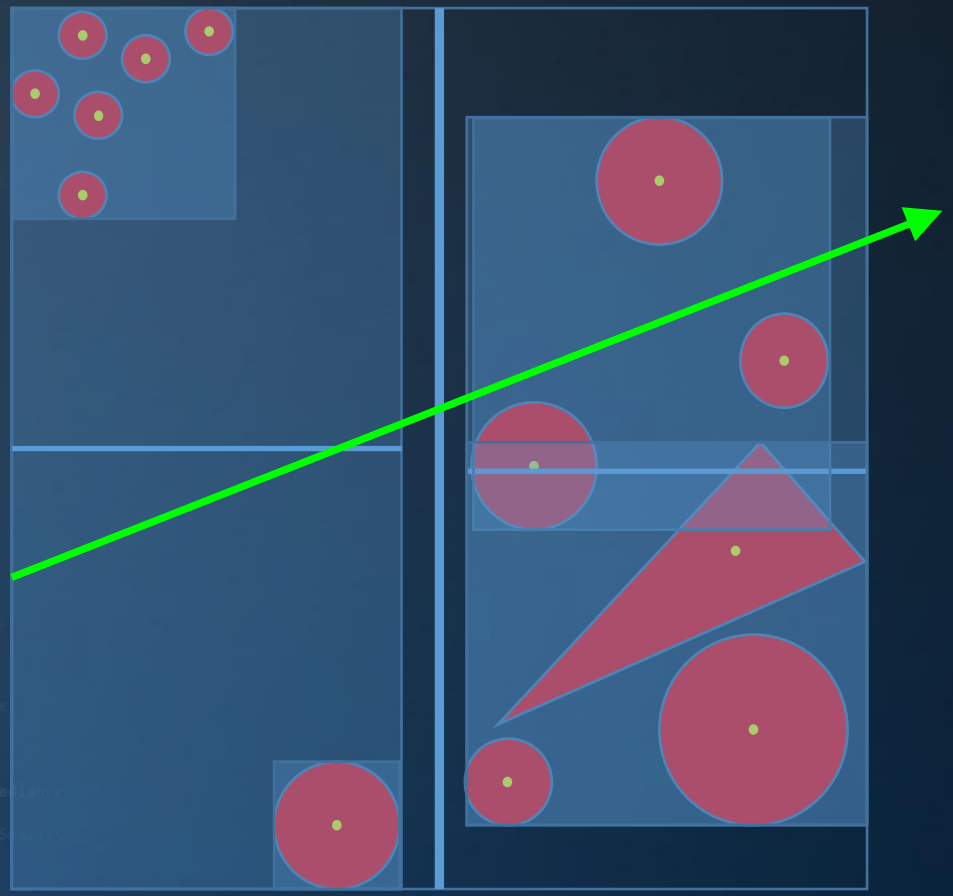
Optimal BVH representation:

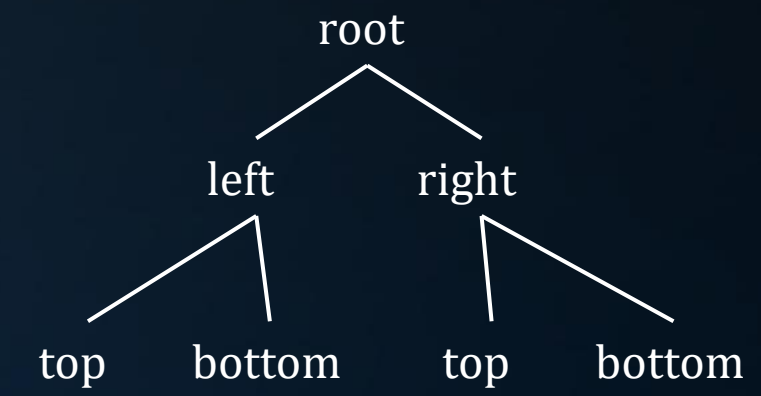
- Partitioning of array of indices pointing to original triangles
- Using indices of BVH nodes, and assuming right = left + 1
- BVH nodes use exactly 32 bytes (2 per cache line)
- BVH node pool allocated in cache aligned fashion
- AABB splitted in 2x 12 bytes; 1st followed by ‘leftFirst’, 2nd by ‘count’.

Note: the BVH is now ‘relocatable’ and thus ‘serializable’.

# BVH

## BVH Traversal





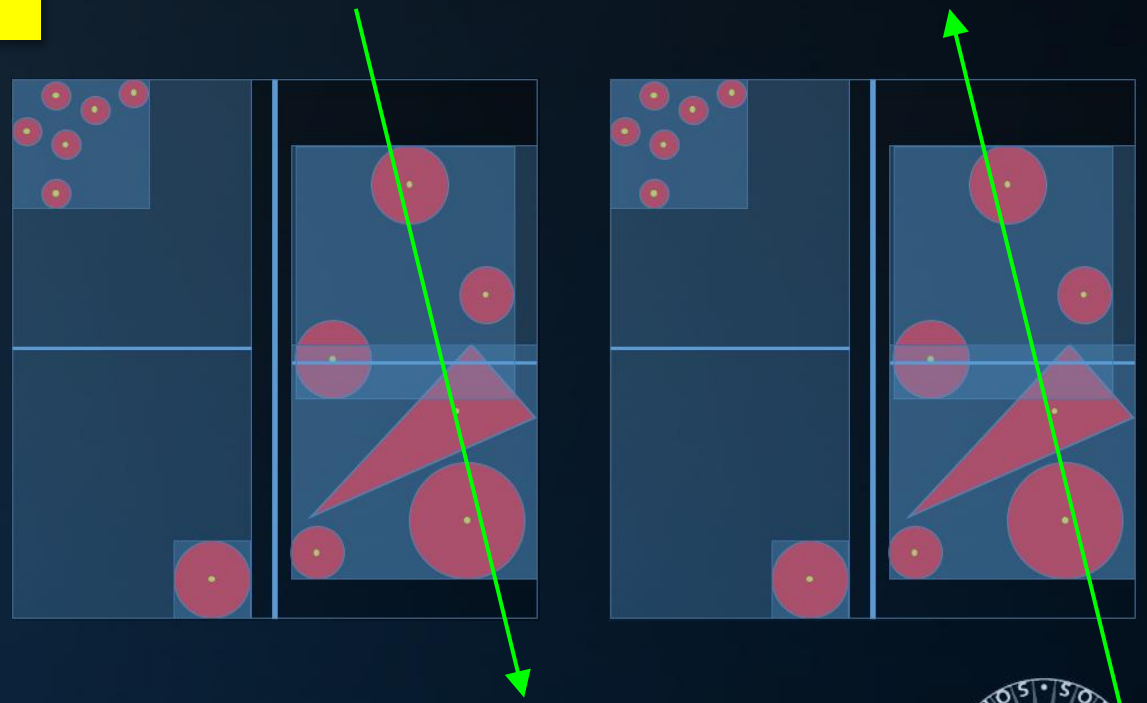
# BVH

## BVH Traversal

Basic process:

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

Ray:
vec3 O, D
float t



# BVH

## BVH Traversal

Ordered traversal, option 1:

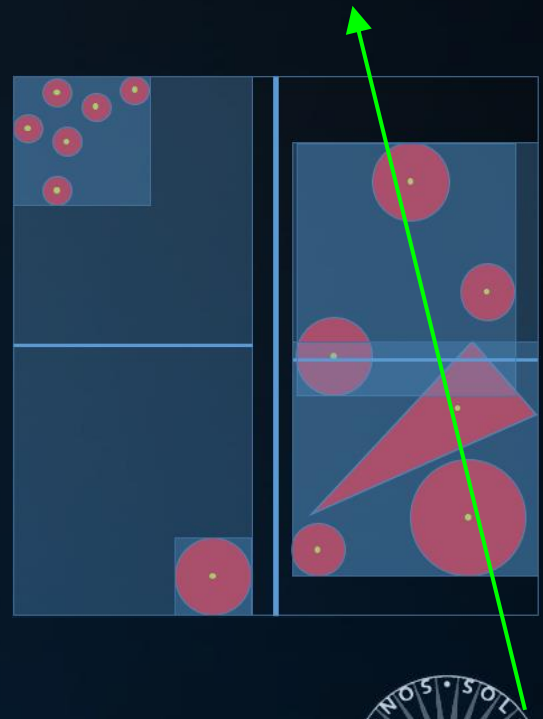
- Calculate distance to both child nodes
- Traverse the nearest child node first

Ordered traversal, option 2:

- For each BVH node, store the axis along which it was split
- Use ray direction sign for that axis to determine near and far

Ordered traversal, option 3:

- Determine the axis for which the child node centroids are furthest apart
- Use ray direction sign for that axis to determine near and far.



# BVH

## BVH Traversal

Ordered traversal of a BVH is approximative.

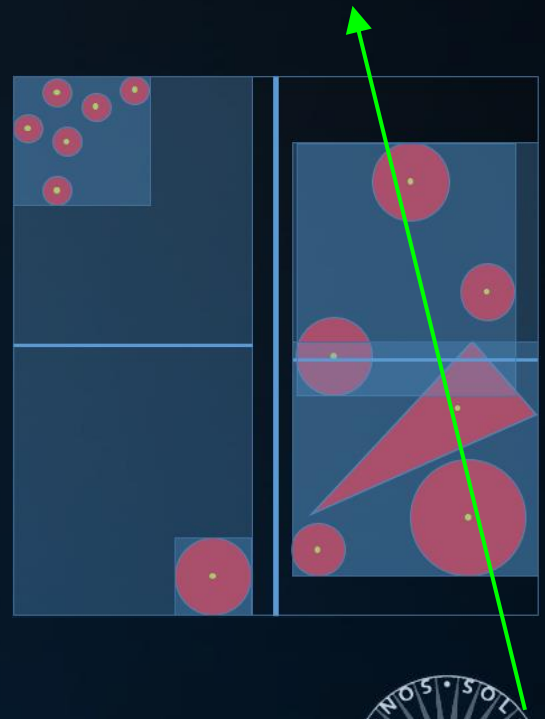
- Nodes may overlap.

And:

- We may find a closer intersection in a node that we visit later.

However:

- We do not have to visit nodes beyond an already found intersection distance.



# Today’s Agenda:

- Problem Analysis
- Early Work
- BVH Up Close

# INFOMAGR – Advanced Graphics

Jacco Bikker - November 2022 - February 2023

# END of “Acceleration Structures”

next lecture: “The Perfect BVH”