INFOGR – Computer Graphics

J. Bikker - April-July 2015 - Lecture 11: “Accelerate”

Welcome!

Today’s Agenda:

- High-speed Ray Tracing
- Acceleration Structures
- The Bounding Volume Hierarchy
- BVH Construction
- BVH Traversal
- Optimizing Construction
- High-speed Traversal

High-speed Ray Tracing

Ray Tracing – Needful things

Whitted-style ray tracing:

1 primary ray per pixel
1 shadow ray per pixel per light
Optional: rays for reflections & refraction

Estimate:

- 10 rays per pixel
- 1M pixels (~1280x800)
- 30 fps

➔ 300Mrays/s

How does one intersect 300Mrays/s on a 3Ghz CPU?
Easy: use no more than 10 cycles per ray.

High-speed Ray Tracing

Actually…

- We have 8 cores (so 80 cycles)
- Executing AVX code (so 640 cycles)
- Plus 20% gains from hyperthreading (768 cycles).

But really…

Assuming we get a linear increase in performance for the number of cores and AVX, how do we intersect thousands of triangles in 768 cycles?

High-speed Ray Tracing

Optimization

1. Measure: performance & scalability
2. High level optimizations: improve algorithmic complexity
3. Low level optimization: instruction level & thread-level parallelism, caching
4. GPGPU

More in the master course Optimization & Vectorization.

Today’s Agenda:

- High-speed Ray Tracing
- Acceleration Structures
- The Bounding Volume Hierarchy
- BVH Construction
- BVH Traversal
- Optimizing Construction
- High-speed Traversal

High-speed Ray Tracing

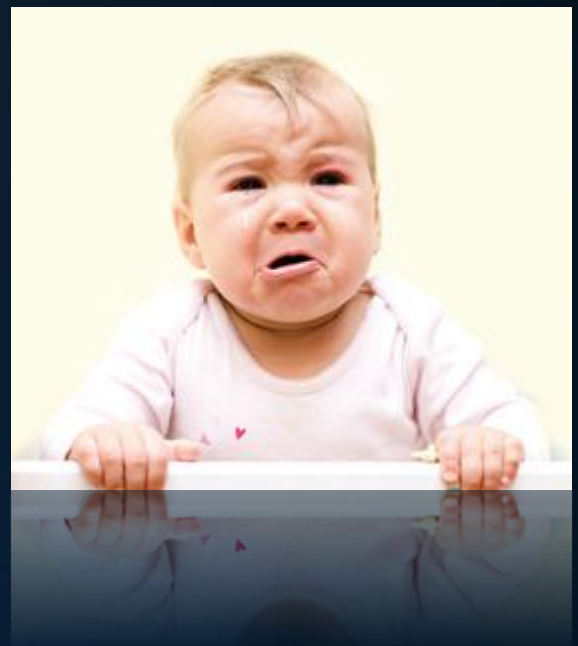
Optimization: reduce algorithmic complexity

Complexity:

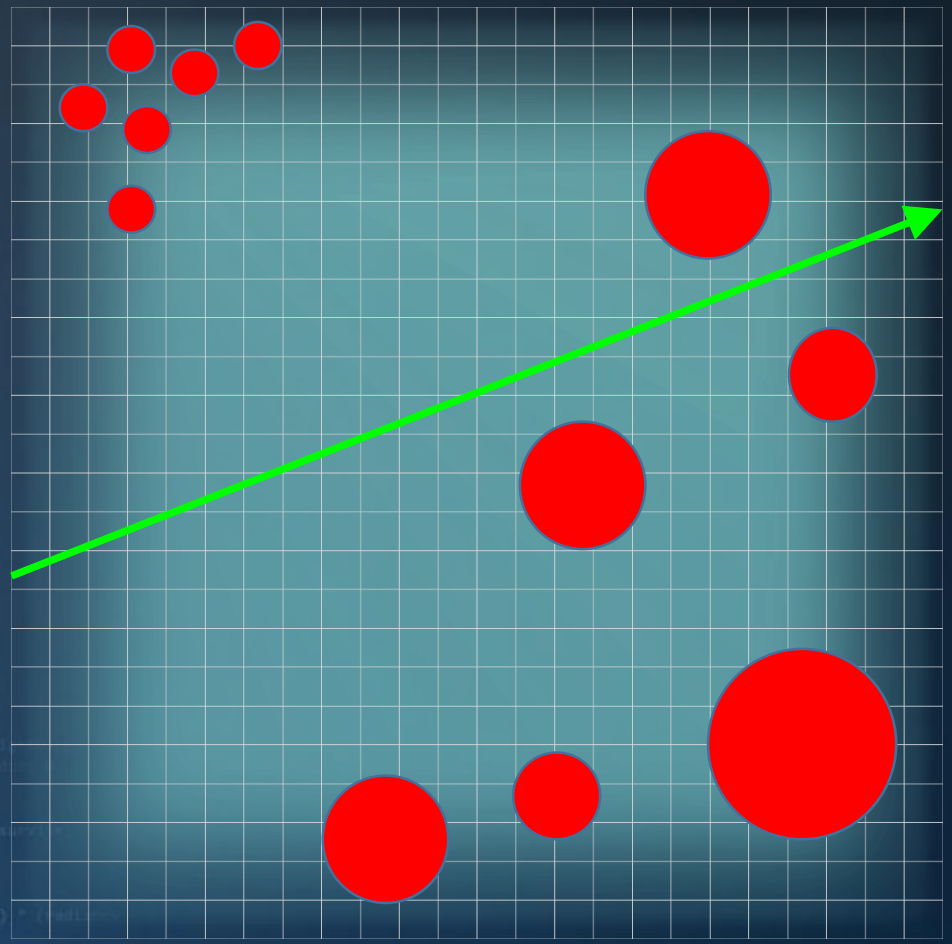
number of ray/primitive intersections

= pixels * paths per pixel * average path length * primitives

= 1M * 1 * 2 * 1M



Acceleration Structures



Option 1:

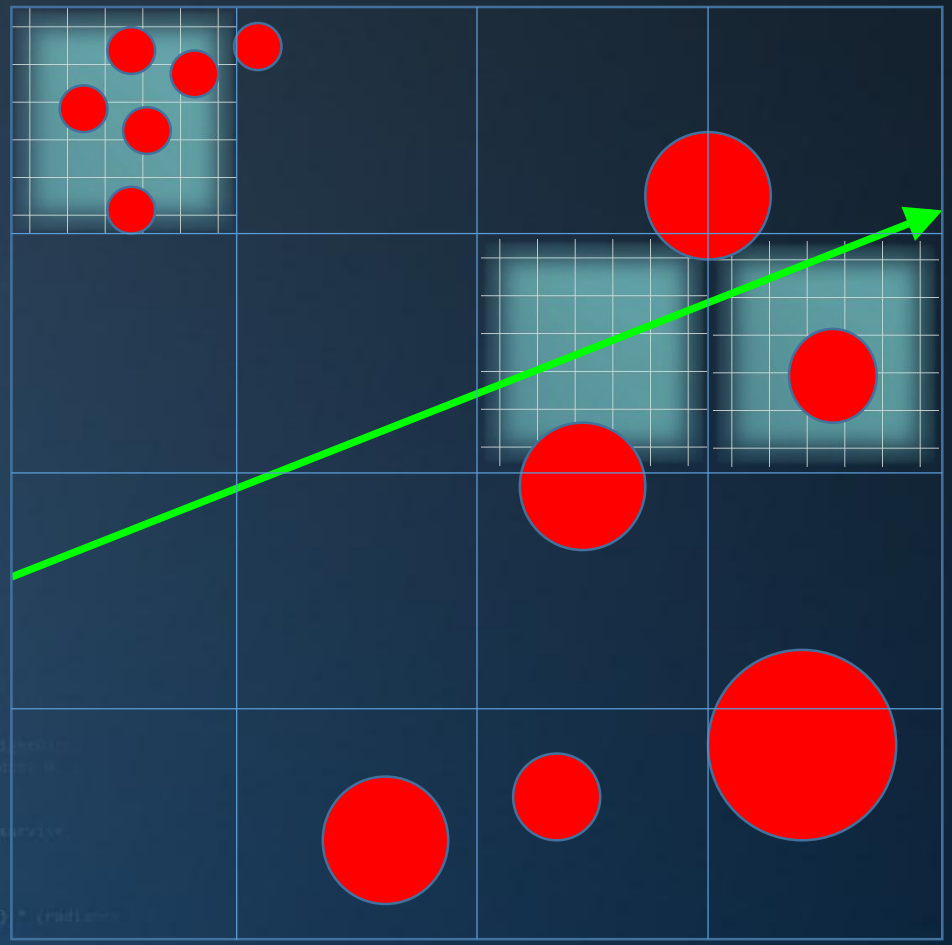
Use a grid.

- Each grid cell has a list of primitives that overlap it.
- The ray traverses the grid, and intersects only primitives in the grid cells it visits.

Problems:

- Many primitives will be checked more than once.
- It costs to traverse the grid.
- How do we chose grid resolution?
- What if scene detail is not uniform?

Acceleration Structures



Option 2:

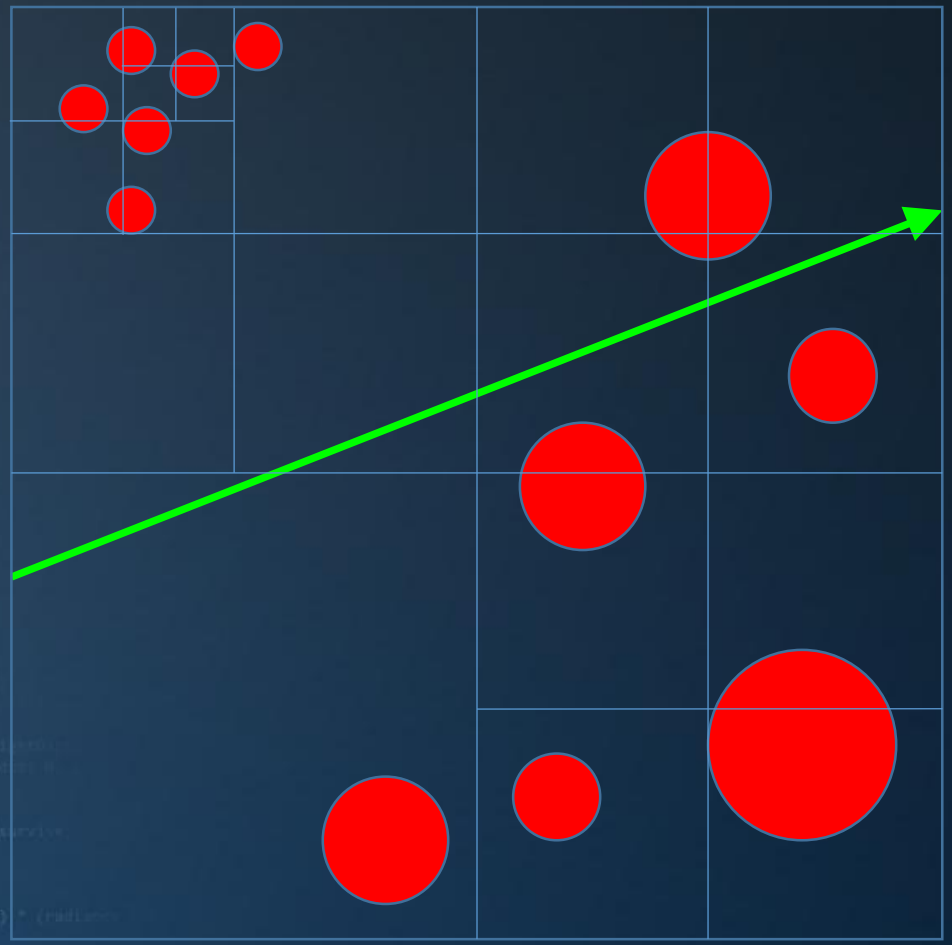
Use a nested grid.

- We use fewer cells. Each grid cell that overlaps multiple primitives has a smaller grid in it.
- The ray rapidly traverses empty space, and checks the nested grids when needed.

Problems:

- How do we chose grid resolutions?
- Is this the optimal way to traverse space?

Acceleration Structures



Option 3:

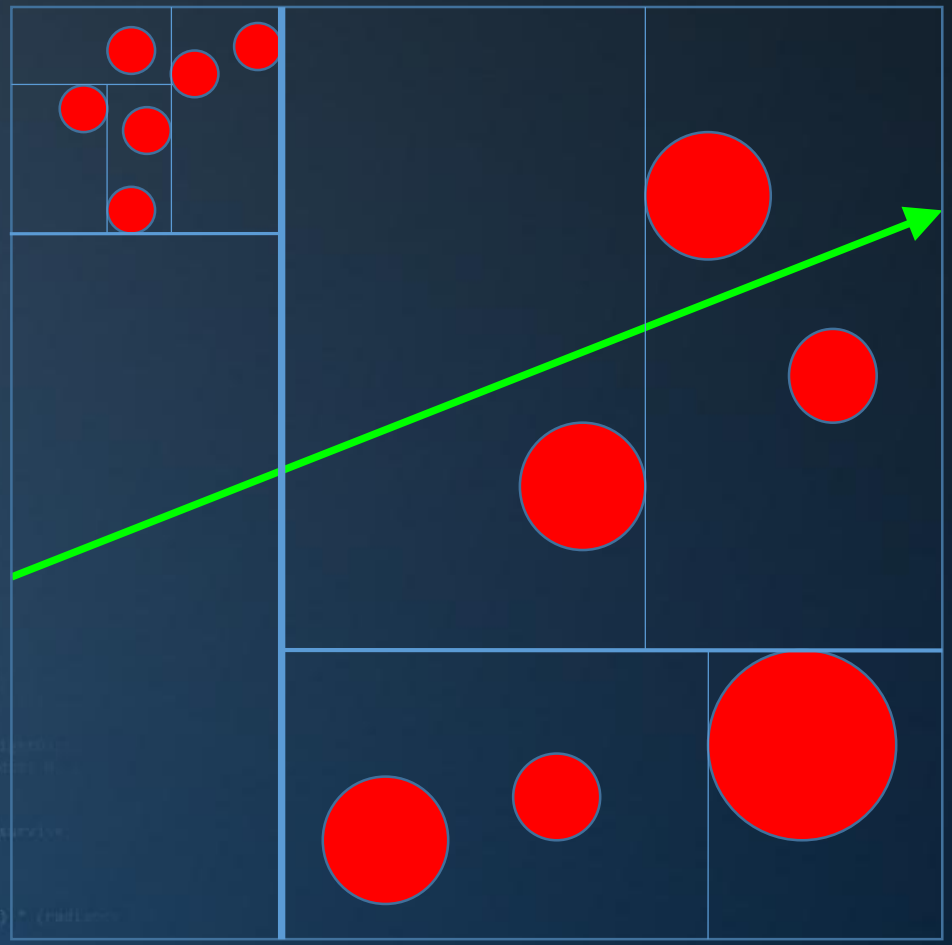
Use an octree.

- We start with a bounding box of the scene;
- The box is recursively subdivided in 8 equal boxes as long as it contains more than X primitives.

Problems:

- What if all the detail is exactly in the centre of the scene?
- Splitting in 8 boxes: is that the optimal subdivision?

Acceleration Structures



Option 4:

Use an kD-tree.

- We start with a bounding box of the scene;
- Using arbitrary axis-aligned planes, we recursively cut it in two halves as long as it contains more than X primitives.

Problems:

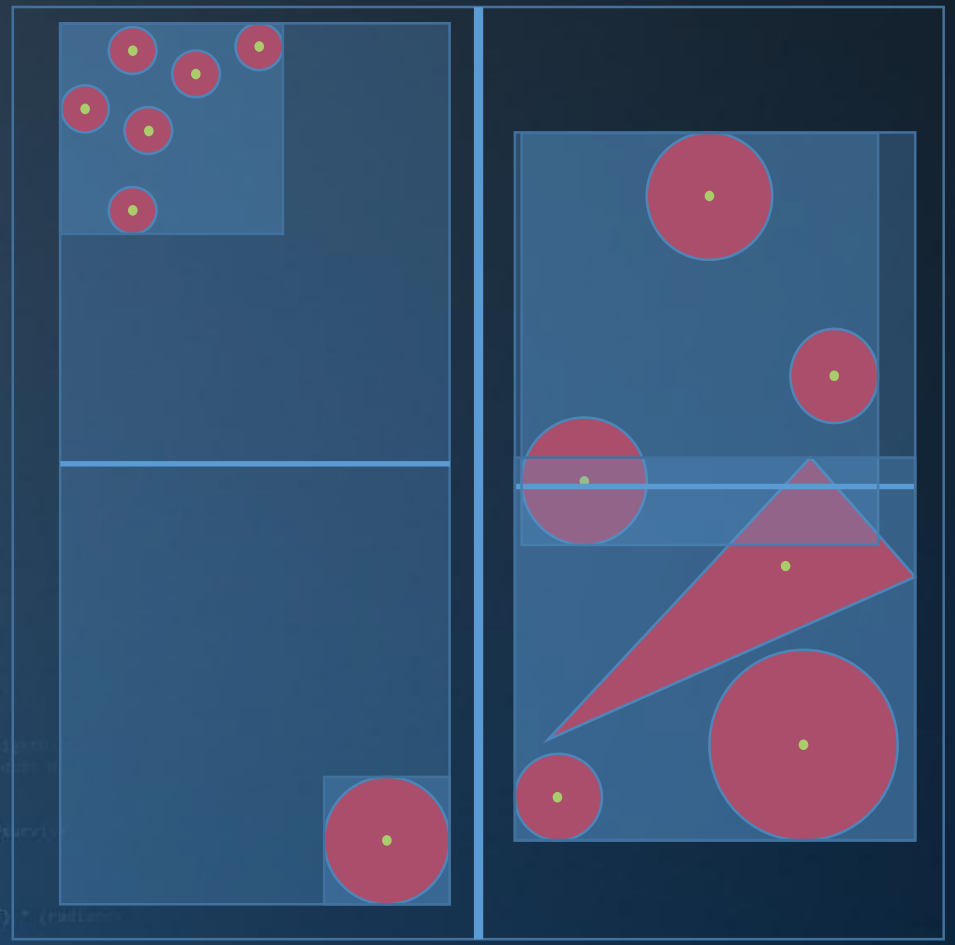
- Primitives may end up in multiple leaf nodes.
- How hard is it to build such a tree?


FACES

Today’s Agenda:

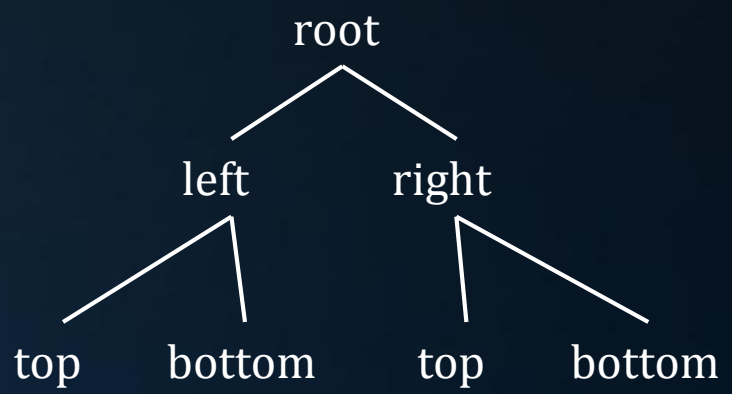
- High-speed Ray Tracing
- Acceleration Structures
- The Bounding Volume Hierarchy
- BVH Construction
- BVH Traversal
- Optimizing Construction
- High-speed Traversal

BVH



Option 5:

Use a bounding volume hierarchy.



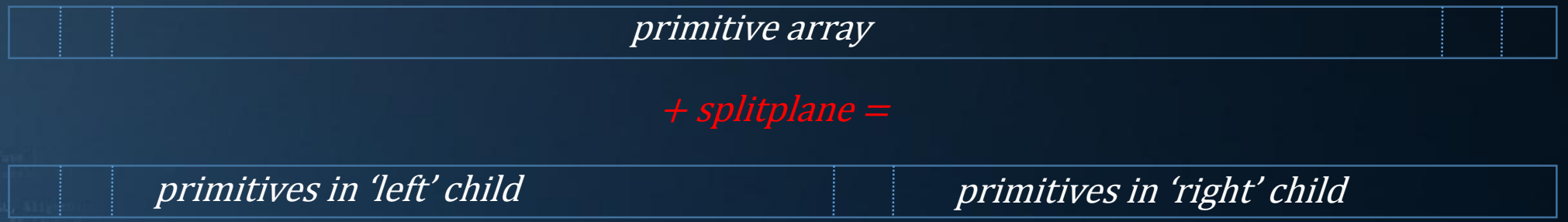
BVH

The Bounding Volume Hierarchy

BSPs, grids, octrees and kD-trees are examples of *spatial subdivisions*.

The BVH is of a different category: it is an *object partitioning scheme*:

Rather than recursively splitting space, it splits collections of objects.

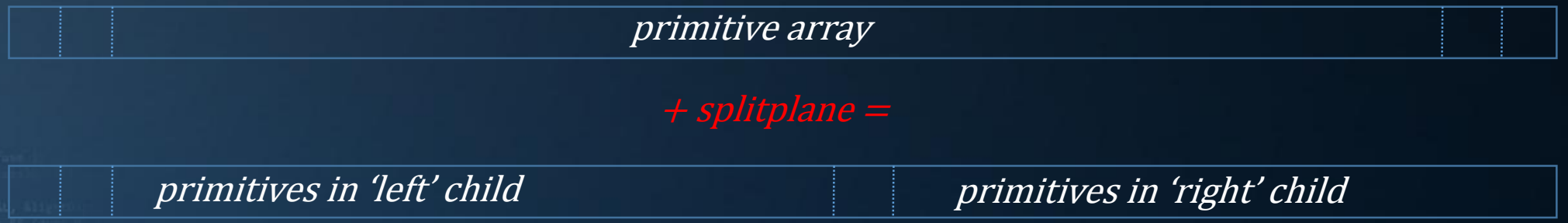


BVH

The Bounding Volume Hierarchy

Sorting an array of elements based on a value:
BVH is very similar to *QuickSort*.

In the BVH construction algorithm, the split plane position is the pivot.



BVH

Bounding Volume Hierarchy: data structure

```
struct BVHNode
{
   BVHNode* left;                    // 4 or 8 bytes
   BVHNode* right;                   // 4 or 8 bytes
   aabb bounds;                      // 2 * 3 * 4 = 24 bytes
   bool isLeaf;                      // ?
   vector<Primitive*> primitives;    // ?
};
```

BVH

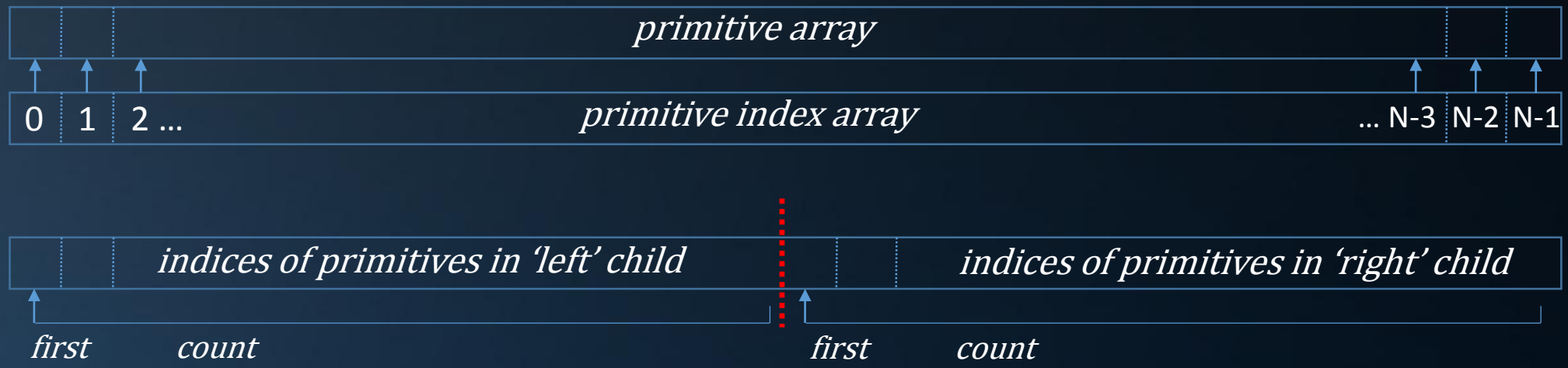
Bounding Volume Hierarchy: construction

```
void ConstructBVH( Primitive* primitives )
{
   BVHNode* root = new BVHNode();
   root->primitives = primitives;
   root->bounds = CalculateBounds( primitives );
   root->isLeaf = true;
   root->Subdivide();
}

void BVHNode::Subdivide()
{
   if (primitives.size() < 3) return;
   this.left = new BVHNode(), this.right = new BVHNode();
   …split ‘bounds’ in two halves, assign primitives to each half…
   this.left->Subdivide();
   this.right->Subdivide();
   this.isLeaf = false;
}
```

BVH

Bounding Volume Hierarchy: construction



Construction consequences:

- Construction happens *in place:* primitive array is constant, index array is changed
- Very similar to Quicksort (split plane = pivot)

Data consequences:

- ‘Primitive list’ for node becomes *offset + count*
- No pointers!
- No pointers? (what about left / right?)

BVH

Bounding Volume Hierarchy: data structure

```
struct BVHNode
{
   BVHNode* left;
   BVHNode* right;
   aabb bounds;
   bool isLeaf;
   vector<Primitive*> primitives;
};
```

```
struct BVHNode
{
   uint left;              // 4 bytes
   uint right;             // 4 bytes
   aabb bounds;            // 24 bytes
   bool isLeaf;            // 4 bytes
   uint first;             // 4 bytes
   uint count;             // 4 bytes
};                         // --------
                           // 44 bytes
```

BVH

Bounding Volume Hierarchy: data structure

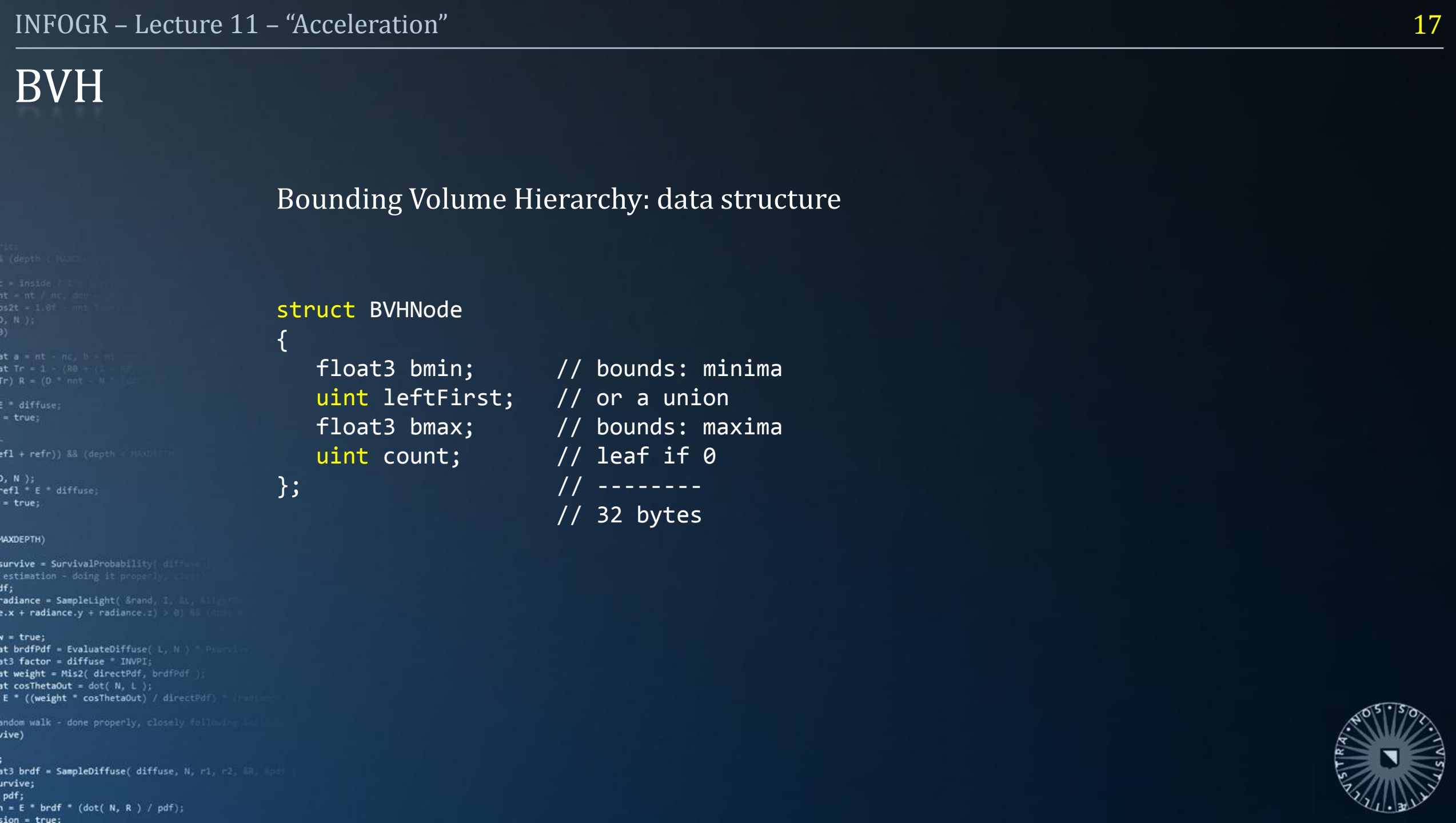
```
struct BVHNode
{
   union                // 4 bytes
   {
      uint left;
      uint first;
   };
   aabb bounds;         // 24 bytes
   uint count;          //  4 bytes
};                      // --------
                        // 32 bytes
```

```
struct BVHNode
{
   uint left;           // 4 bytes
   uint right;          // 4 bytes
   aabb bounds;         // 24 bytes
   bool isLeaf;         // 4 bytes
   uint first;          // 4 bytes
   uint count;          // 4 bytes
};                      // --------
                        // 44 bytes
```

BVH

Bounding Volume Hierarchy: data structure

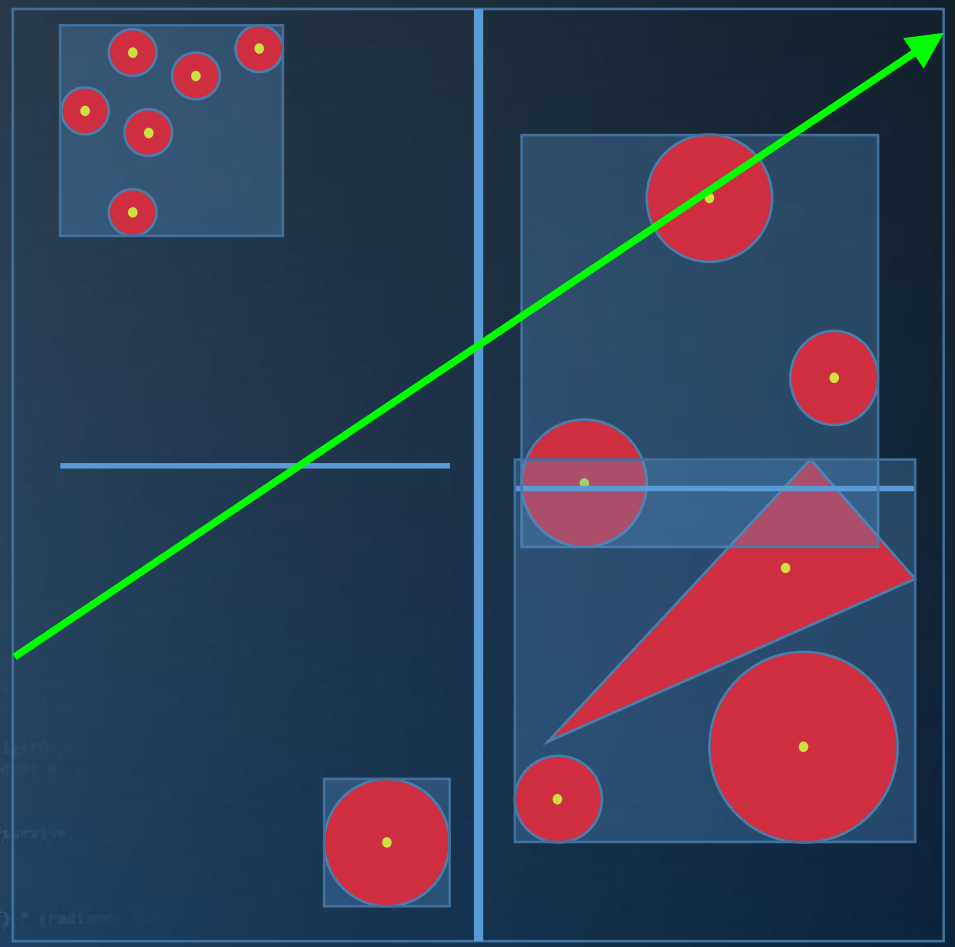
```
struct BVHNode
{
   float3 bmin;      // bounds: minima
   uint leftFirst;   // or a union
   float3 bmax;      // bounds: maxima
   uint count;       // leaf if 0
};                   // --------
                     // 32 bytes
```



Today’s Agenda:

- High-speed Ray Tracing
- Acceleration Structures
- The Bounding Volume Hierarchy
- BVH Construction
- BVH Traversal
- Optimizing Construction
- High-speed Traversal

BVH Traversal



BVH Traversal Algorithm:

Starting with the root:

If the node is a leaf node:
    Intersect triangles.
Else:
    If the ray intersects the left child AABB:
        Traverse left child
    If the ray intersects the right child AABB:
        Traverse right child

How efficient is this?

In this case: we check every AABB, but we only try to intersect one red sphere.
(total: 8 tests)

BVH Traversal

BVH Efficiency

The number of nodes in a BVH is at most $2N - 1$.
Example:

16	1
8 + 8	2
(4 + 4) + (4 + 4)	4
((2+2) + (2+2))+((2+2) + (2+2))	8
1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1	16

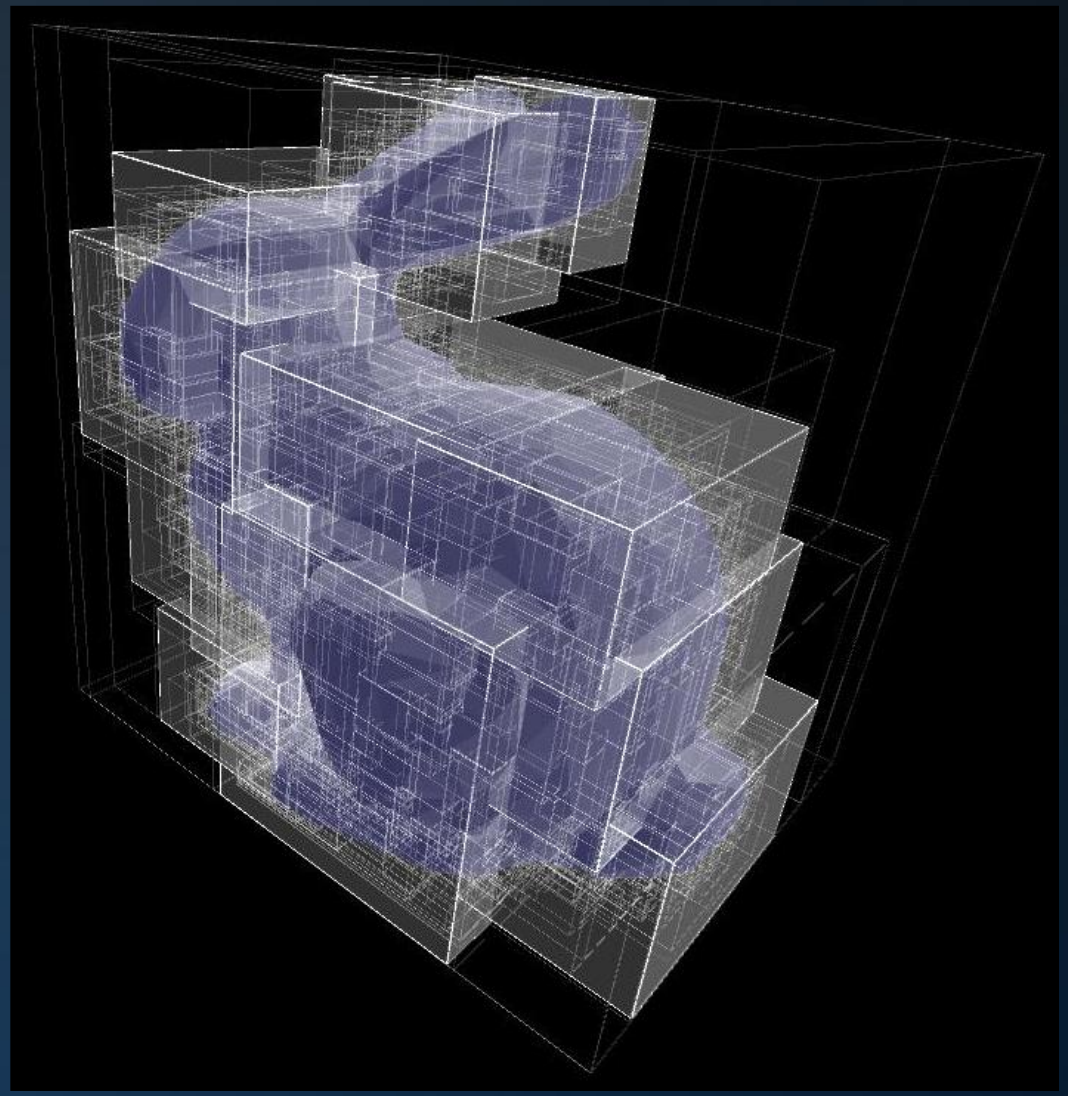
	31

- In this case, we get from the root to a leaf in 5 steps, or: $\log_2 N + 1$.
- For 1024 primitives, we get to a leaf in 11 steps.
- For 1M primitives, we get to a leaf in 21 steps.

Today’s Agenda:

- High-speed Ray Tracing
- Acceleration Structures
- The Bounding Volume Hierarchy
- BVH Construction
- BVH Traversal
- Optimizing Construction
- High-speed Traversal

Optimizing Construction



How do we construct a ‘good’ BVH?

What is a good BVH?

➔ One that minimizes the number of ray/primitive intersections, and the number of ray/AABB intersections.

Optimizing Construction

BVH Quality

A good BVH minimizes the number of intersections.

Concrete:

$$Q_{bvh} = \sum_{1}^{N} P_{AABB}(C_{AABB} + N_{tri}C_{tri})$$

Where:

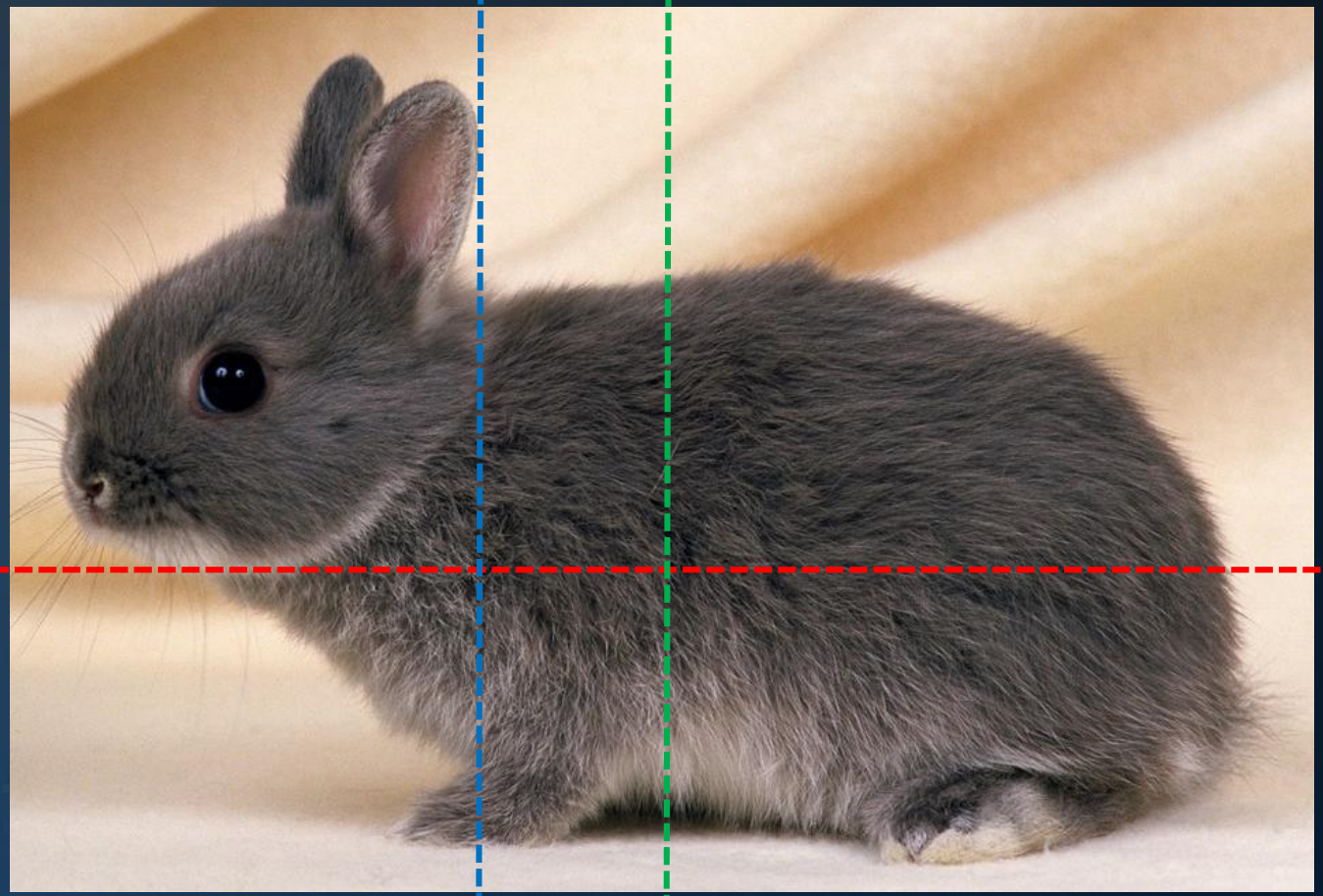
N is the number of BVH nodes;
P_{AABB} is the probability of a ray hitting the AABB;
C_{AABB} is the cost of a ray intersecting the AABB;
N_{tri} is the number of triangles in the node;
C_{tri} is the cost of intersecting a triangle.

Probability of hitting an AABB with an arbitrary ray:

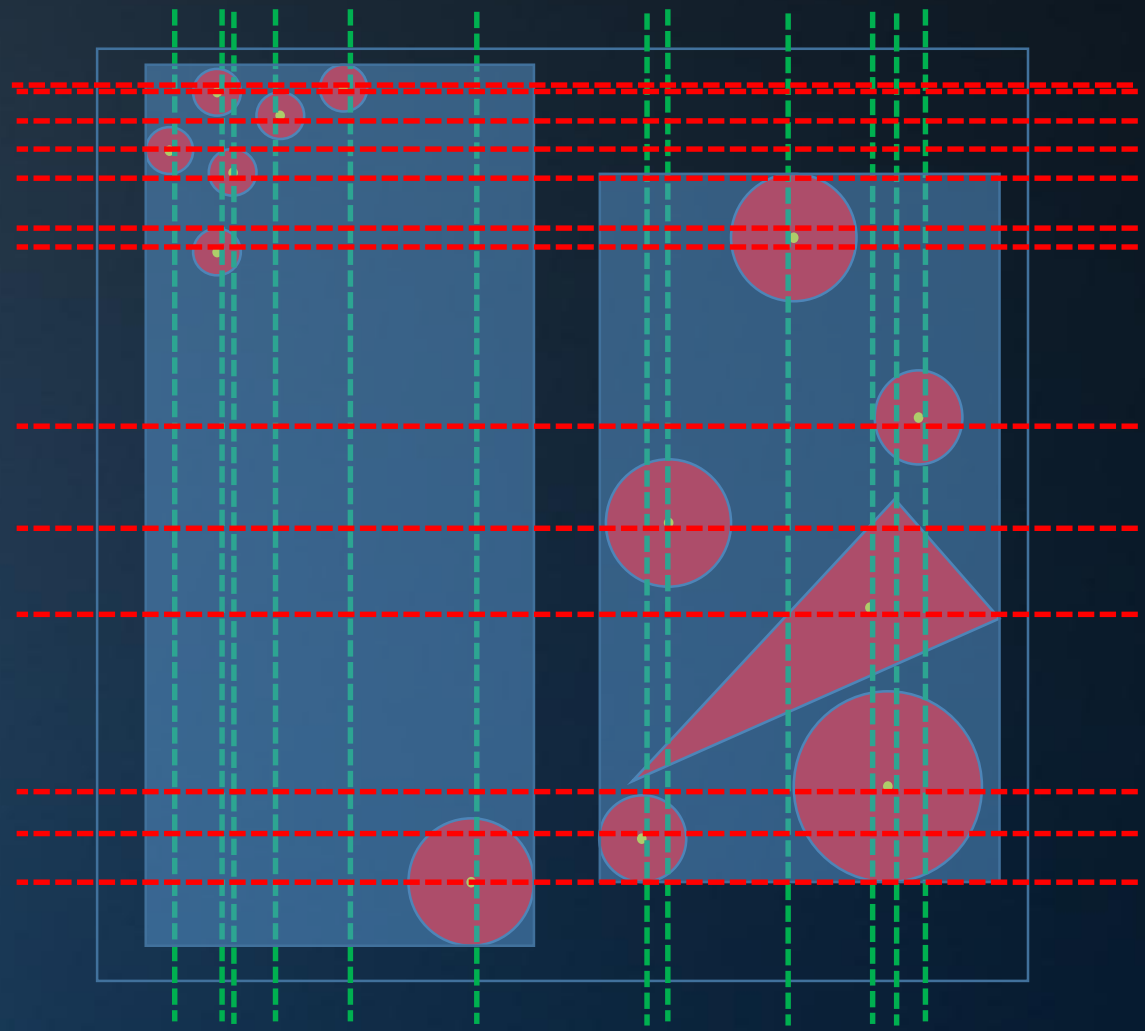
Proportional to the surface area of the AABB.

Binned BVH Construction

Surface Area Heuristic *(Or: what is the best way to slice a bunny?)*



Binned BVH Construction



Cost:

$N_{left} * A_{left} + N_{right} * A_{right}$

Select the split with the lowest cost.

Optimizing Construction

Surface Area Heuristic

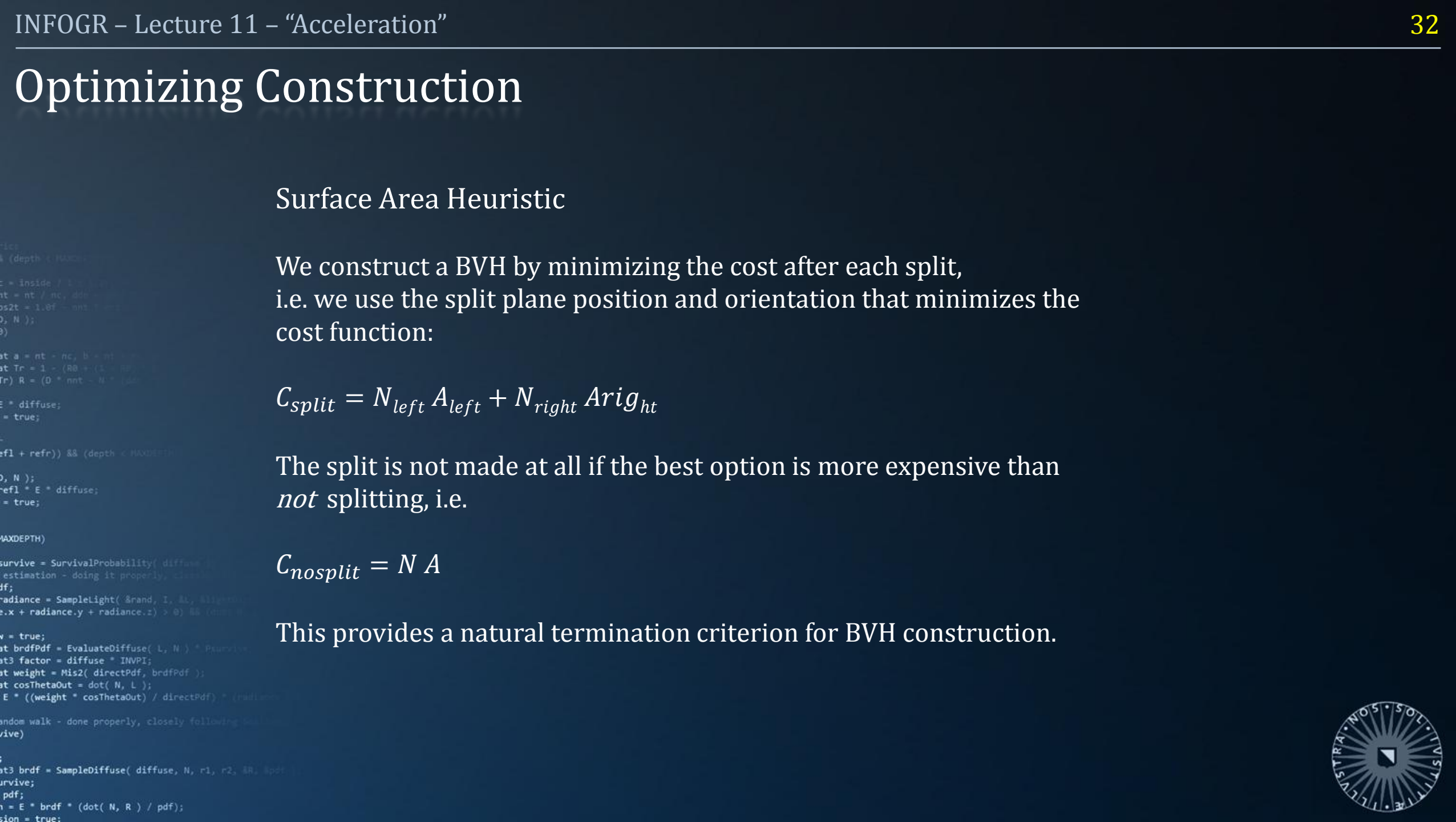
We construct a BVH by minimizing the cost after each split,
i.e. we use the split plane position and orientation that minimizes the cost function:

$$C_{split} = N_{left}\, A_{left} + N_{right}\, Arig_{ht}$$

The split is not made at all if the best option is more expensive than *not* splitting, i.e.

$$C_{nosplit} = N\, A$$

This provides a natural termination criterion for BVH construction.



Optimizing Construction

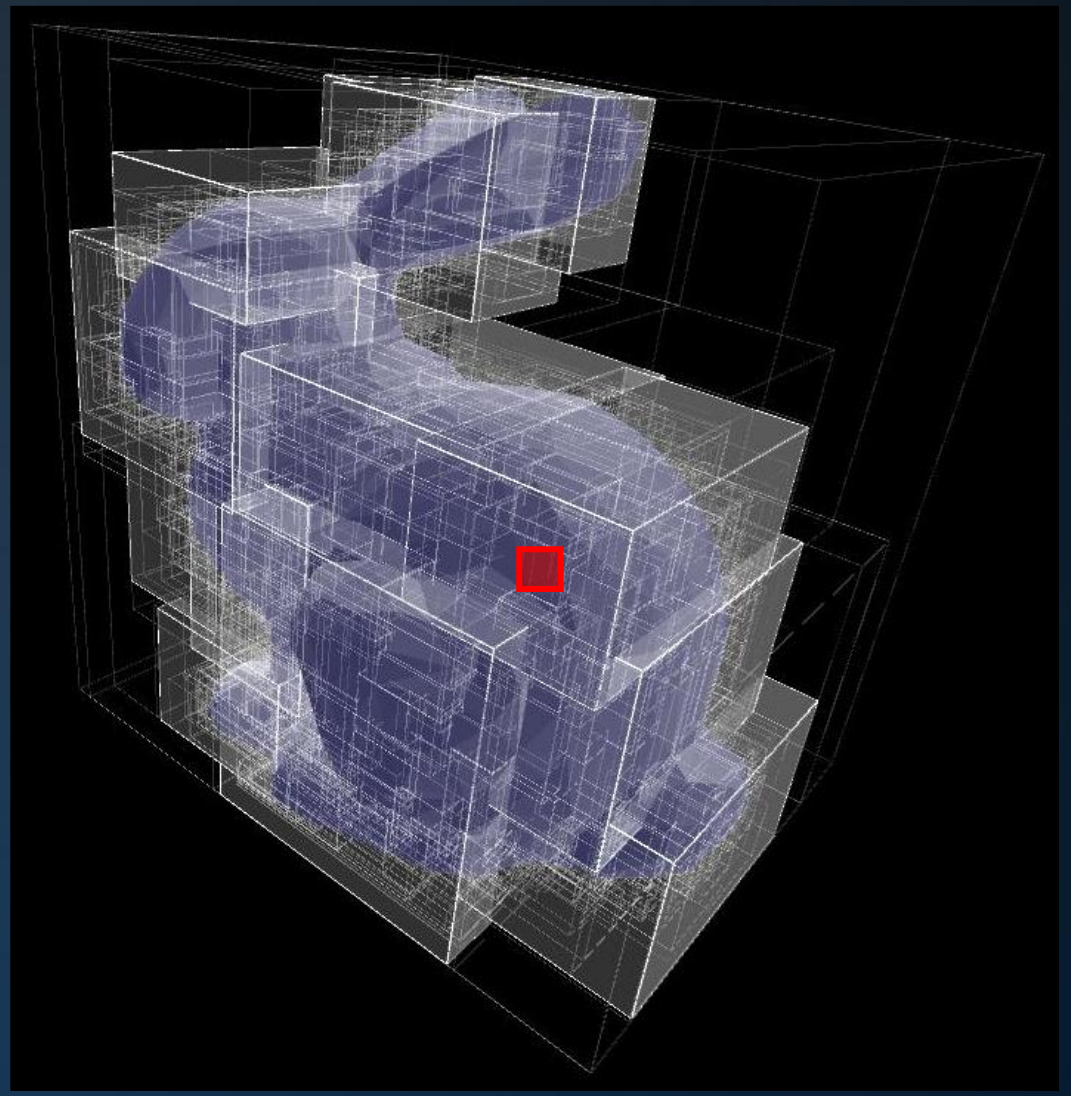
Efficiency of the Surface Area Heuristic

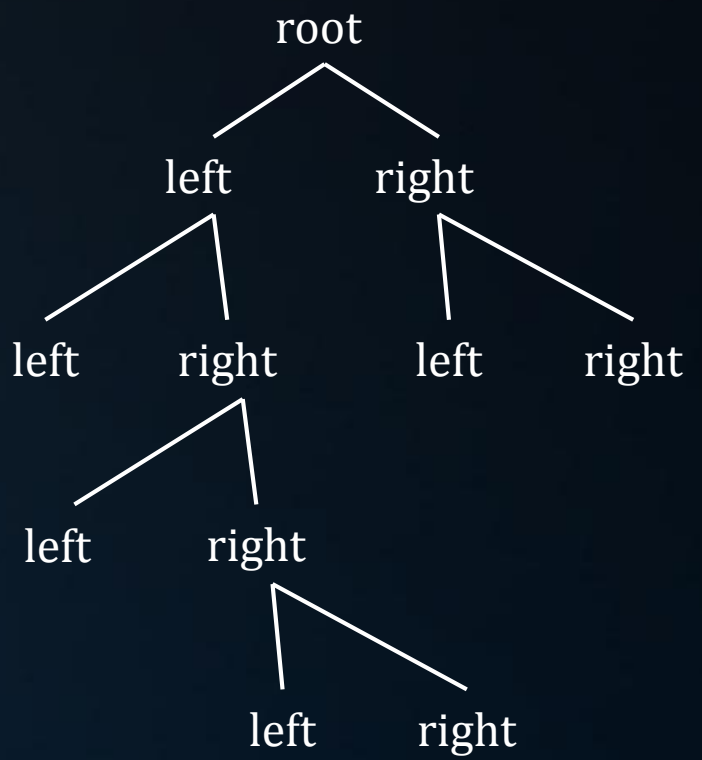
A BVH constructed with the Surface Area Heuristic is typically *twice as efficient* as a tree constructed with naïve midpoint subdivision.

Today’s Agenda:

- High-speed Ray Tracing
- Acceleration Structures
- The Bounding Volume Hierarchy
- BVH Construction
- BVH Traversal
- Optimizing Construction
- High-speed Traversal

Fast Traversal





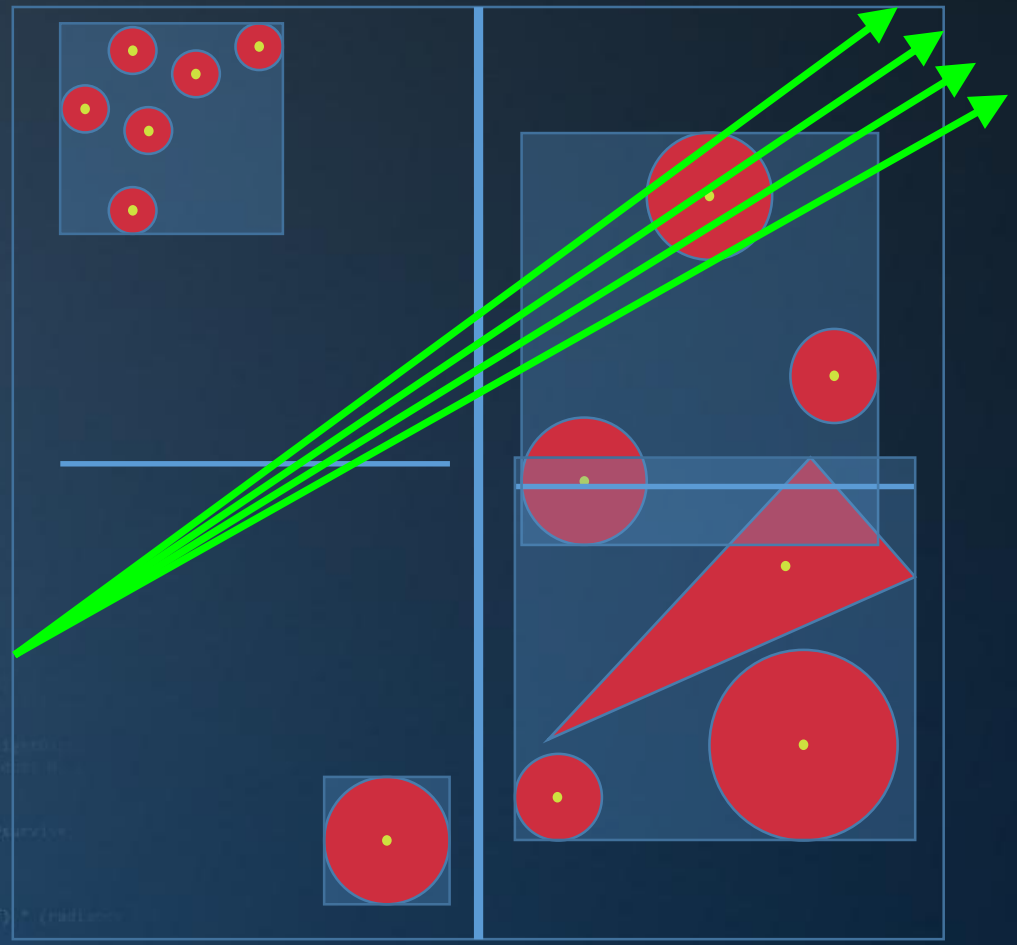
Fast Traversal

Ray Packet Traversal

Primary rays for a small square of pixels tend to travel the same BVH nodes.

We can exploit this by explicitly traversing *ray packets.*

Fast Traversal



Packet Traversal Algorithm:

Starting with the root:

If the node is a leaf node:
    Intersect triangles.
Else:
    If **any** ray intersects the left child AABB:
        Traverse left child
    If **any** ray intersects the right child AABB:
        Traverse right child

Fast Traversal

Ray Packet Traversal

Quickly determining if *any* ray intersects a node:

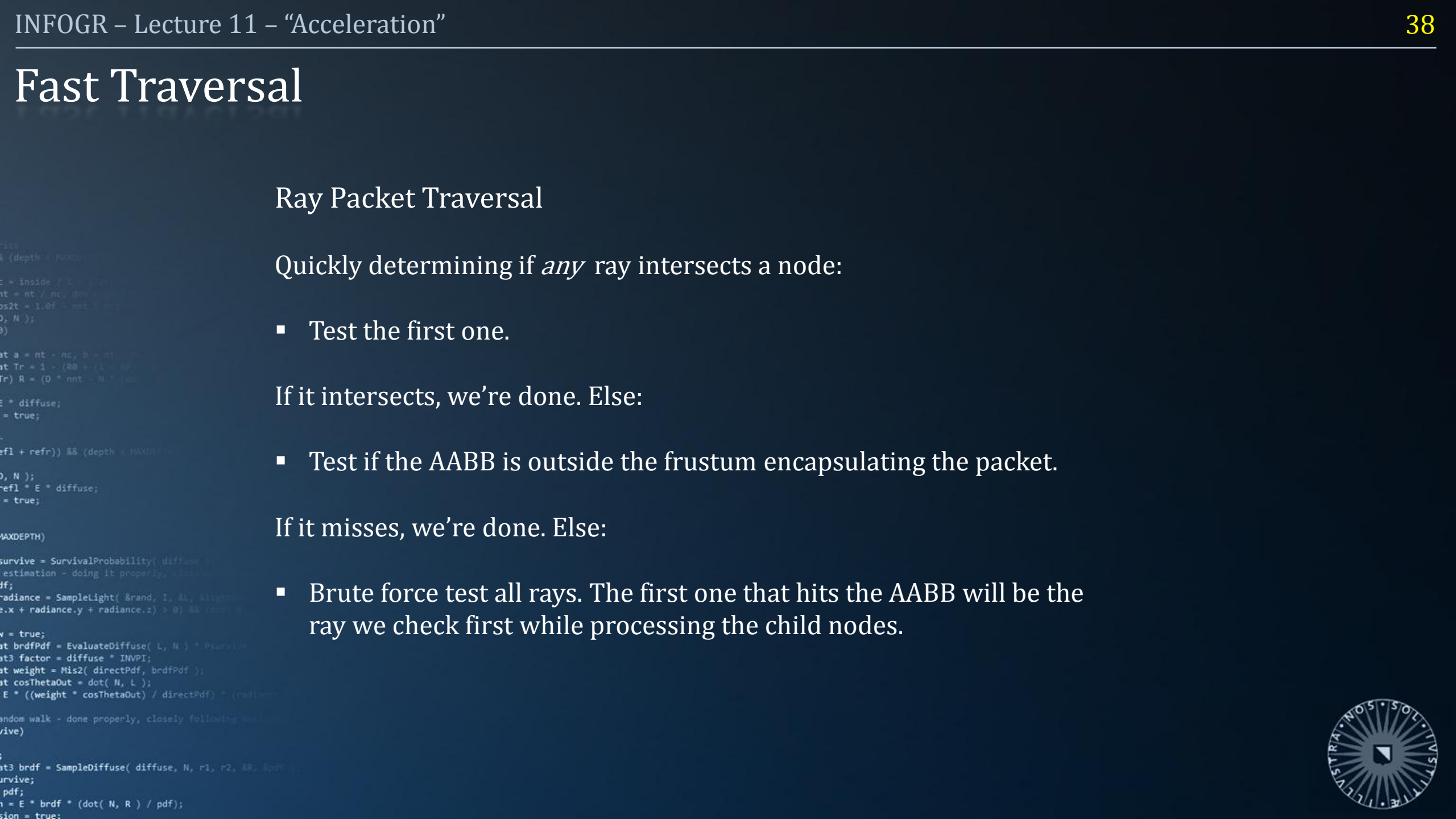
- Test the first one.

If it intersects, we’re done. Else:

- Test if the AABB is outside the frustum encapsulating the packet.

If it misses, we’re done. Else:

- Brute force test all rays. The first one that hits the AABB will be the ray we check first while processing the child nodes.



Fast Traversal

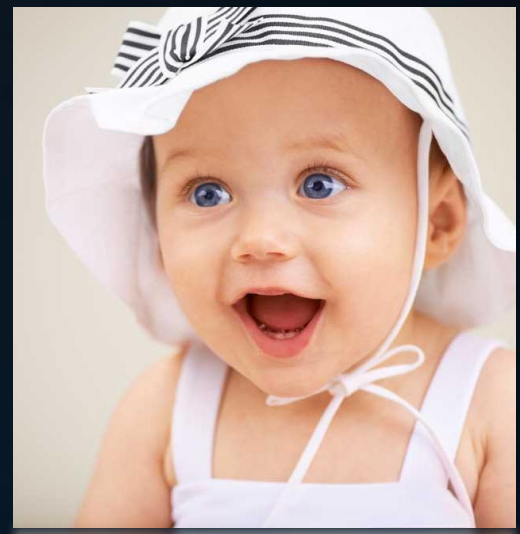
Ray Packet Traversal Efficiency

Using the packet traversal approach, we can very efficiently traverse large packets of rays that travel roughly in the same direction. For primary rays, this can be 32x faster than single ray traversal.

Note that this requires the rays in the packet to traverse a similar set of BVH nodes. The ray packet must be *coherent* (as opposed to *divergent*). Ray coherence can be expressed as the extend to which rays in a packet travel the same nodes, or:

$$coherence = \frac{\#rays\ in\ packet}{average\ \#rays\ intersecting\ a\ node}$$

Combined with an efficient BVH, we now have the performance needed for real-time ray tracing.



Today’s Agenda:

- High-speed Ray Tracing
- Acceleration Structures
- The Bounding Volume Hierarchy
- BVH Construction
- BVH Traversal
- Optimizing Construction
- High-speed Traversal

INFOGR – Computer Graphics

J. Bikker - April-July 2015 - Lecture 11: “Accelerate”

END of “Accelerate”

next lecture: “Shading Models”