Ray Tracing for Games

Dr. Jacco Bikker - FEL/CVUT, Prague, March 9 - 20

Fast Ray Tracing

Nelze to přeložit

Agenda:

- Introduction
- Acceleration Structures
- Ray Packets
- Binned BVH Construction

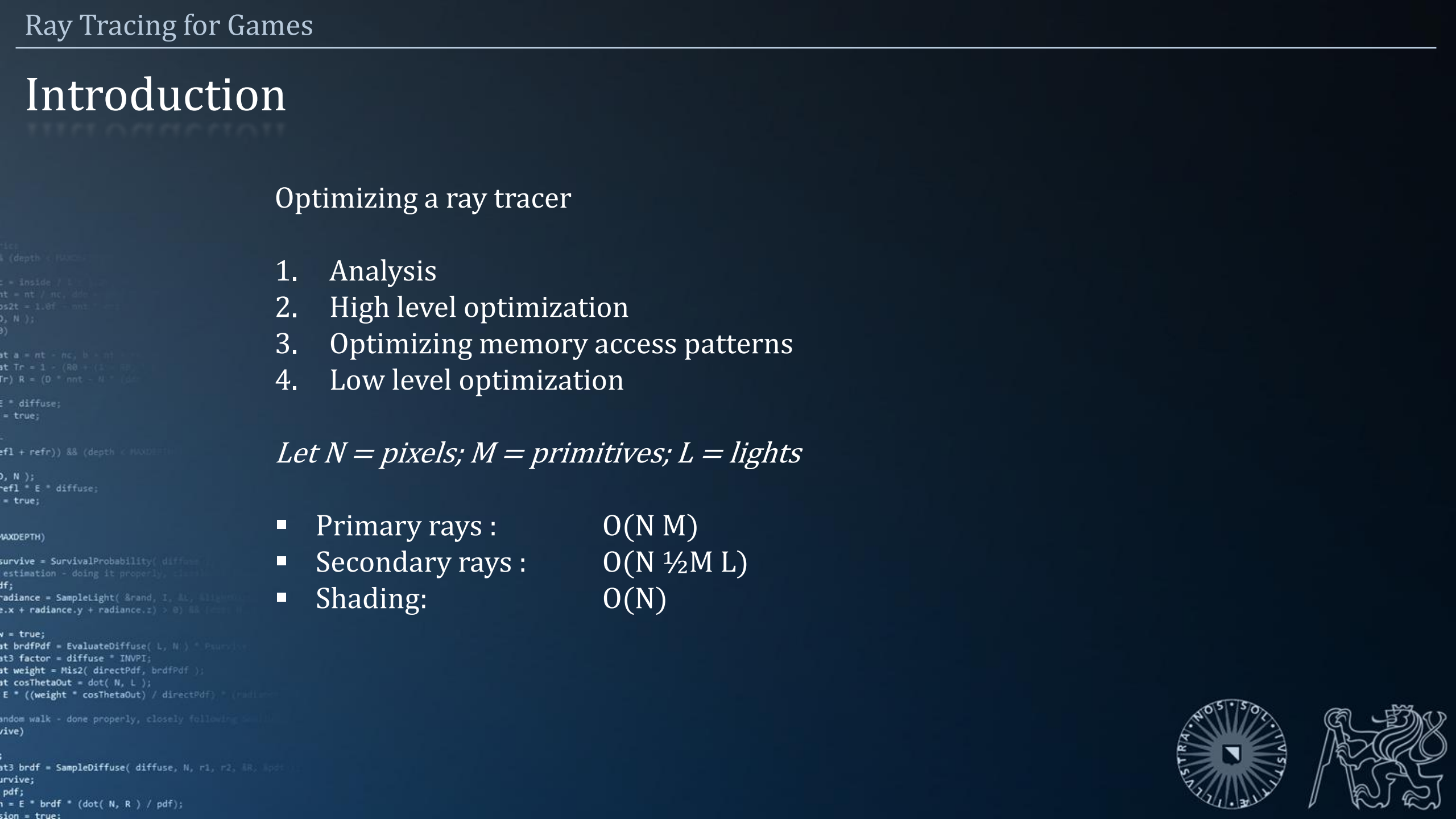
Introduction

Optimizing a ray tracer

1. Analysis
2. High level optimization
3. Optimizing memory access patterns
4. Low level optimization

Let N = pixels; M = primitives; L = lights

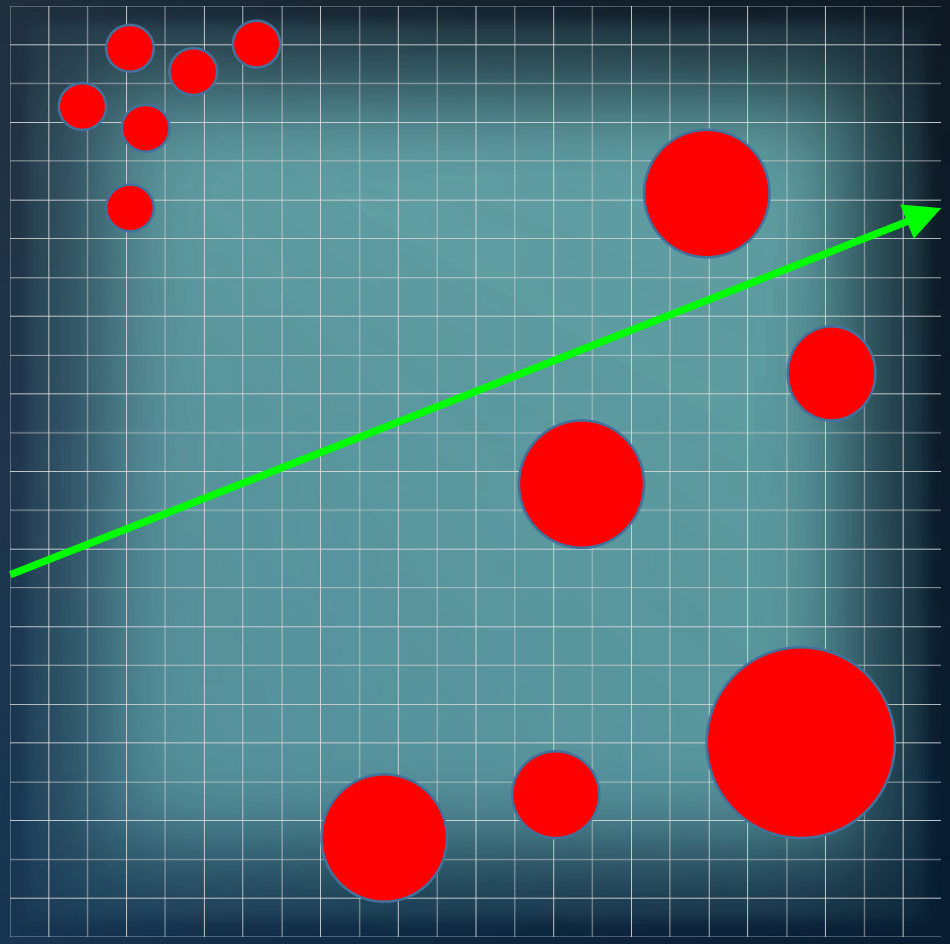
- Primary rays : O(N M)
- Secondary rays : O(N ½M L)
- Shading: O(N)



Agenda:

- Introduction
- Acceleration Structures
- Ray Packets
- Binned BVH Construction

Acceleration Structures



Let N = pixels; M = primitives; L = lights

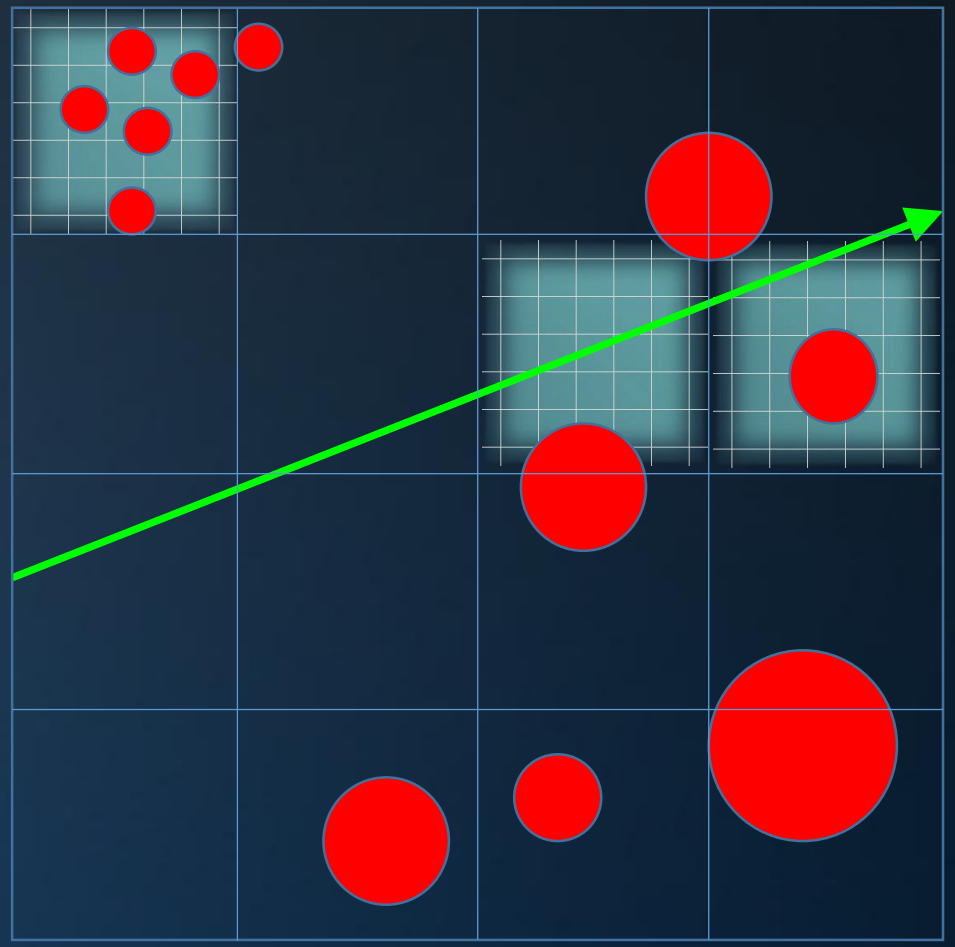
Reducing M by stepping through a regular grid

Note: cost of stepping through the grid is not 0.

Optimal grid resolution: minimizes
$C_{step} * N_{steps} + C_{intersect} * N_{intersect}$

This is scene-dependent, and even region-dependent.

Acceleration Structures

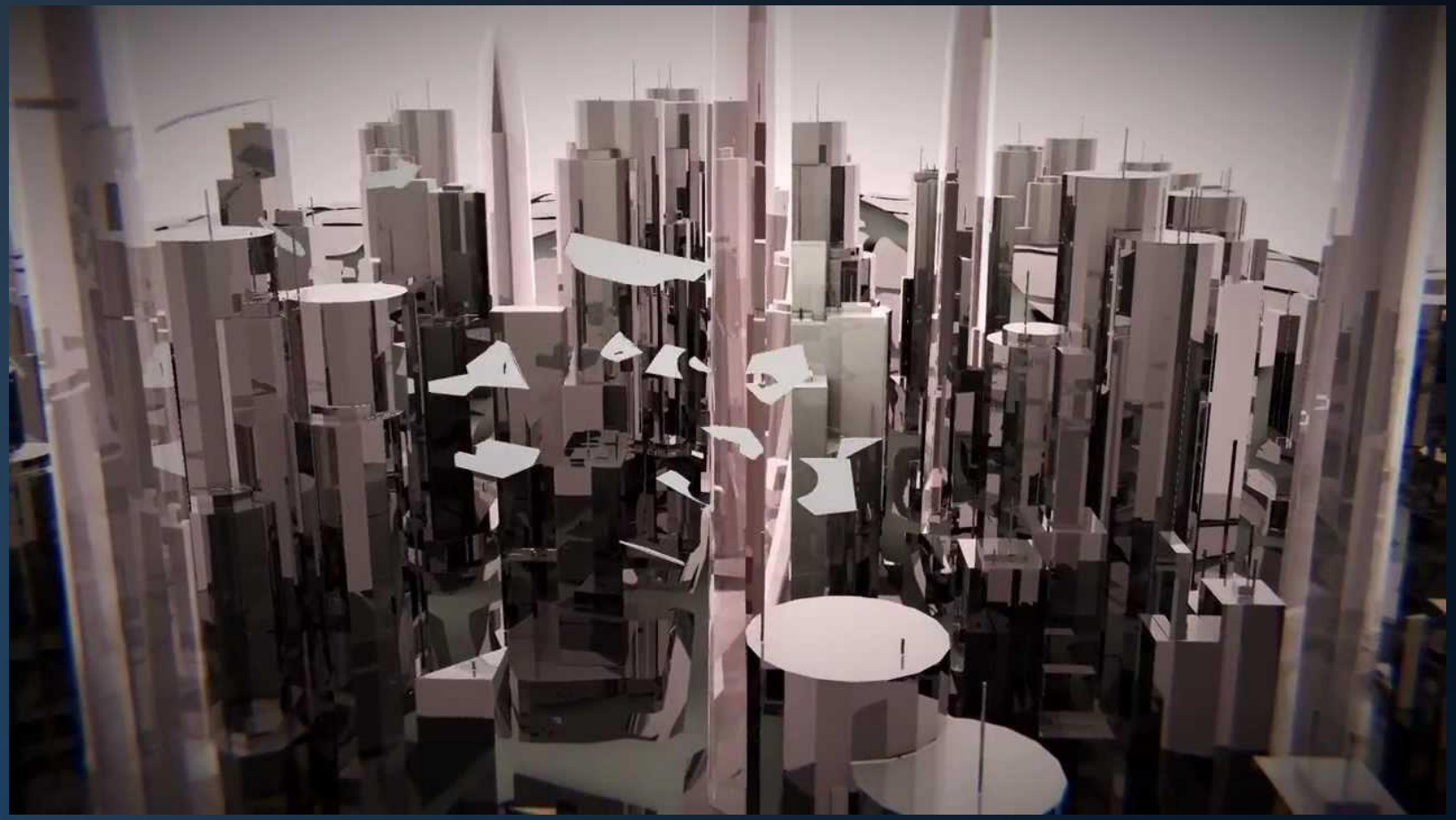


Let N = pixels; M = primitives; L = lights

Reducing M by stepping through a nested grid

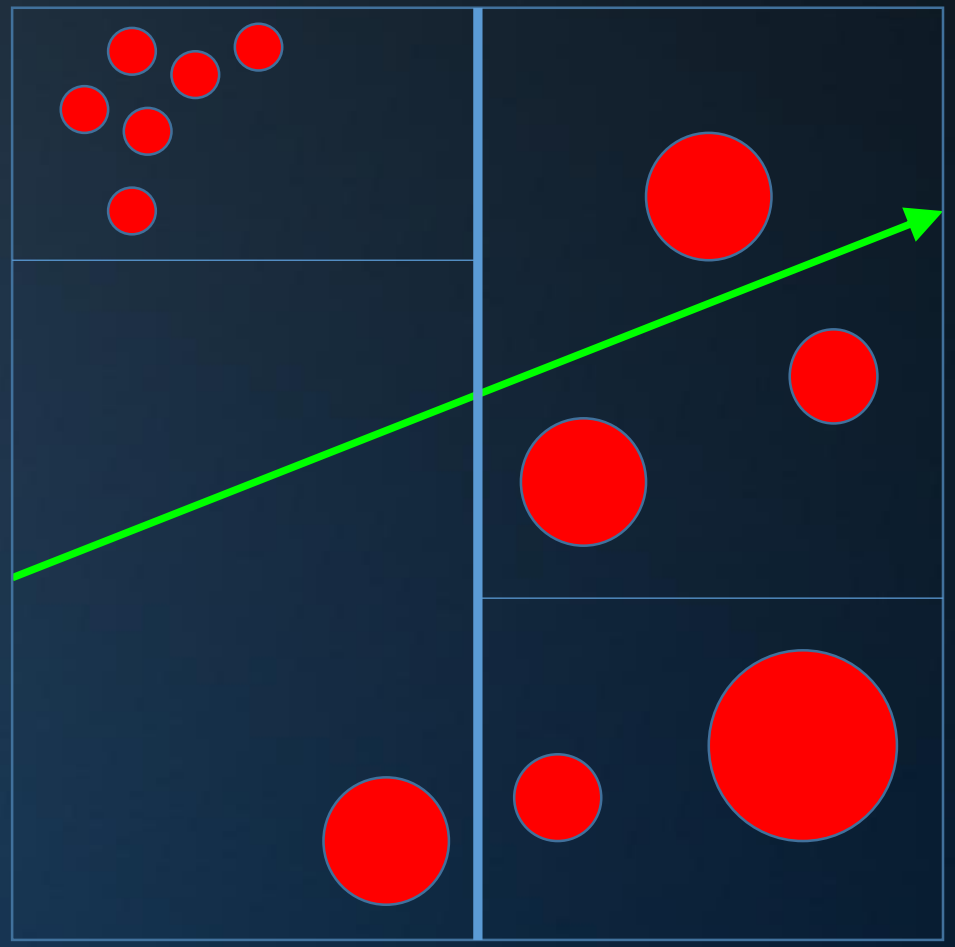
The nested grid adapts itself to local scene complexity, but resolutions still require manual tweaking.

Acceleration Structures



https://directtovideo.wordpress.com/2013/05/08/real-time-ray-tracing-part-2

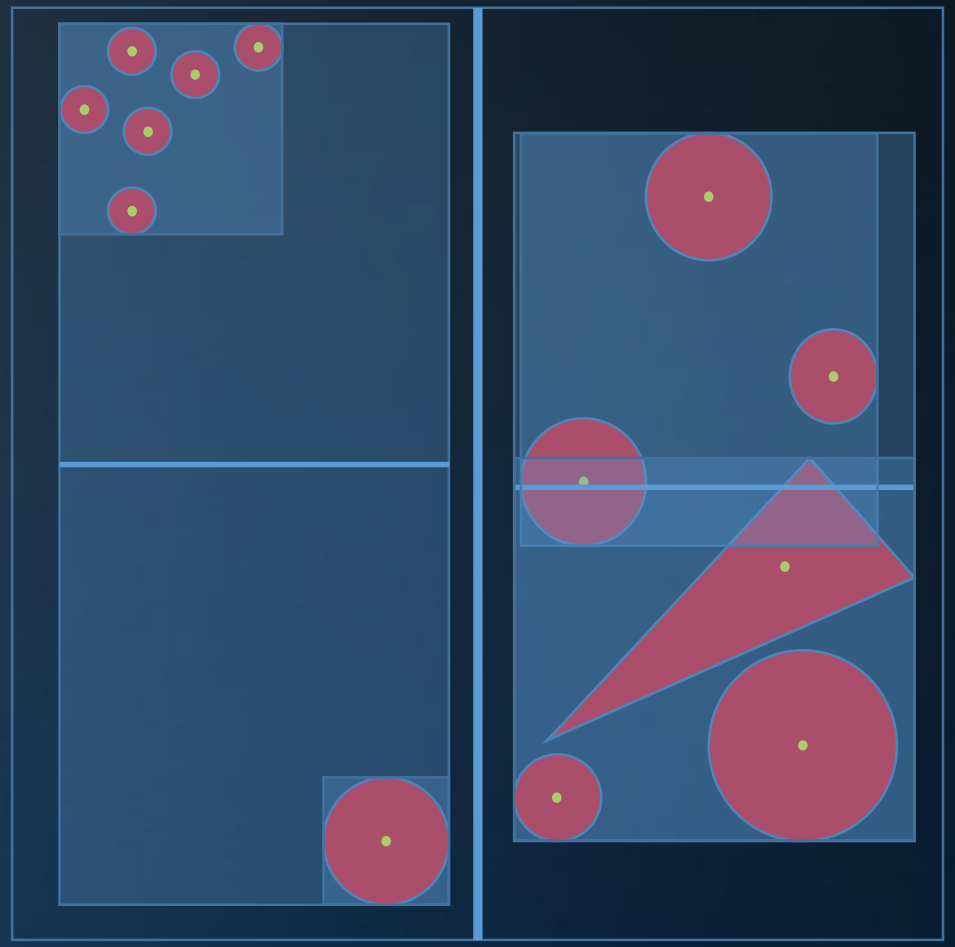
Acceleration Structures



Reducing M by traversing a binary tree

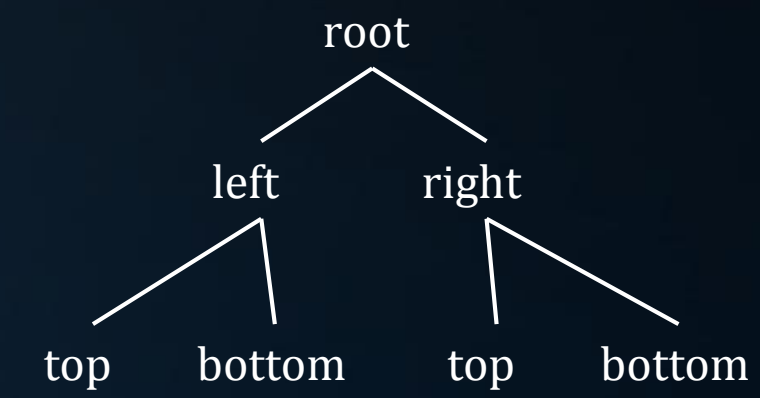
Let N = pixels; M = primitives; L = lights

Acceleration Structures



Let N = pixels; M = primitives; L = lights

Reducing M by traversing a binary tree



Acceleration Structures

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;    // ?
};
```

Acceleration Structures

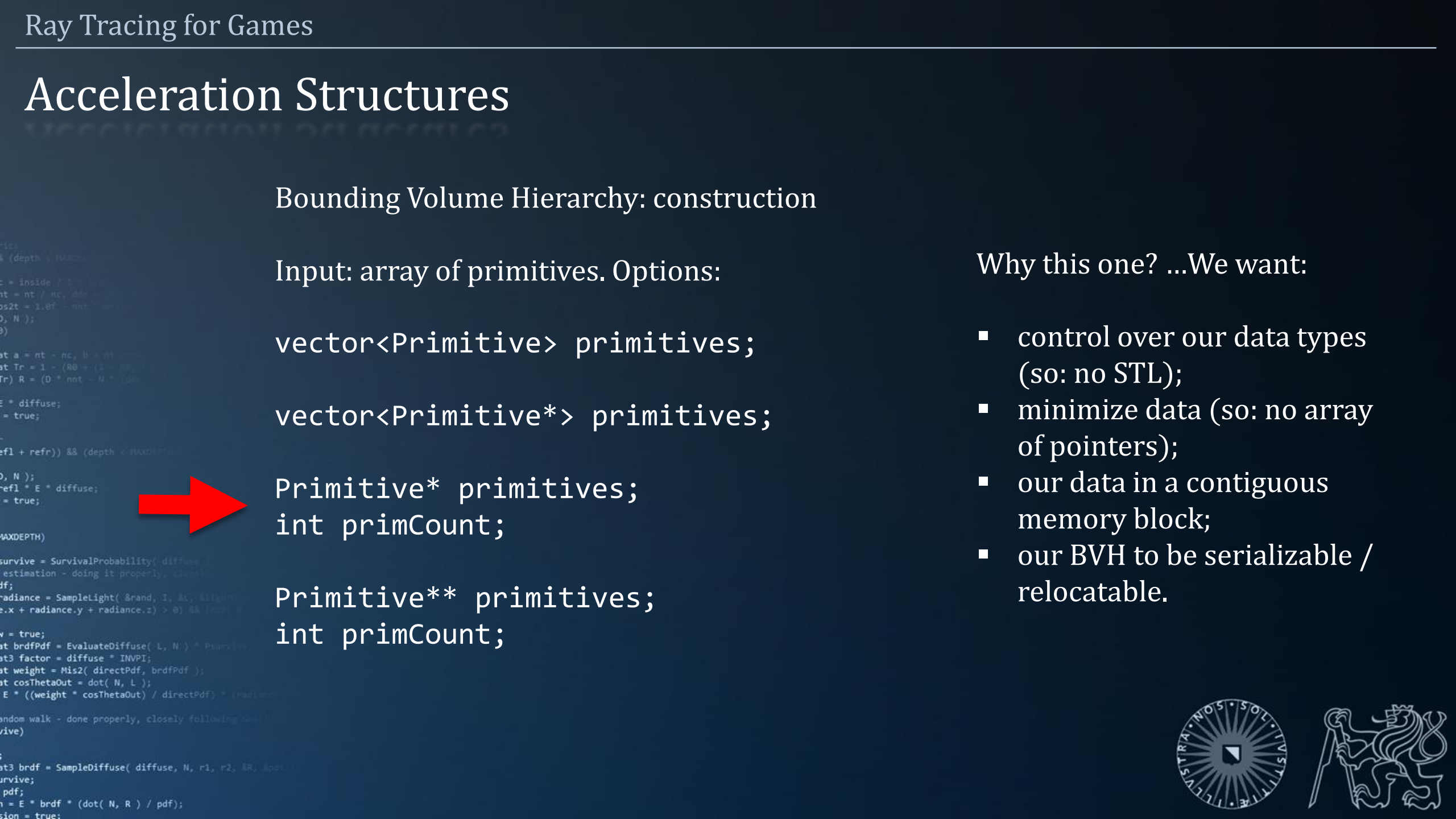
Bounding Volume Hierarchy: construction

Input: array of primitives. Options:

```
vector<Primitive> primitives;

vector<Primitive*> primitives;

Primitive* primitives;
int primCount;

Primitive** primitives;
int primCount;
```

Why this one? …We want:

- control over our data types (so: no STL);
- minimize data (so: no array of pointers);
- our data in a contiguous memory block;
- our BVH to be serializable / relocatable.



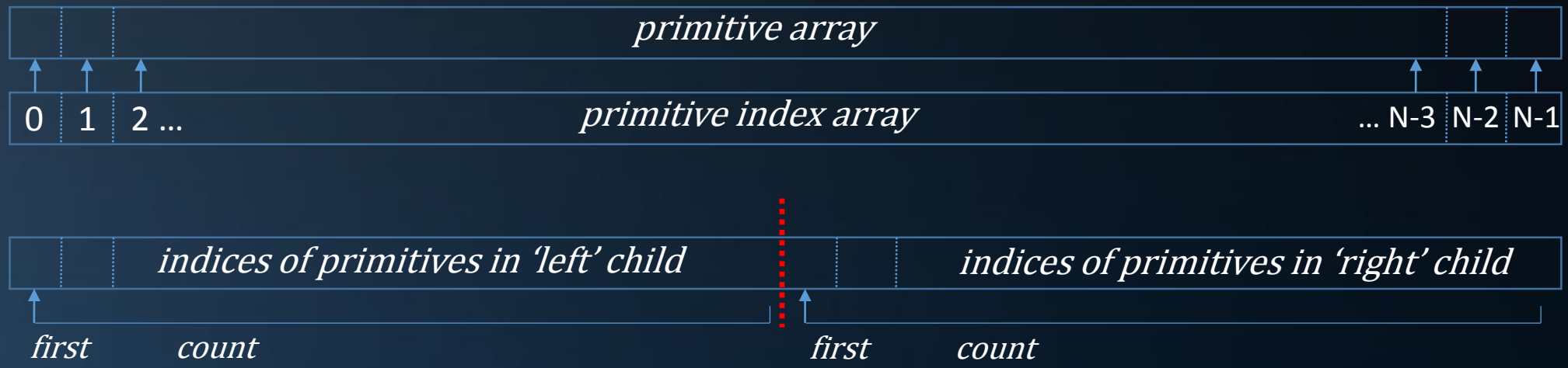
Acceleration Structures

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;
}
```

Acceleration Structures

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?)

Acceleration Structures

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
```

0	1	2 …	*BVH node pool*		

Acceleration Structures

Bounding Volume Hierarchy: data structure

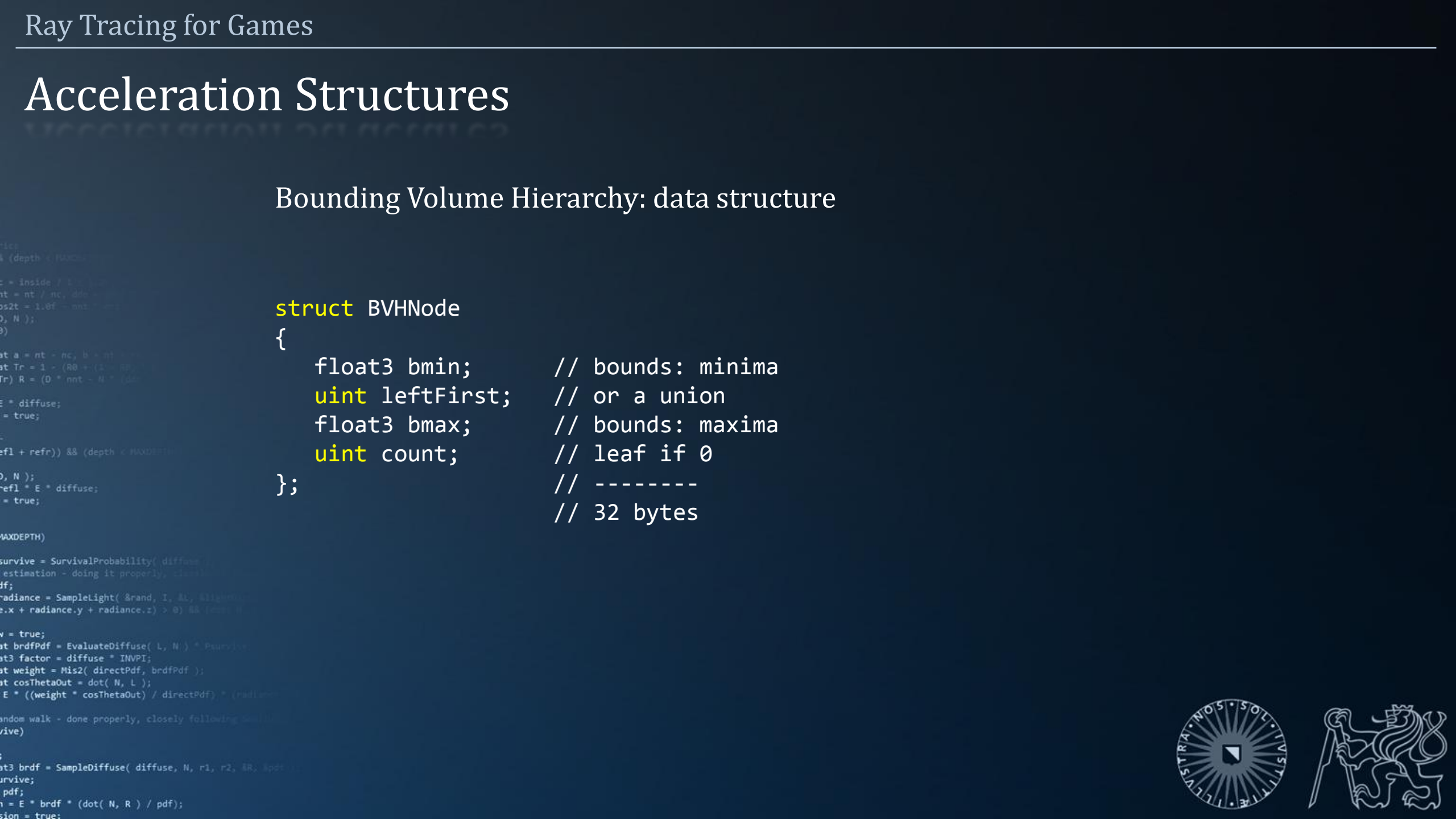
```
struct BVHNode
{
   union                // 4 bytes
   {
      uint left;
      uint first;
   };
   aabb bounds;         // 24 bytes
   uint count;          //  4 bytes
};                      // --------
                        // 32 bytes
```

```
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
```

Acceleration Structures

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
```



Acceleration Structures

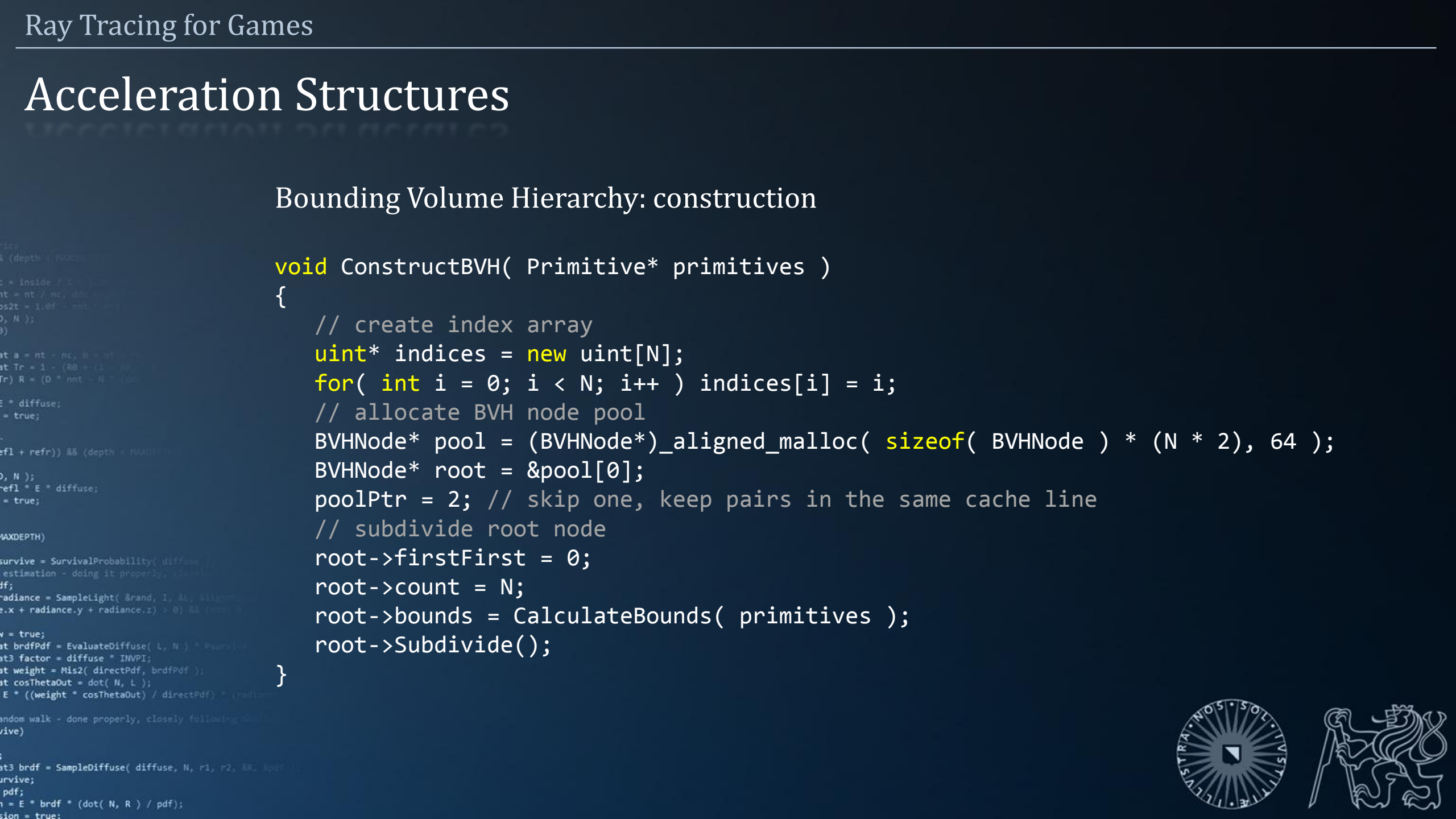
Bounding Volume Hierarchy: data structure

```
struct BVHNode
{
    union
    {
        struct { float3 bmin; uint leftFirst; };
        __m128 bmin4;
    };
    union
    {
        struct { float3 bmax; uint count; };
        __m128 bmax4;
    };
};
```

Acceleration Structures

Bounding Volume Hierarchy: construction

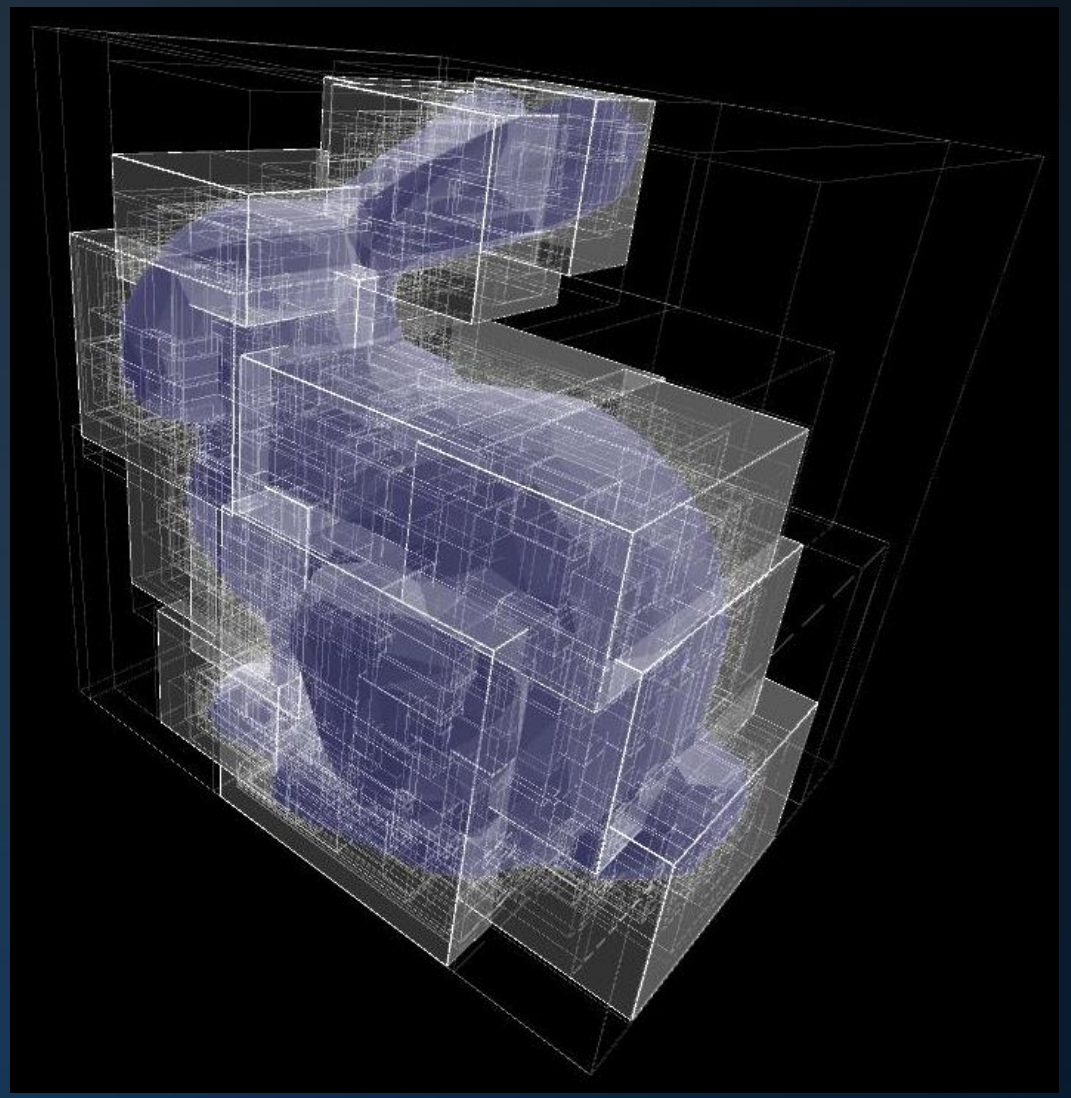
```
void ConstructBVH( Primitive* primitives )
{
   // create index array
   uint* indices = new uint[N];
   for( int i = 0; i < N; i++ ) indices[i] = i;
   // allocate BVH node pool
   BVHNode* pool = (BVHNode*)_aligned_malloc( sizeof( BVHNode ) * (N * 2), 64 );
   BVHNode* root = &pool[0];
   poolPtr = 2; // skip one, keep pairs in the same cache line
   // subdivide root node
   root->firstFirst = 0;
   root->count = N;
   root->bounds = CalculateBounds( primitives );
   root->Subdivide();
}
```

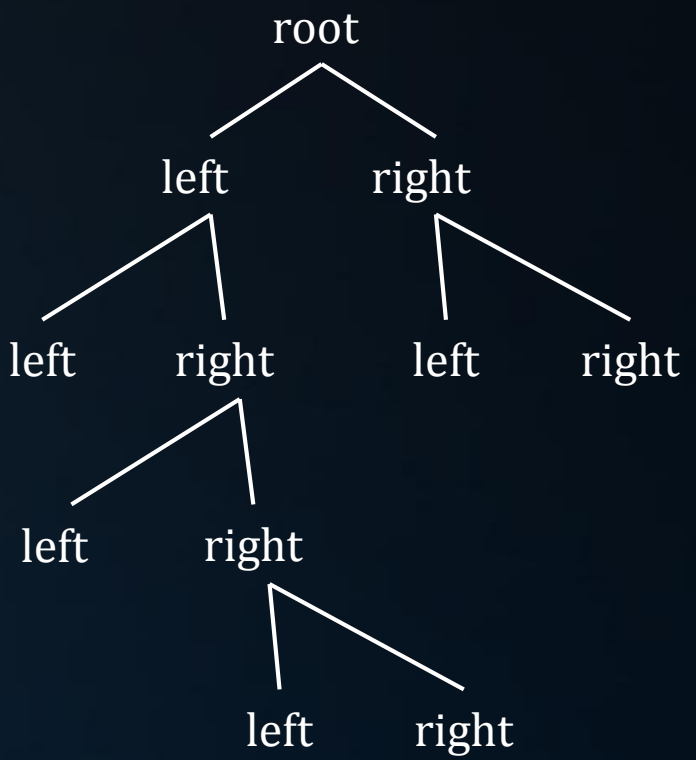


Agenda:

- Introduction
- Acceleration Structures
- Ray Packets
- Binned BVH Construction

Ray Packets





Ray Packets

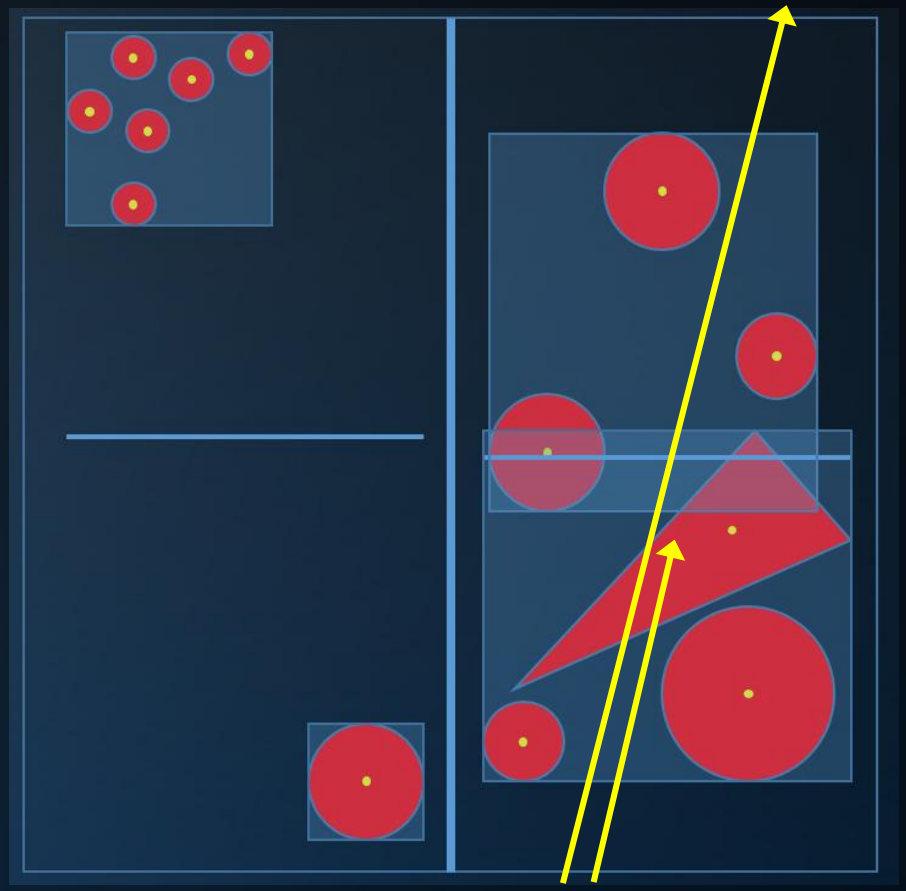
Bounding Volume Hierarchy: traversal

```
BVHNode::Traverse( Ray r, float& t )
```

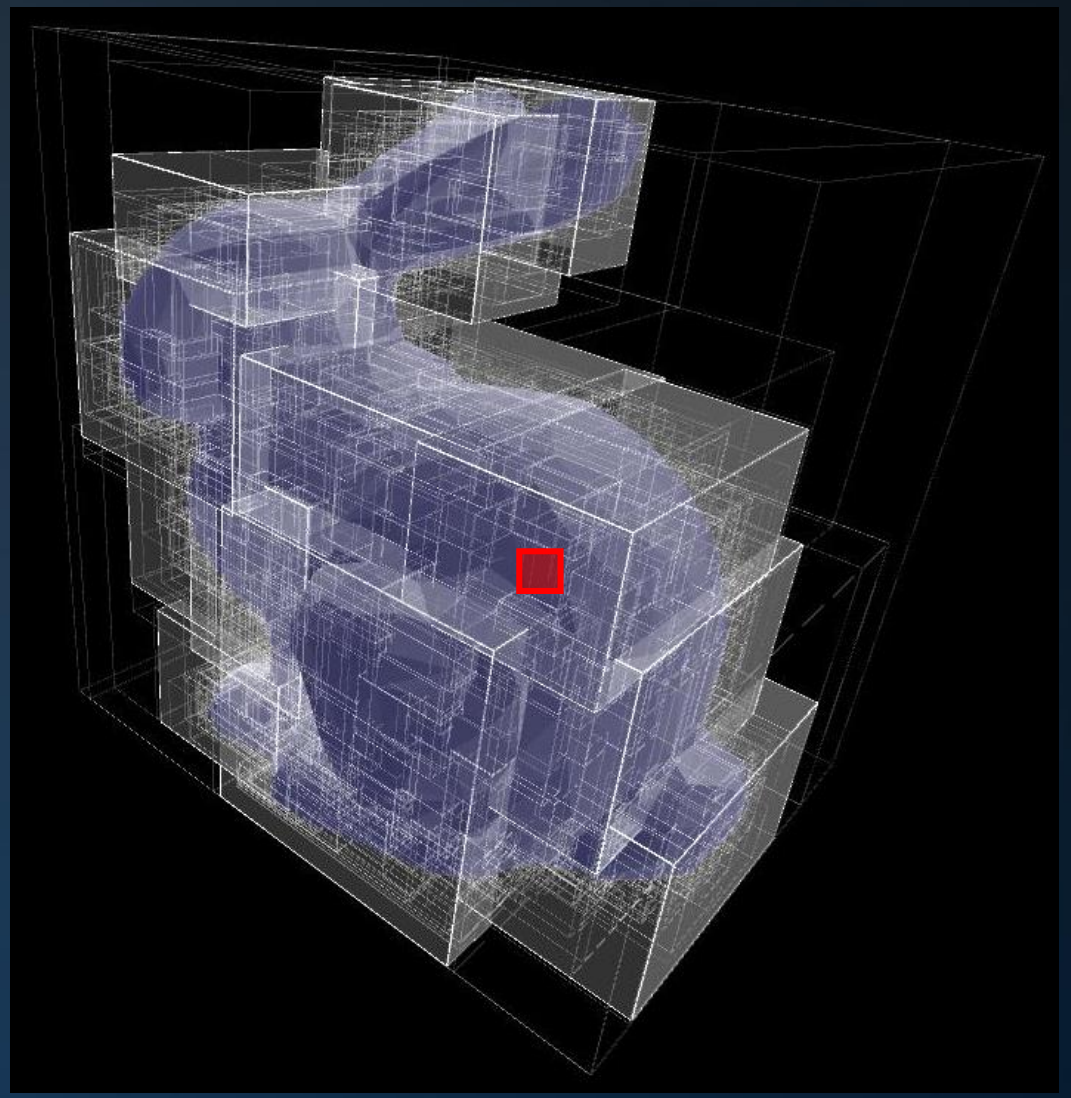
- If ray does not intersect bounds:
  - return
- If node is a leaf:
  - intersect triangles, update *t*
- else:
  - traverse left child
  - traverse right child

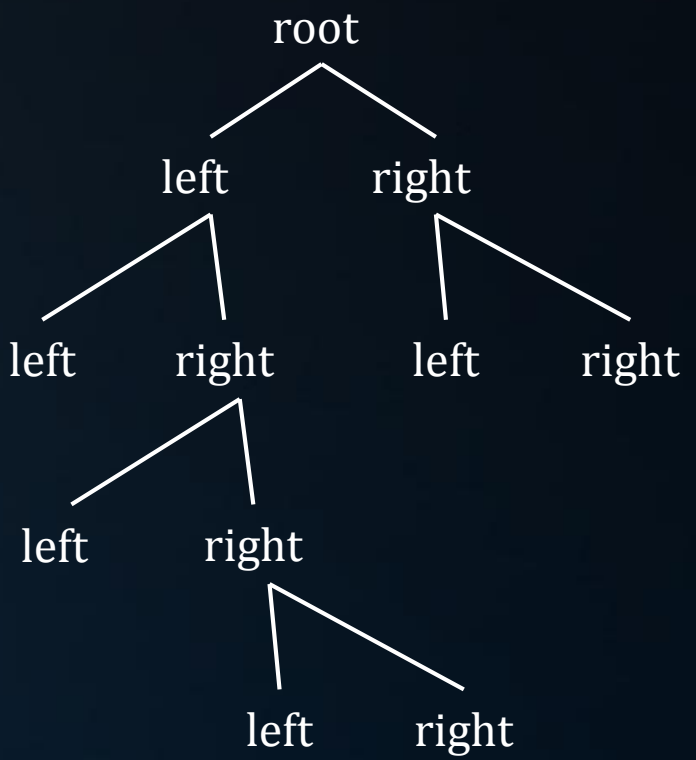
  better:

  - **traverse near child**
  - **traverse far child**



Ray Packets



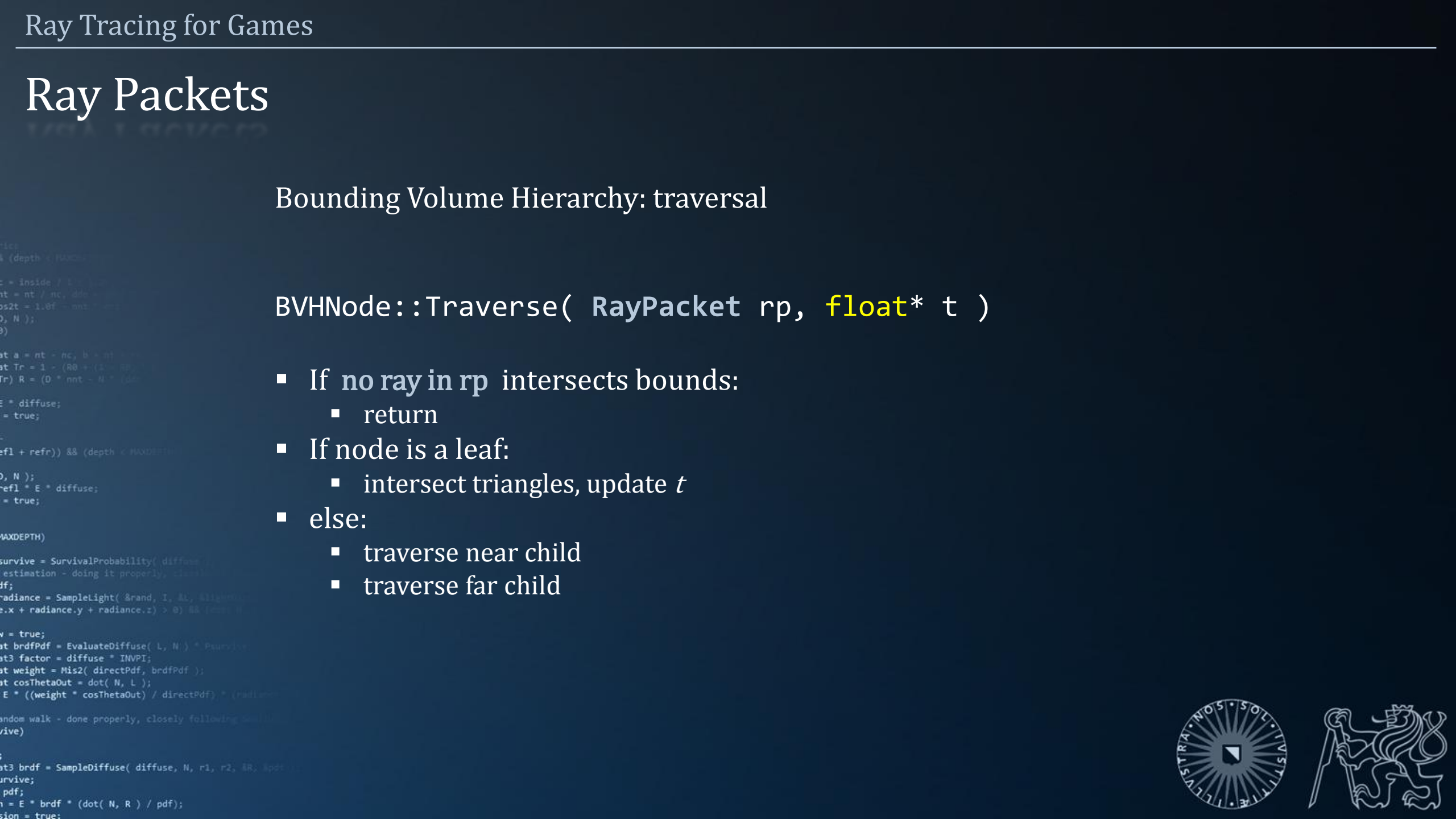


Ray Packets

Bounding Volume Hierarchy: traversal

```
BVHNode::Traverse( RayPacket rp, float* t )
```

- If **no ray in rp** intersects bounds:
  - return
- If node is a leaf:
  - intersect triangles, update *t*
- else:
  - traverse near child
  - traverse far child

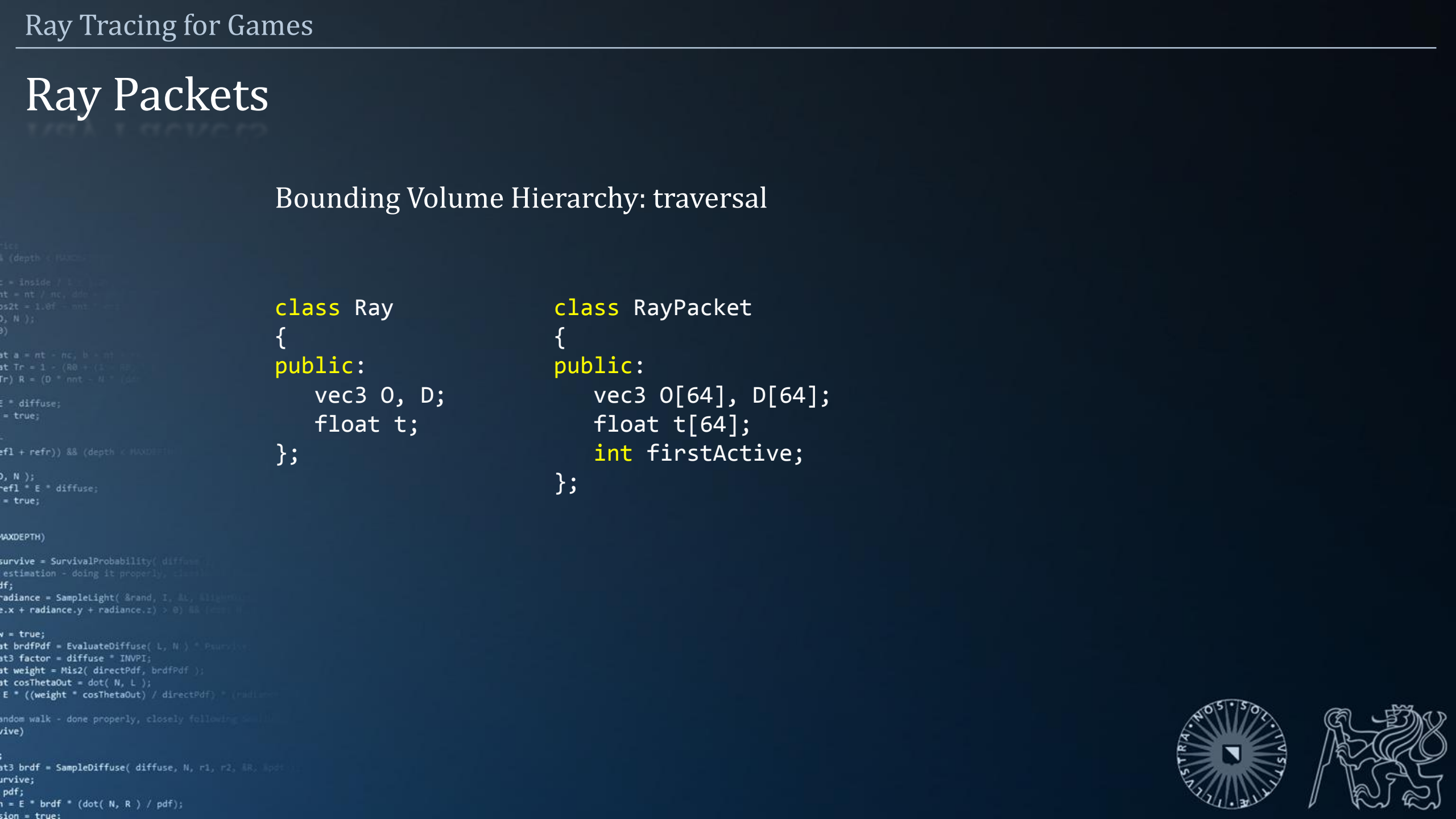


Ray Packets

Bounding Volume Hierarchy: traversal

```
class Ray
{
public:
   vec3 O, D;
   float t;
};
```

```
class RayPacket
{
public:
   vec3 O[64], D[64];
   float t[64];
   int firstActive;
};
```



Ray Packets

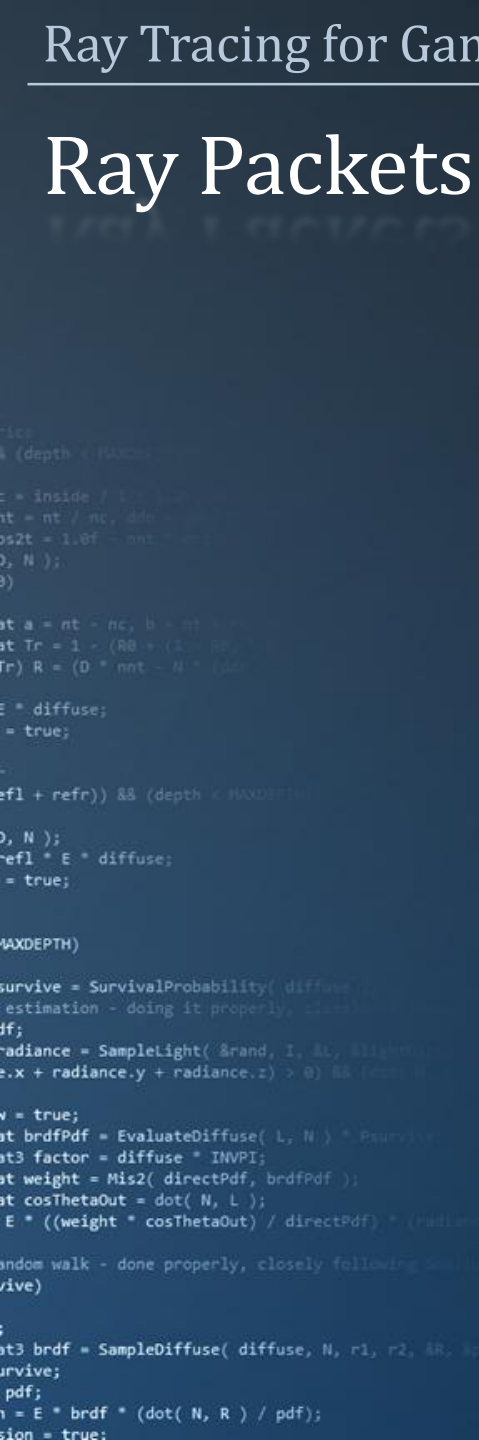
Bounding Volume Hierarchy: traversal

0. `packet.firstActive = 0;`

Traversal loop:

1. Early in: test first active ray against node bounds. If this is a hit:
   - For a leaf: intersect primitives with all remaining rays
   - For an interior node: traverse child nodes.
2. Early in failed: brute force find first ray that intersect this node. If one is found:
   - Update packet.firstActive
   - For a leaf: intersect primitives with all remaining rays
   - For an interior node: traverse child nodes.

Optionally (as in: hard): add an early out test, where the bounding box is tested against a frustum enclosing the ray packet.



Ray Packets

BVH + Ray Packets = *real-time performance*.

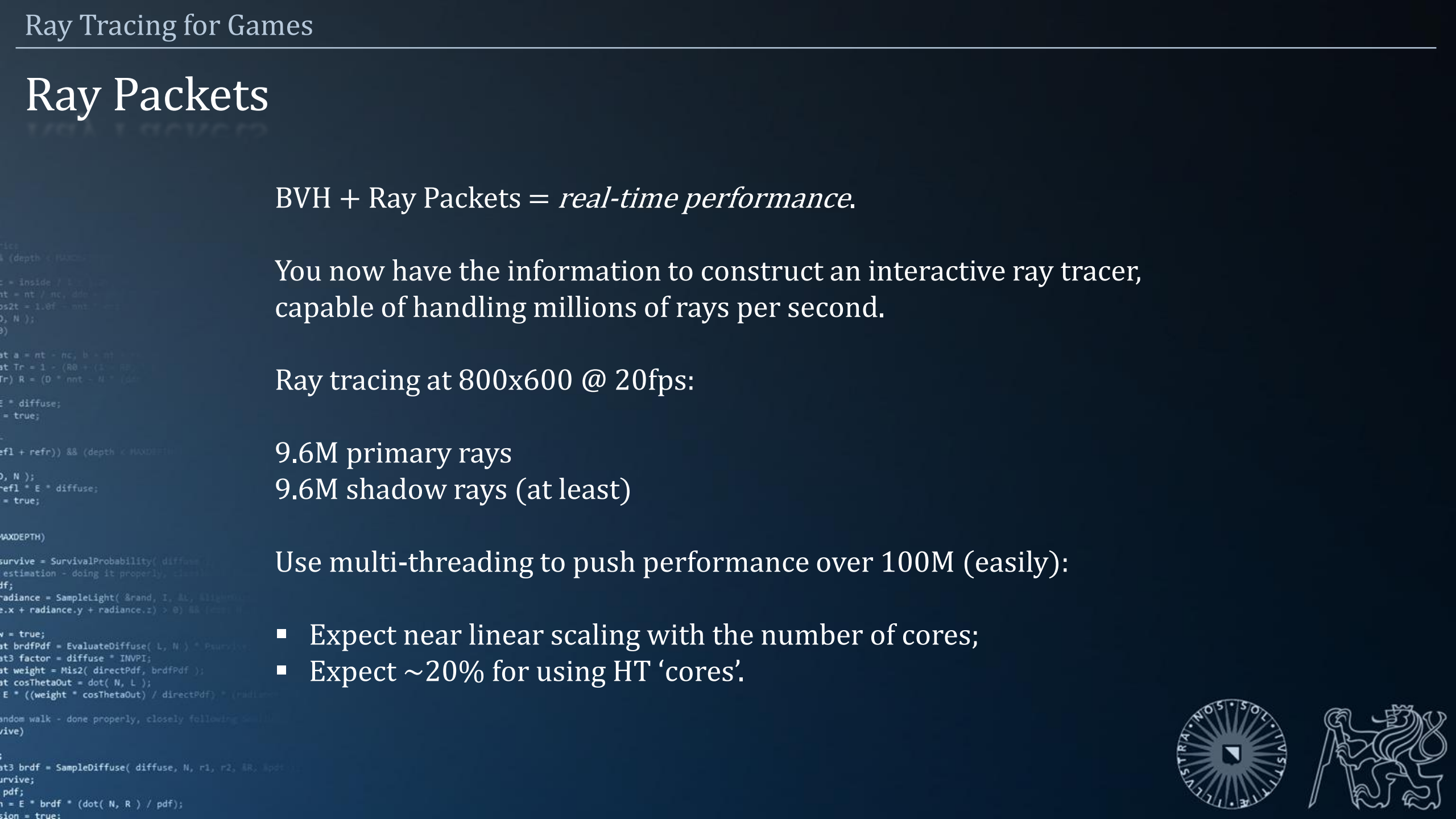
You now have the information to construct an interactive ray tracer, capable of handling millions of rays per second.

Ray tracing at 800x600 @ 20fps:

9.6M primary rays
9.6M shadow rays (at least)

Use multi-threading to push performance over 100M (easily):

- Expect near linear scaling with the number of cores;
- Expect ~20% for using HT ‘cores’.



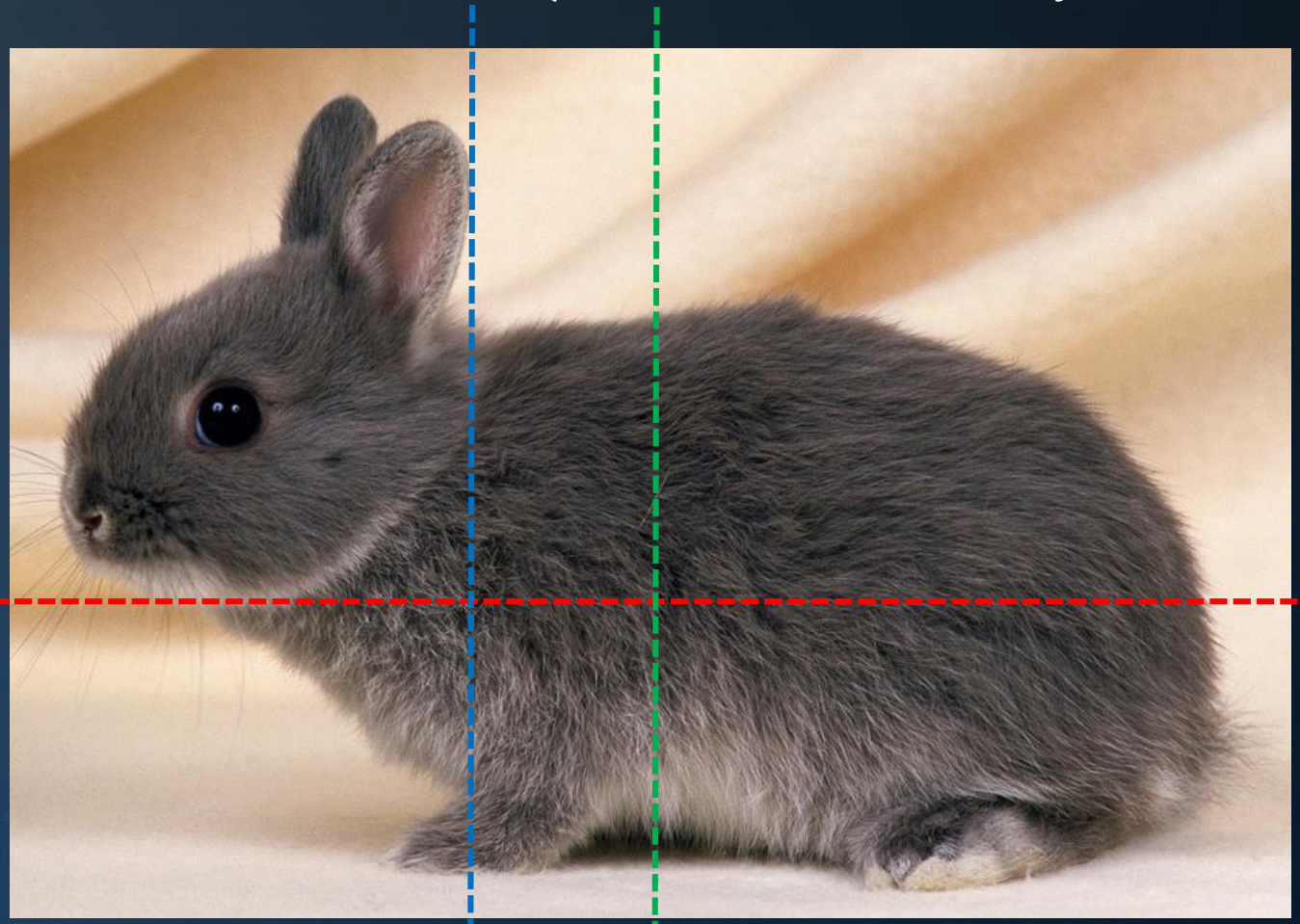
Agenda:

- Introduction
- Acceleration Structures
- Ray Packets
- Binned BVH Construction

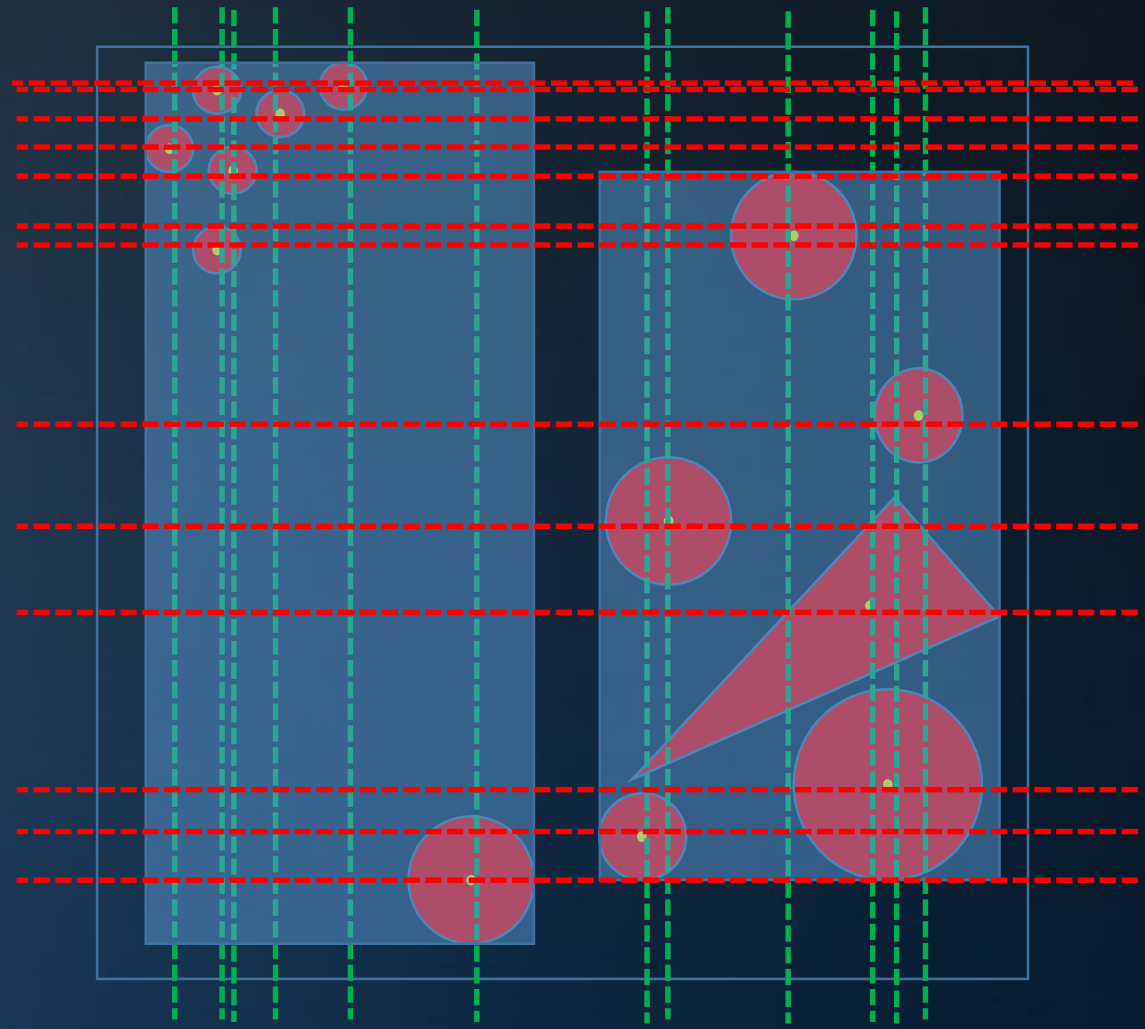
Binned BVH Construction

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.

Ray Packets

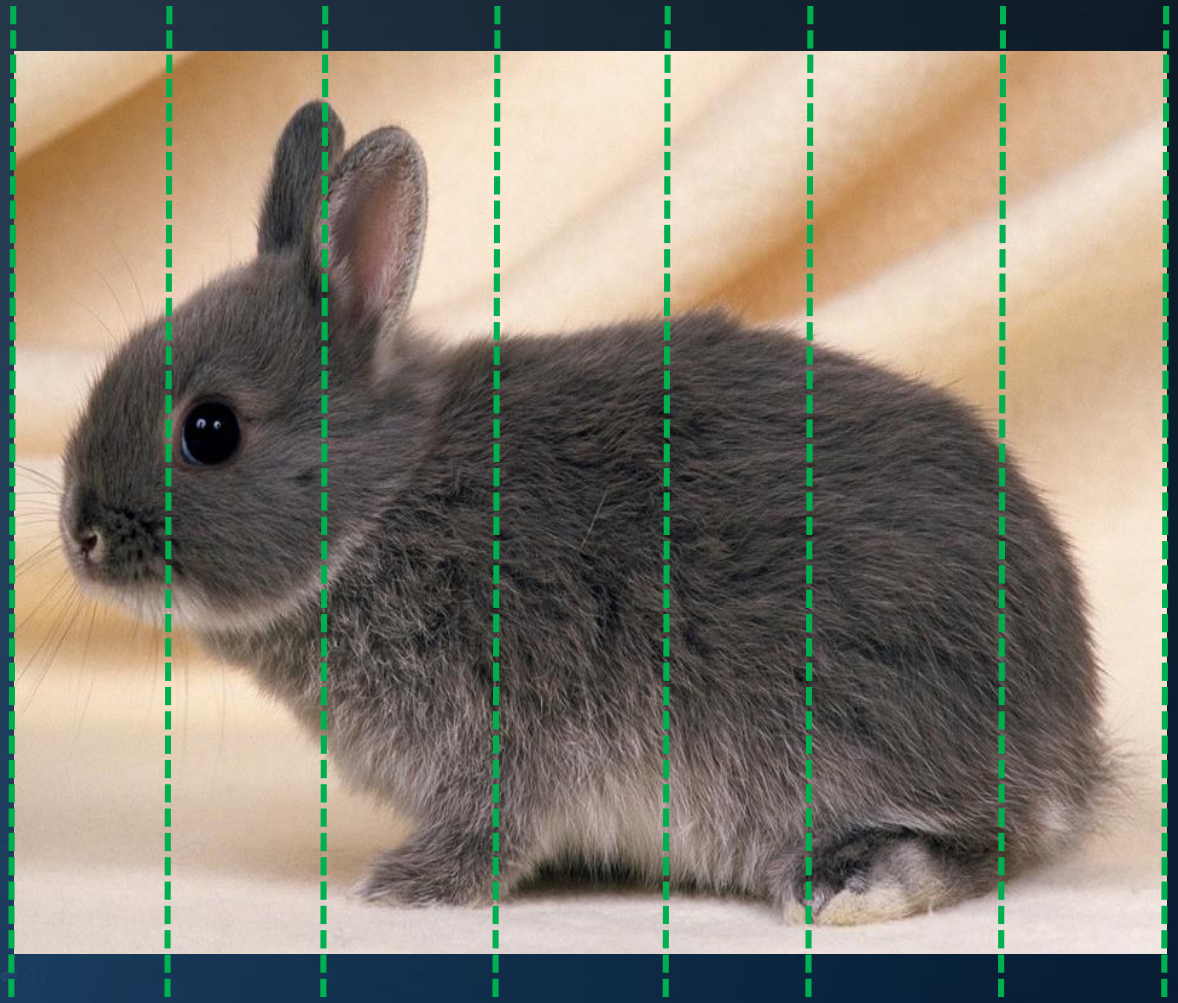
BVH + Ray Packets = *real-time performance*.

Using the Surface Area Heuristic will *double* your performance.

It will also lead to much longer BVH construction times.

We need to go faster.

Binned BVH Construction



Binned SAH:

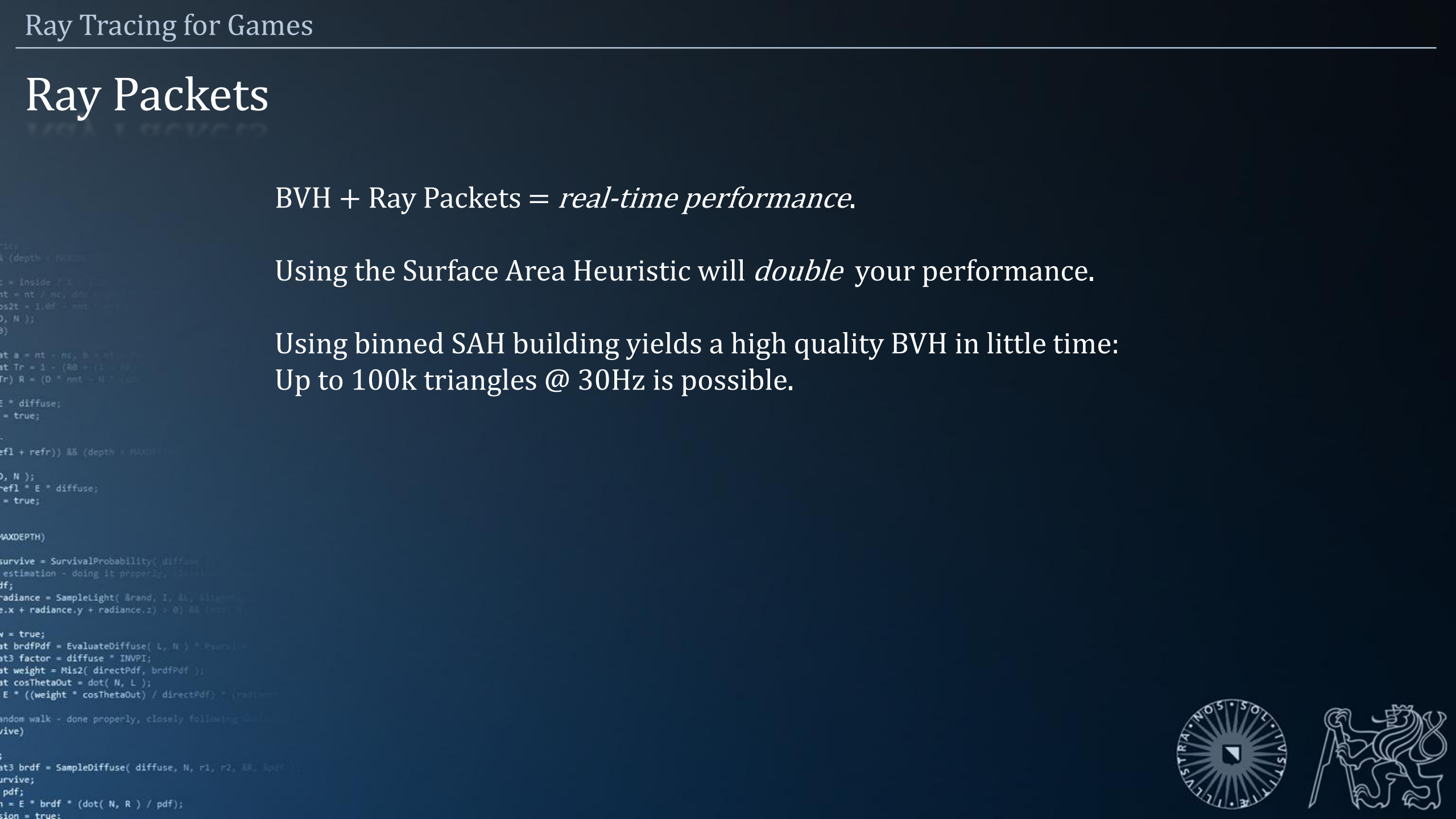
- Select axis of greatest extend
- Evaluate cost function at N discrete intervals
- Select best split plane.

Ray Packets

BVH + Ray Packets = *real-time performance*.

Using the Surface Area Heuristic will *double* your performance.

Using binned SAH building yields a high quality BVH in little time:
Up to 100k triangles @ 30Hz is possible.



Agenda:

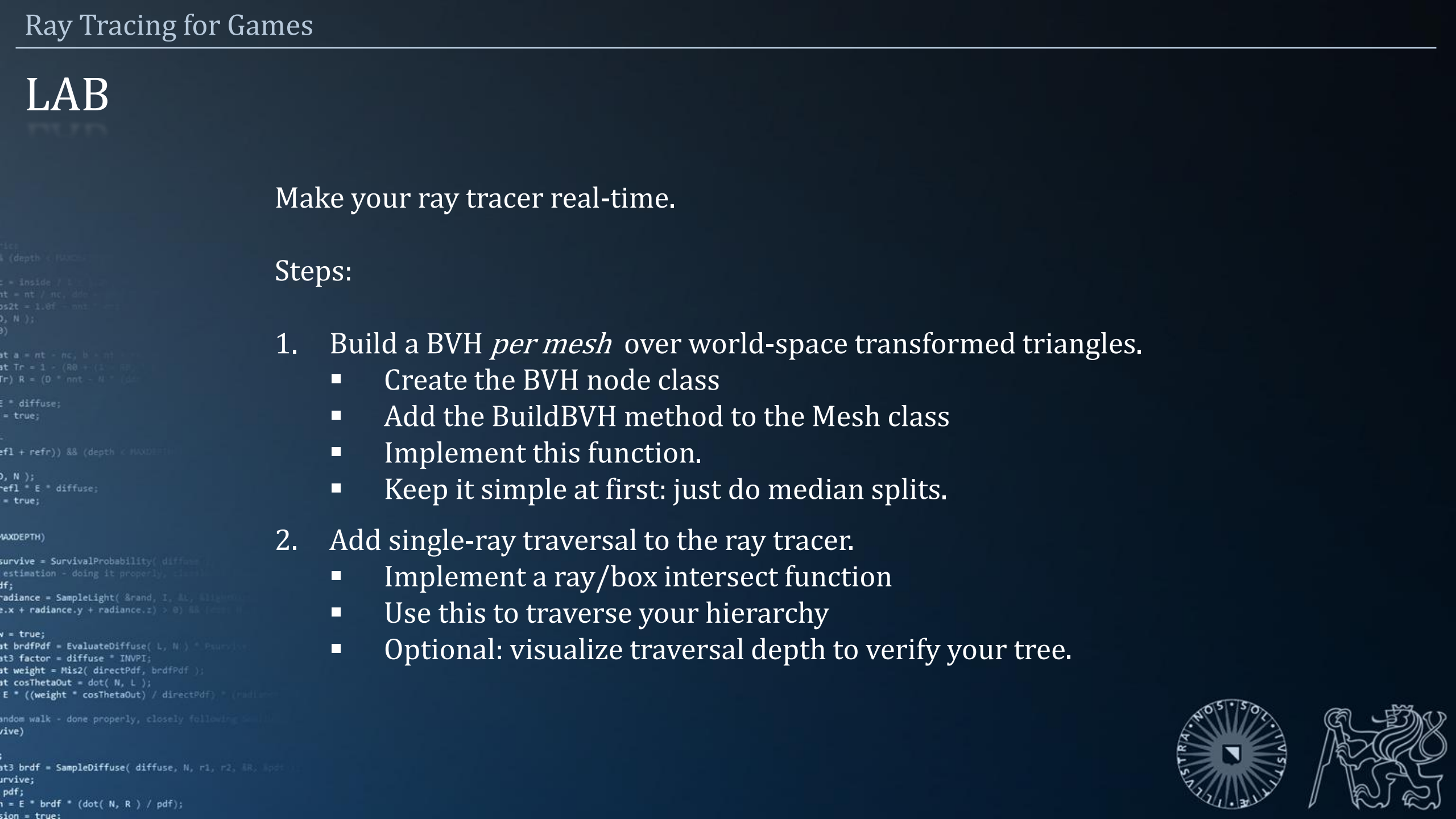
- Introduction
- Acceleration Structures
- Ray Packets
- Binned BVH Construction

LAB

Make your ray tracer real-time.

Steps:

1. Build a BVH *per mesh* over world-space transformed triangles.
   - Create the BVH node class
   - Add the BuildBVH method to the Mesh class
   - Implement this function.
   - Keep it simple at first: just do median splits.
2. Add single-ray traversal to the ray tracer.
   - Implement a ray/box intersect function
   - Use this to traverse your hierarchy
   - Optional: visualize traversal depth to verify your tree.

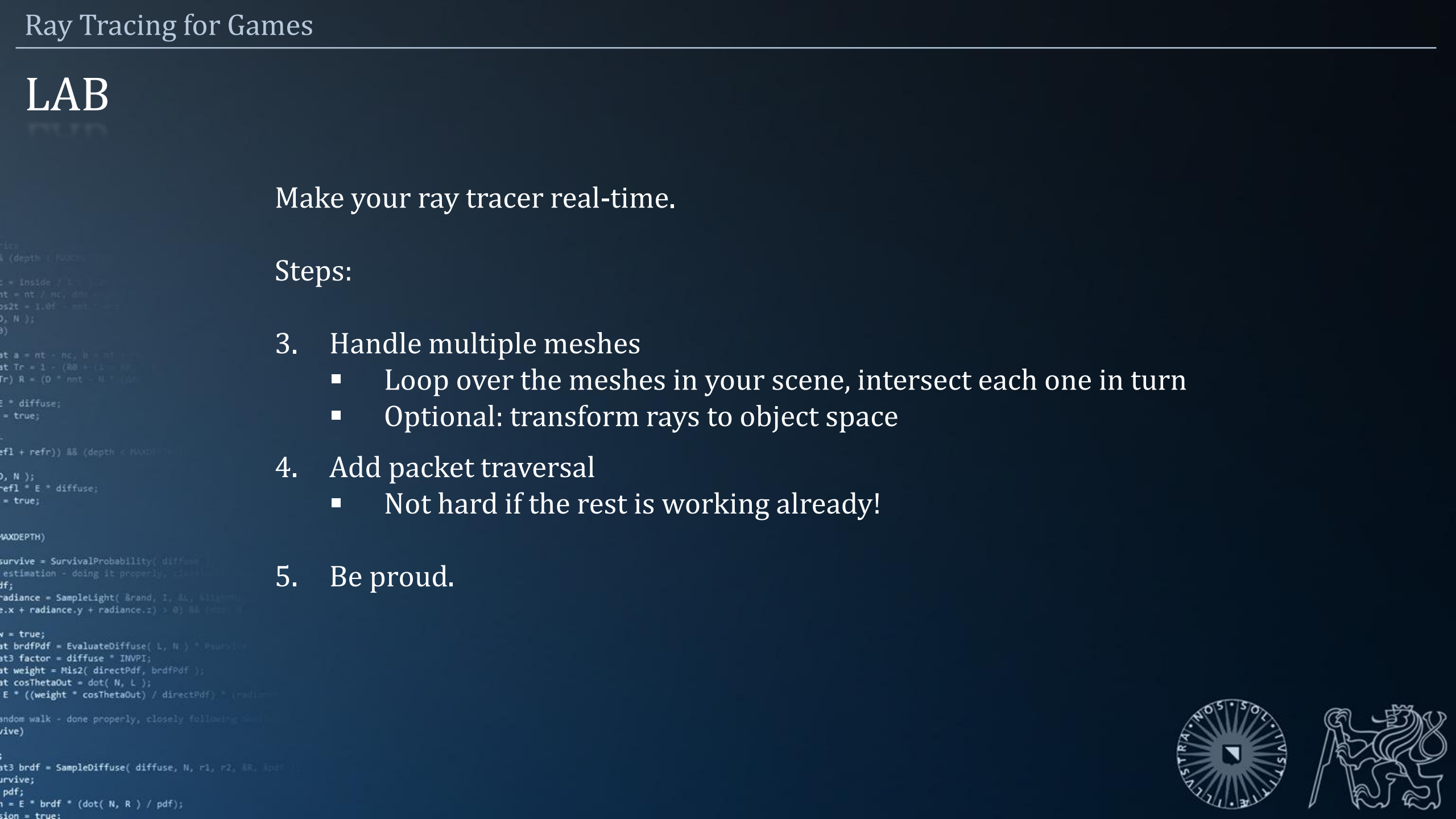


LAB

Make your ray tracer real-time.

Steps:

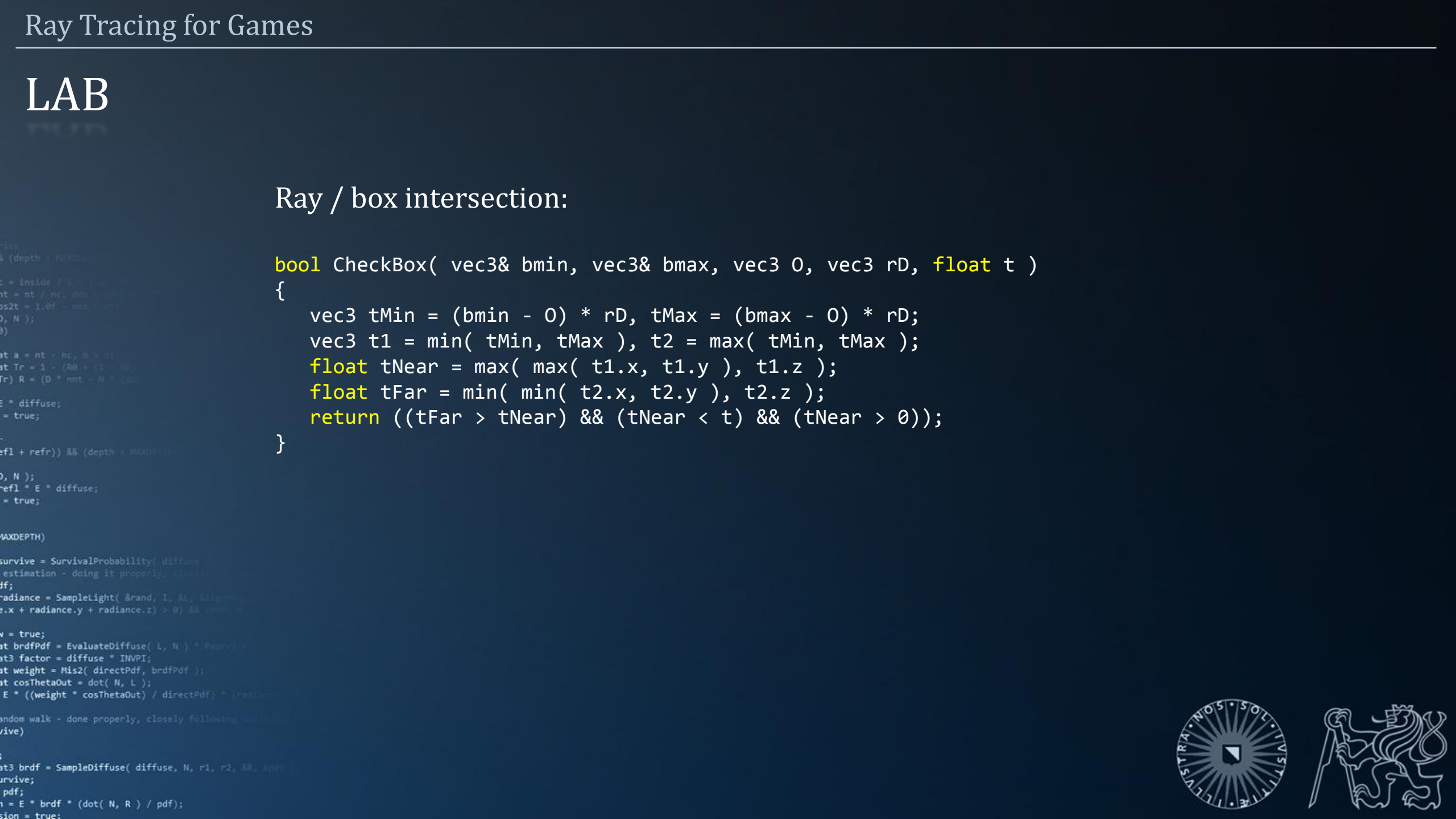
3. Handle multiple meshes
   - Loop over the meshes in your scene, intersect each one in turn
   - Optional: transform rays to object space
4. Add packet traversal
   - Not hard if the rest is working already!
5. Be proud.



LAB

Ray / box intersection:

```
bool CheckBox( vec3& bmin, vec3& bmax, vec3 O, vec3 rD, float t )
{
   vec3 tMin = (bmin - O) * rD, tMax = (bmax - O) * rD;
   vec3 t1 = min( tMin, tMax ), t2 = max( tMin, tMax );
   float tNear = max( max( t1.x, t1.y ), t1.z );
   float tFar = min( min( t2.x, t2.y ), t2.z );
   return ((tFar > tNear) && (tNear < t) && (tNear > 0));
}
```



End of lecture 4.