

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 $N = \text{pixels}$; $M = \text{primitives}$; $L = \text{lights}$

- Primary rays : $O(N M)$
- Secondary rays : $O(N \frac{1}{2} M L)$
- Shading: $O(N)$



Agenda:

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

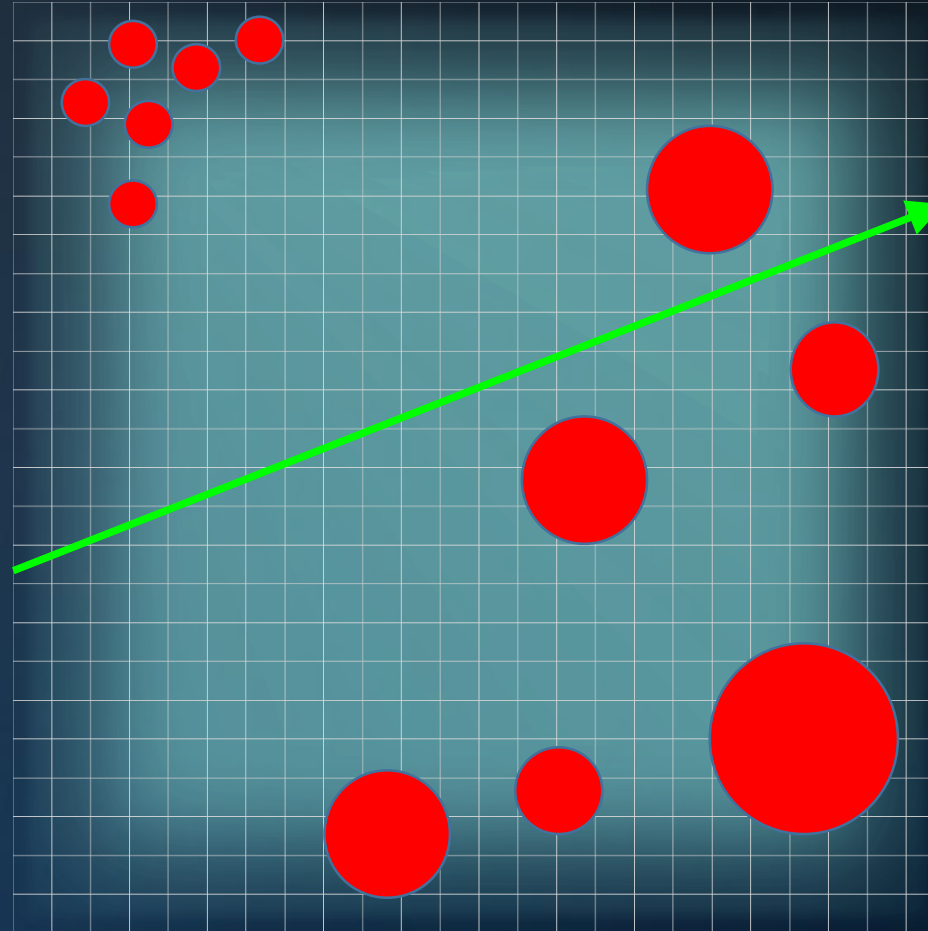


Acceleration Structures

```

    if (depth < MAXDEPTH)
    {
        // Inside / Outside test
        int nt = ncount;
        float cos2t = 1.0f - nnt * nnt;
        float D, N;
        // ...
        float a = nt - nc, b = nt - nc;
        float Tr = 1 - (R0 + (1 - R0) * cos2t);
        float R = (D * nnt - N * (1 - R0));
        // ...
        float E * diffuse;
        // ...
        float refl + refr) && (depth < MAXDEPTH)
        // ...
        float D, N;
        float refl * E * diffuse;
        // ...
        MAXDEPTH)
        // ...
        survive = SurvivalProbability( diffuse );
        // ...
        estimation - doing it properly, closely following walk
        // ...
        radiance = SampleLight( &rand, I, &t, &light );
        // ...
        e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
        // ...
        v = true;
        // ...
        float brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        // ...
        float3 factor = diffuse * INVPI;
        // ...
        float weight = Mis2( directPdf, brdfPdf );
        // ...
        float cosThetaOut = dot( N, L );
        // ...
        E * ((weight * cosThetaOut) / directPdf) * (radiance);
        // ...
        random walk - done properly, closely following walk
        // ...
        survive)
        // ...
        float3 brdf = SampleDiffuse( diffuse, N, r1, r2, &rand, &pdf );
        // ...
        survive;
        // ...
        pdf;
        // ...
        n = E * brdf * (dot( N, R ) / pdf);
        // ...
        // ...
    }

```



Reducing M by stepping through a regular grid

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

Optimal grid resolution:
minimizes

$$C_{\text{step}} * N_{\text{steps}} + C_{\text{intersect}} * N_{\text{intersect}}$$

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

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



Acceleration Structures

```

    if (depth < MAXDEPTH) {
        // Inside the sphere
        t = inside / 1.5;
        nt = nt / nc;
        pos2t = 1.0f - nt;
        D, N );
    }

    // Sample a point on the sphere
    at a = nt - nc; b = nt - nc;
    at Tr = 1 - (RB + (1 - RB) * t);
    Tr) R = (D * nnt - N * (1 - t));

    // Sample a point on the sphere
    E * diffuse;
    = true;

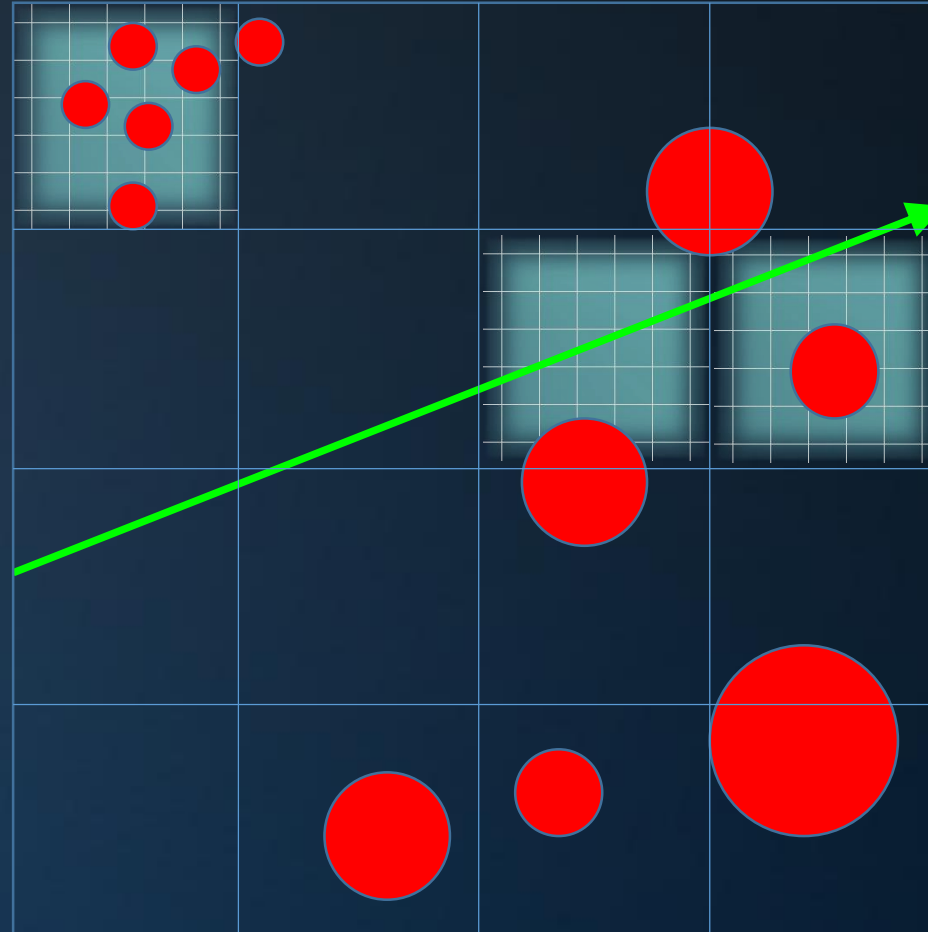
    // Sample a point on the sphere
    refl + refr) && (depth < MAXDEPTH) {
        D, N );
        refl * E * diffuse;
        = true;

    // Sample a point on the sphere
    MAXDEPTH)

    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &t, &light );
    e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH) {
        v = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (radiance);

    // Random walk - done properly, closely following
    survive)

    // Sample a point on the sphere
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &rand, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
  
```



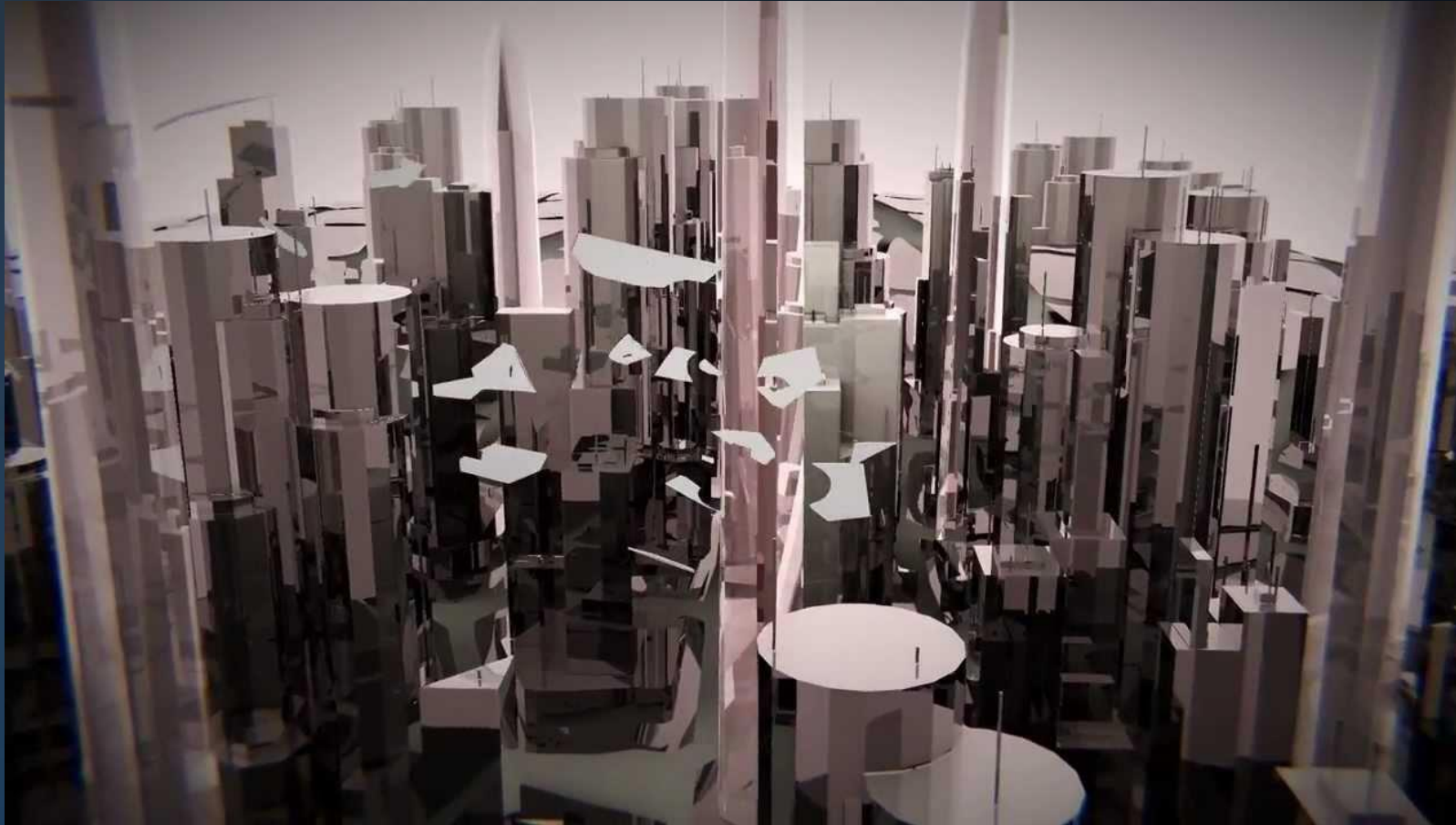
Reducing M by stepping through a nested grid

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

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



Acceleration Structures



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

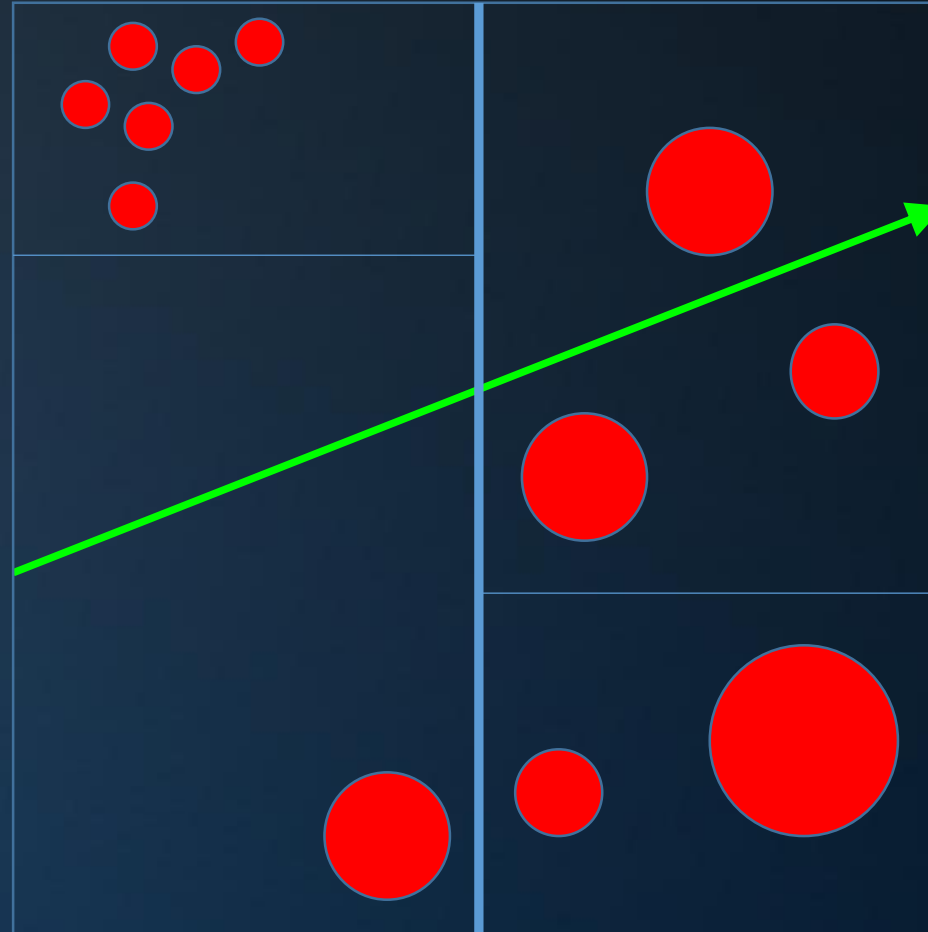


Acceleration Structures

```

    if (depth < MAXDEPTH)
    {
        // Inside / Outside test
        float t = inside / 1.0f;
        float nt = nt / nc;
        float cos2t = 1.0f - nnt;
        float D, N;
        // ...
        float a = nt - nc, b = nt - nc;
        float Tr = 1 - (R0 + (1 - R0) * t);
        float R = (D * nnt - N * (1 - t));
        // ...
        E * diffuse;
        // ...
        refl + refr)) && (depth < MAXDEPTH)
        // ...
        D, N;
        refl * E * diffuse;
        // ...
        MAXDEPTH)
        // ...
        survive = SurvivalProbability( diffuse );
        // ...
        estimation - doing it properly, closely following
        // ...
        radiance = SampleLight( &rand, I, &L, &light );
        // ...
        e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
        // ...
        v = true;
        // ...
        brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        // ...
        factor = diffuse * INVPI;
        // ...
        weight = Mis2( directPdf, brdfPdf );
        // ...
        cosThetaOut = dot( N, L );
        // ...
        E * ((weight * cosThetaOut) / directPdf) * (radiance);
        // ...
        random walk - done properly, closely following
        // ...
        survive)
        // ...
        brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        // ...
        survive;
        // ...
        pdf;
        // ...
        n = E * brdf * (dot( N, R ) / pdf);
        // ...
        // ...
    }

```



Reducing M by traversing a binary tree

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



```
graph TD; root --> left; root --> right; left --> top1[top]; left --> bottom1[bottom]; right --> top2[top]; right --> bottom2[bottom]
```

Let $N = \text{pixels}$; $M = \text{primitives}$; $L = \text{lights}$



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

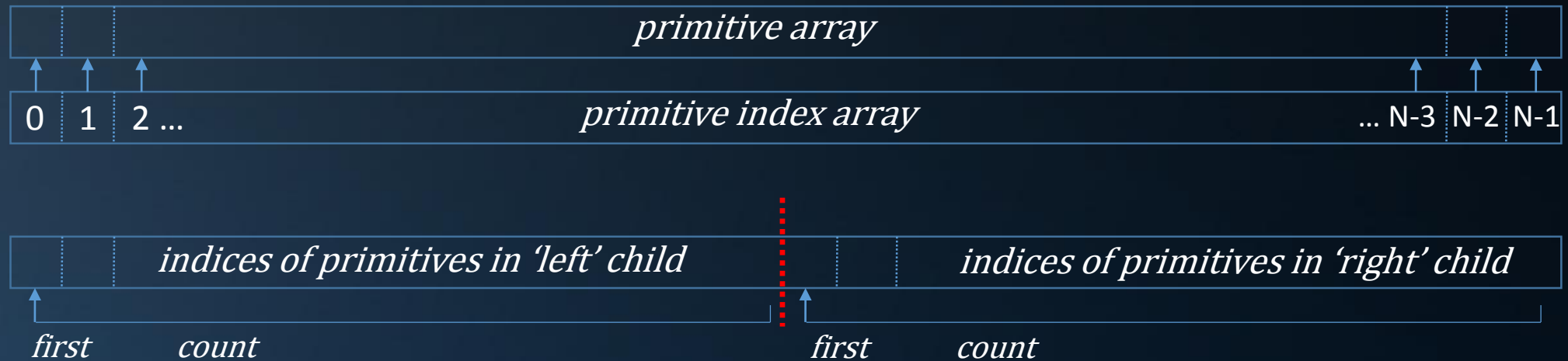
```
void ConstructBVH( Primitive* primitives )
{
    BVHNode* root = new BVHNode();
    root->primitives = primitives;
    root->bounds = CalculateBounds( primitives );
    root->isLeaf = true;
    root->Subdivide();
}

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
};
```



Acceleration Structures

Bounding Volume Hierarchy: data structure

```

    if (depth < MAXDEPTH)
    {
        // Inside / Outside test
        float nt = nt / nc;
        float nnt = nt * nt;
        float D = D * nnt;
        float N = N * nnt;
        float a = nt - nc;
        float b = nt - nc;
        float Tr = 1 - (R0 + (1 - R0) * a);
        float R = (D * nnt - N * (1 - R0));
        float E = diffuse;
        float refl = true;
        float refl + refr;
        float depth < MAXDEPTH;
        float D, N;
        float refl * E * diffuse;
        float refl = true;
        float MAXDEPTH;
        float survive = SurvivalProbability( diffuse );
        float estimation - doing it properly, closely following walk;
        float if;
        float radiance = SampleLight( &rand, I, &t, &light );
        float e.x + radiance.y + radiance.z > 0;
        float v = true;
        float brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        float factor = diffuse * INVPI;
        float weight = Mis2( directPdf, brdfPdf );
        float cosThetaOut = dot( N, L );
        float E * ((weight * cosThetaOut) / directPdf) * (radiance);
        float random walk - done properly, closely following walk;
        float survive;
        float brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        float survive;
        float pdf;
        float n = E * brdf * (dot( N, R ) / pdf);
        float estimation = true;

```

```

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

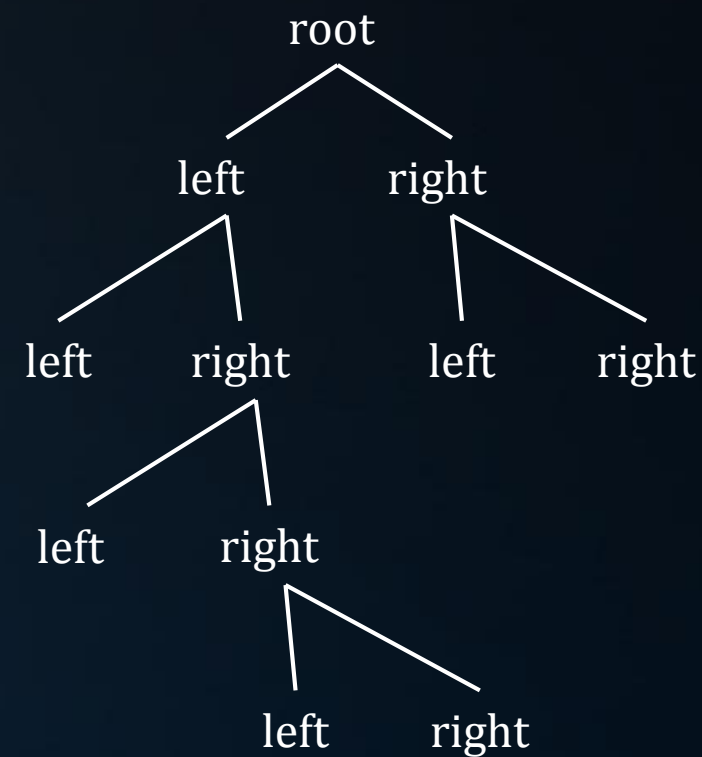
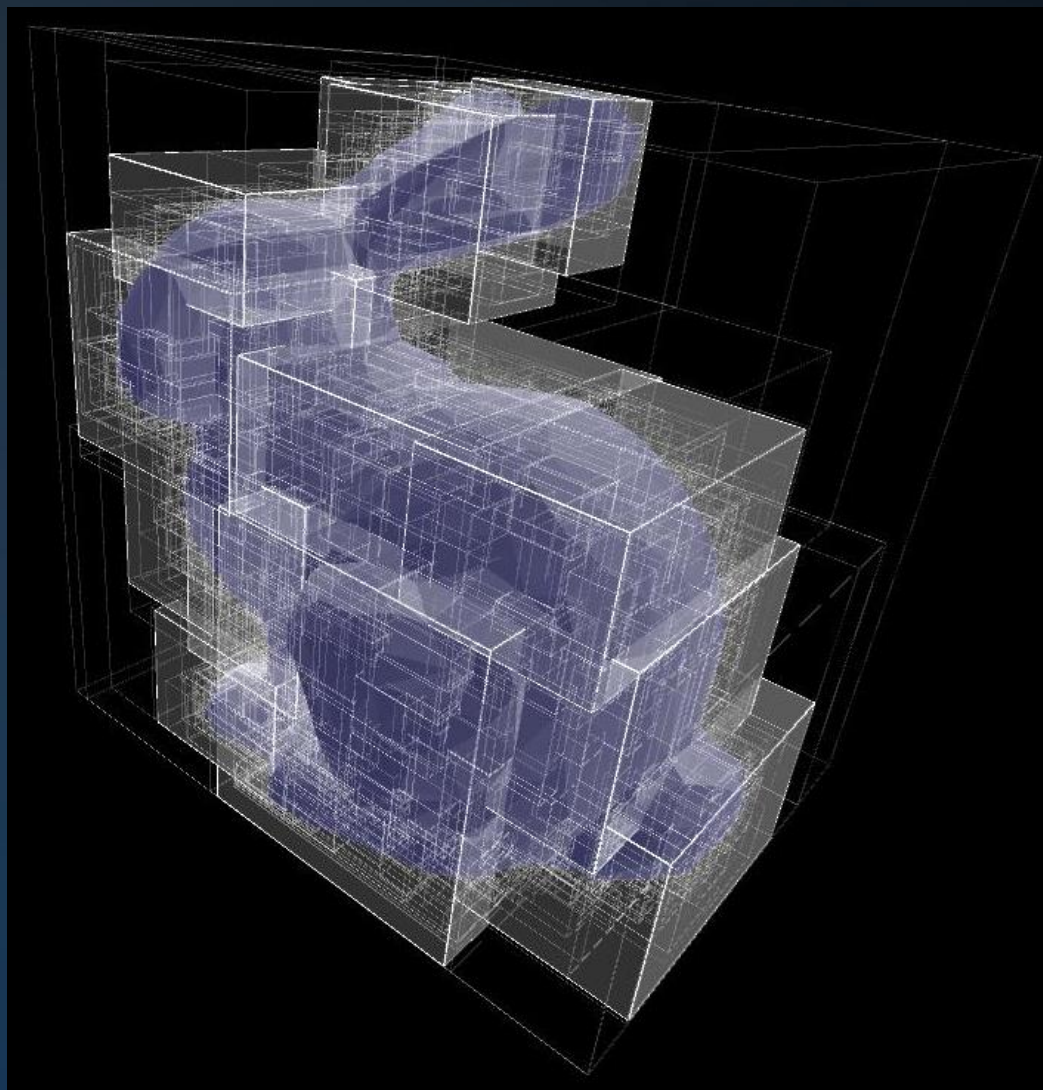
```
void ConstructBVH( Primitive* primitives )
{
    // create index array
    uint* indices = new uint[N];
    for( int i = 0; i < N; i++ ) indices[i] = i;
    // allocate BVH node pool
    BVHNode* pool = (BVHNode*)_aligned_malloc( sizeof( BVHNode ) * (N * 2), 64 );
    BVHNode* root = &pool[0];
    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

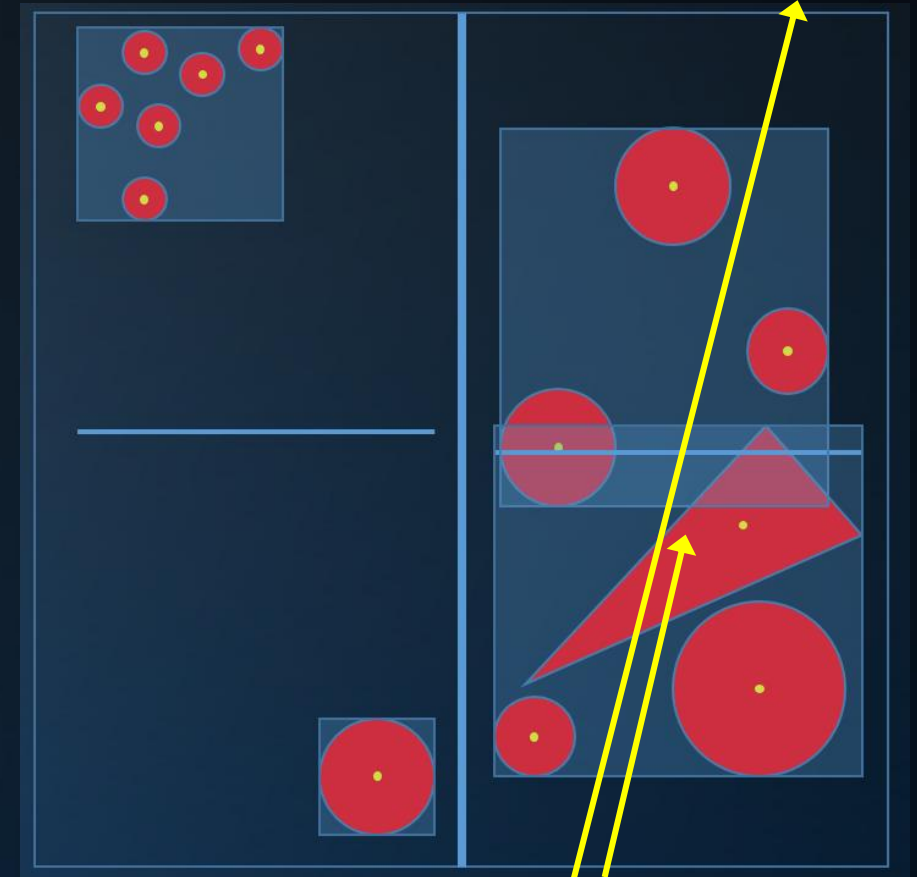
Bounding Volume Hierarchy: traversal

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

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




```
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

```
...ics
& (depth < MAXD);

...t = inside / 1.5f;
nt = nt / nc; dde = dde / nc;
os2t = 1.0f - nnt * dde;
D, N );
0);

...t a = nt - nc; b = nt - nc;
at Tr = 1 - (R0 + (1 - R0) * a);
Tr) R = (D * nnt - N * (dde

...E * diffuse;
= true;

...efl + refr)) && (depth < MAXDEPTH)

D, N );
refl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following walk (survive)
if;
radiance = SampleLight( &rand, I, &t, &light );
e.x + radiance.y + radiance.z) > 0) && (e.x + radiance.y + radiance.z) > 0)

v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
t3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

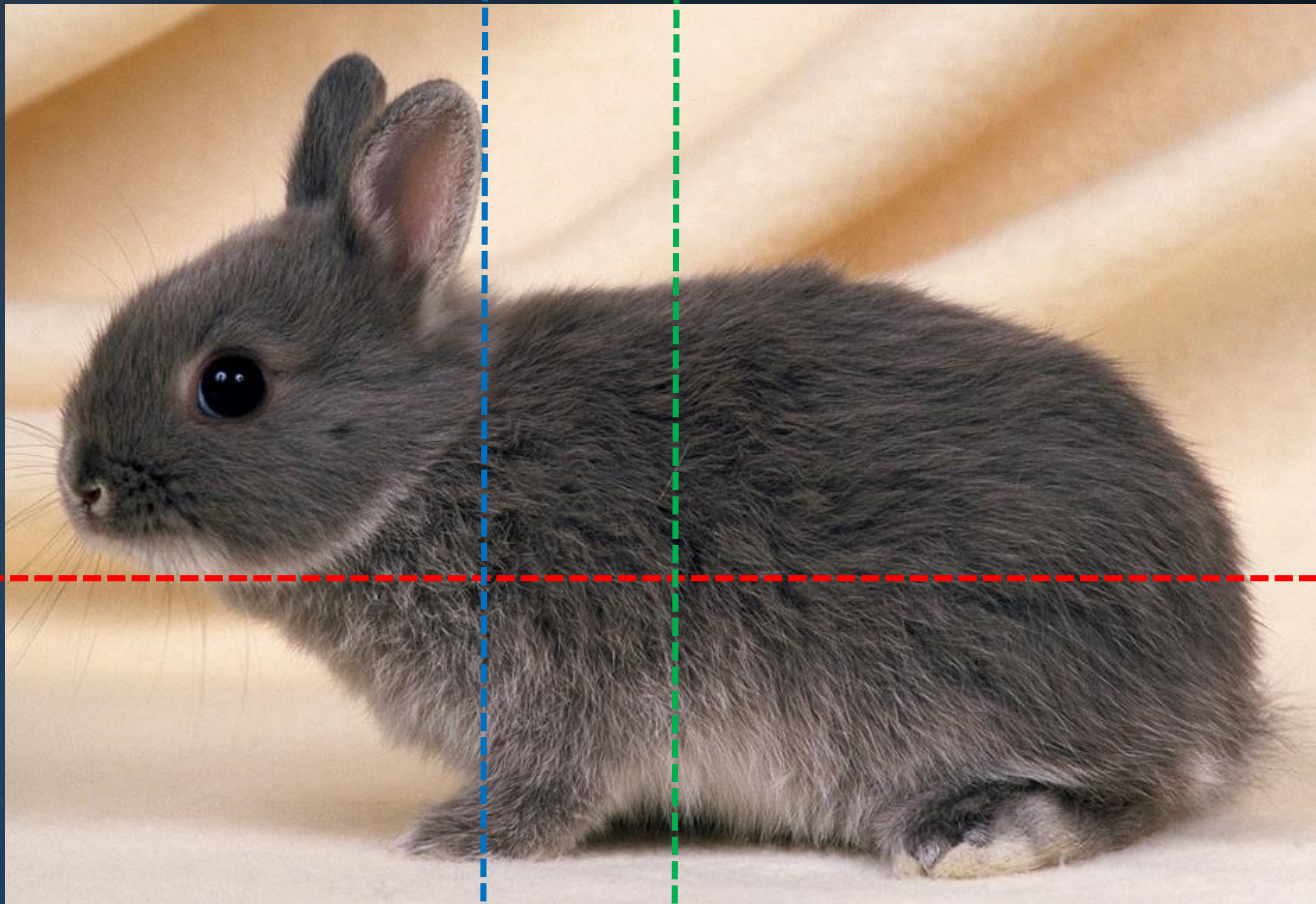
random walk - done properly, closely following walk (survive)
survive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

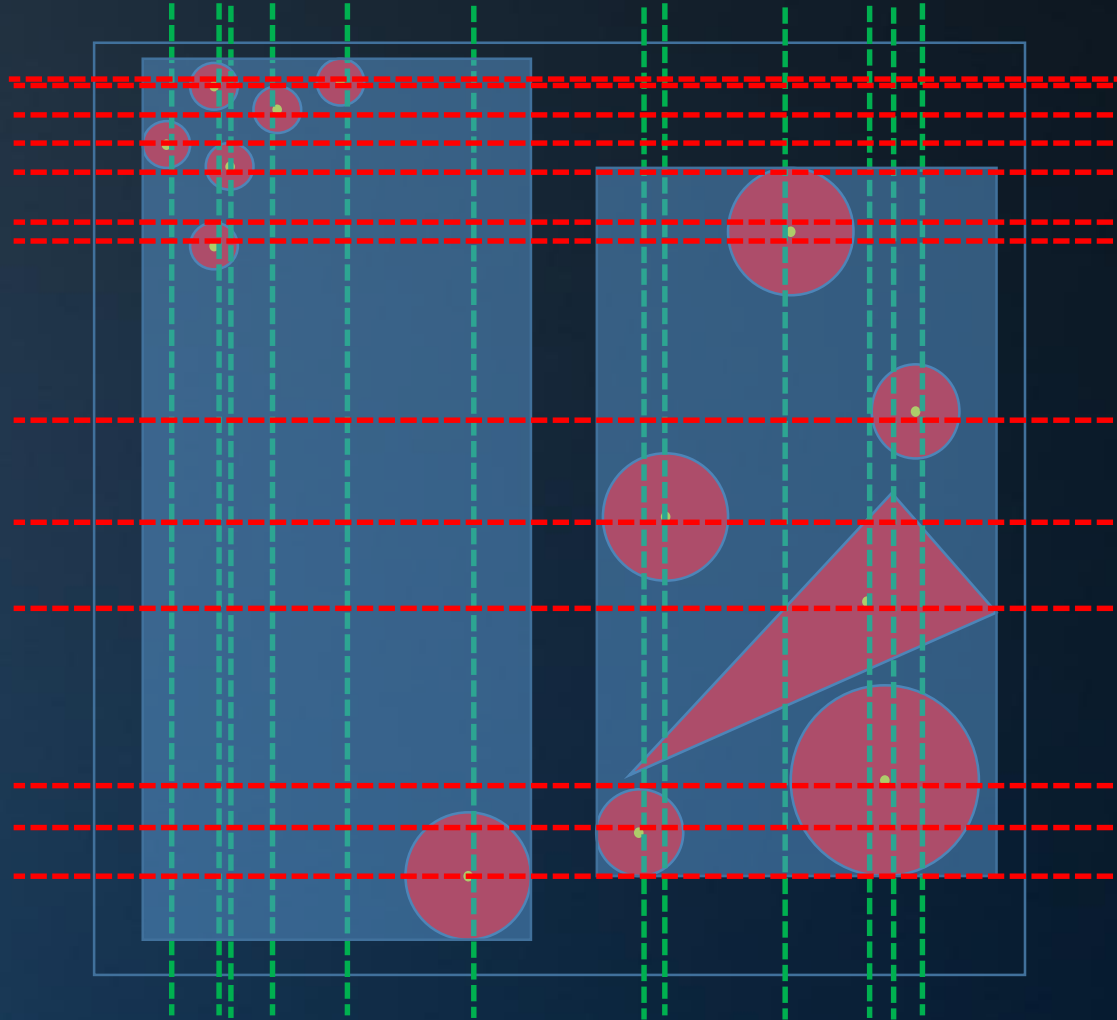


Binned BVH Construction

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



Binned BVH Construction



Cost:

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

Select the split with the lowest cost.



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

It will also lead to much longer BVH construction times.

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

It will also lead to much longer BVH construction times.

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

It will also lead to much longer BVH construction times.

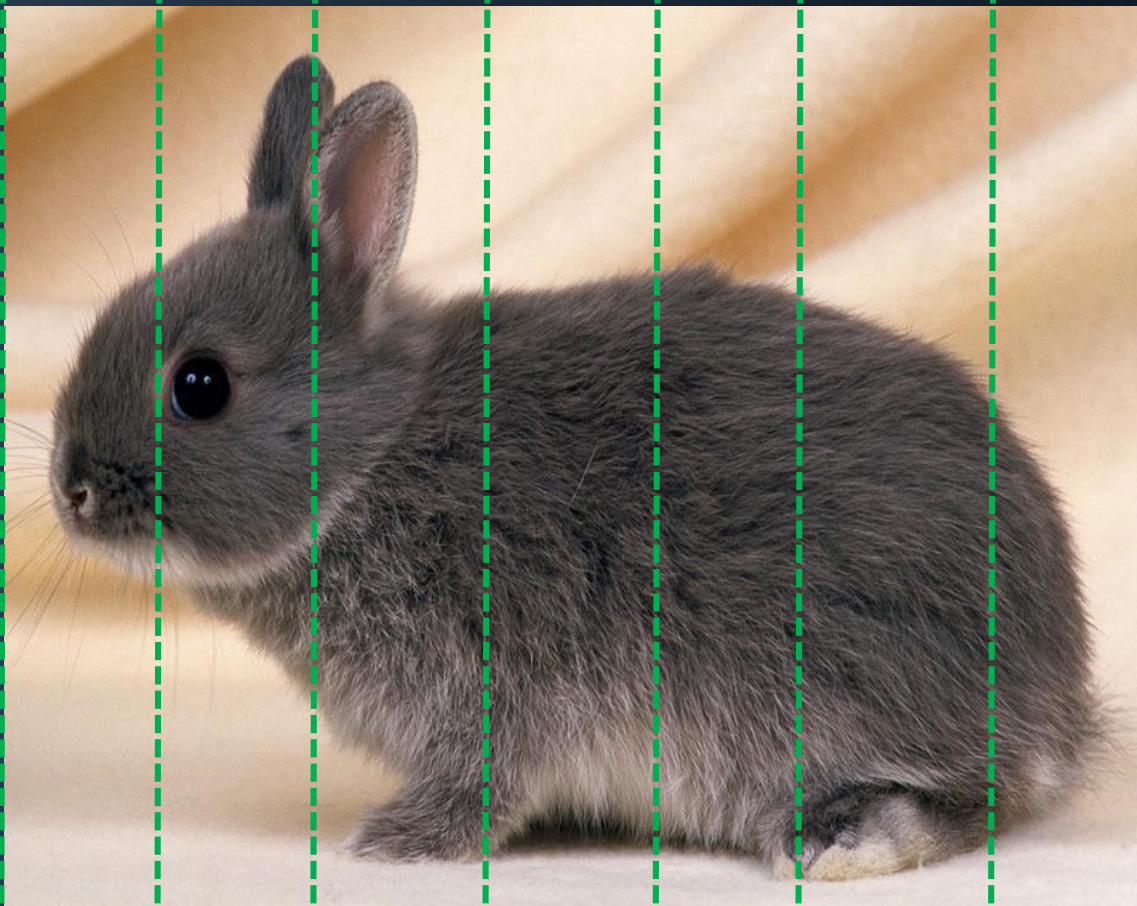


We need to go faster.



Binned BVH Construction

```
...
    & (depth < MAXDEPTH)
    {
        t = inside / L.N;
        nt = nt / nc;
        cos2t = 1.0f - nt;
        D, N );
    }
    {
        at a = nt - nc, b = nt - nc;
        at Tr = 1 - (RB + (1 - RB) * a);
        Tr) R = (D * nt - N * (a *
    }
    {
        E * diffuse;
        = true;
    }
    {
        refl + refr)) && (depth < MAXDEPTH)
        {
            D, N );
            refl * E * diffuse;
            = true;
        }
    }
    {
        MAXDEPTH)
    }
    {
        survive = SurvivalProbability( diffuse );
        estimation - doing it properly, closely following
        if;
        radiance = SampleLight( &rand, I, &t, &light );
        e.x + radiance.y + radiance.z) > 0) && (depth <
    }
    {
        w = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
        at3 factor = diffuse * INVPI;
        at weight = Mis2( directPdf, brdfPdf );
        at cosThetaOut = dot( N, L );
        E * ((weight * cosThetaOut) / directPdf) * (radiance
    }
    {
        random walk - done properly, closely following
        (survive)
    }
    {
        at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &P
    }
    {
        survive;
        pdf;
        n = E * brdf * (dot( N, R ) / pdf);
        sion = true;
    }
}
```



Binned SAH:

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



Using binned SAH building yields a high quality BVH in little time:

```

    (depth < MAXDEPTH)
    {
        z = inside / 1.5 * 1.5;
        nt = nt / nc; ddx = dot(N, N);
        cos2t = 1.0f - nnt * nnt;
        t = sqrt(cos2t);
        D = N * t;
    }

    at a = nt - nc; b = nt + nc;
    Tr = 1 - (R0 + (1 - R0) * a);
    Tr) R = (D * nnt - N * (ddx *
    E * diffuse;
    = true;

    refl + refr)) && (depth < MAXDEPTH)
    {
        D, N);
        refl * E * diffuse;
        = true;

    MAXDEPTH)

    survive = SurvivalProbability( diffuse
    estimation - doing it properly,
    if;
    radiance = SampleLight( &rand, 1, &
    .x + radiance.y + radiance.z) > 0)

    w = true;
    at brdfPdf = EvaluateDiffuse( L, N,
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPo
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / dire

    random walk - done properly, closely
    (ive)

    ;
    at3 brdf = SampleDiffuse( diffuse,
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



Agenda:

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



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.



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.

```

    // hit
    if (depth < MAXDEPTH)
    {
        // inside / outside
        nt = nt / nc; addo = addo * nc;
        cos2t = 1.0f - nnt; // nnt = nnt * nnt
        D, N );
    }
}

// hit
// at a = nt - nc, b = nt * nc;
// at Tr = 1 - (R0 + (1 - R0) * a);
// Tr) R = (D * nnt - N * (addo
//
// E * diffuse;
// = true;
//
//
// refl + refr)) && (depth < MAXDEPTH)
//
// D, N );
// refl * E * diffuse;
// = true;
//
// MAXDEPTH)
//
// survive = SurvivalProbability( diffuse,
// estimation - doing it properly, closely following
// if;
// radiance = SampleLight( &rand, I, &t, &light,
// e.x + radiance.y + radiance.z) > 0) && (cosThetaOut
//
// v = true;
// at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
// at3 factor = diffuse * INVPI;
// at weight = Mis2( directPdf, brdfPdf );
// at cosThetaOut = dot( N, L );
// E * ((weight * cosThetaOut) / directPdf) * (radiance
//
// random walk - done properly, closely following well-known
// survive)
//
//
// at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
// survive;
// pdf;
// n = E * brdf * (dot( N, R ) / pdf);
// ion = true;

```



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.

