Ray Tracing for Games

Dr. Jacco Bikker - IGAD/BUAS, Breda, February 3

Welcome!

5

Ray Tracing for Games

Wednesday
13:00 – 17:00

course intro
LH2
template
Whitted
refactoring
RT-centric games

LAB 1

Thursday
09:00 – 14:00

advanced Whitted
audio, AI & physics
faster Whitted
Heaven7

LAB 2

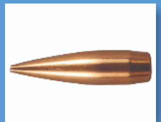
work @ home

End result day 2:

A solid Whitted-style ray tracer, as a basis for subsequent work.

Friday
09:00 – 17:00

optimization
profiling, rules of engagement
threading



LAB 3

SIMD
applied SIMD
SIMD triangle
SIMD AABB

LAB 4

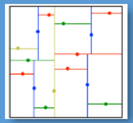
End result day 3:

A 5x faster tracer.

GLOBAL GAME JAM

Monday
09:00 – 17:00

acceleration
grid, BVH, kD-tree
SAH
binning



LAB 5

refitting
top-level BVH
threaded building

LAB 6

End result day 4:

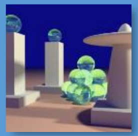
A real-time tracer.

Tuesday
09:00 – 17:00

Monte-Carlo
Cook-style
glossy, AA
area lights, DOF

LAB 7

path tracing



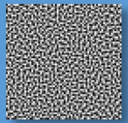
LAB 8

End result day 5:

Cook or Kajiya.

Thursday
09:00 – 17:00

random numbers
stratification
blue noise



LAB 9

importance sampling
next event estimation

LAB 10

End result day 6:

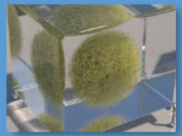
Efficiency.

Friday
09:00 – 17:00

future work

LAB 11

path guiding



LAB 10

End result day 6:

Great product.

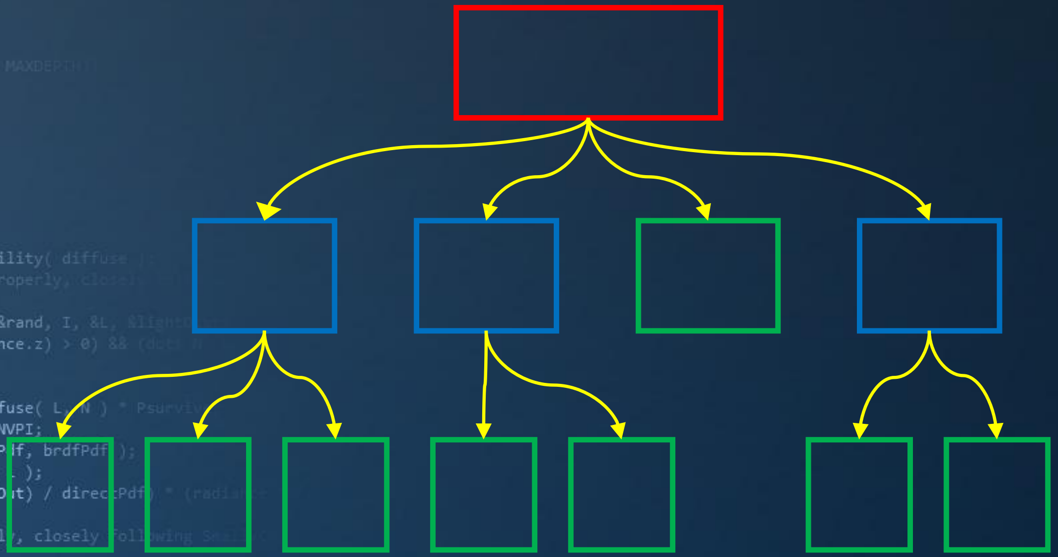
Agenda:

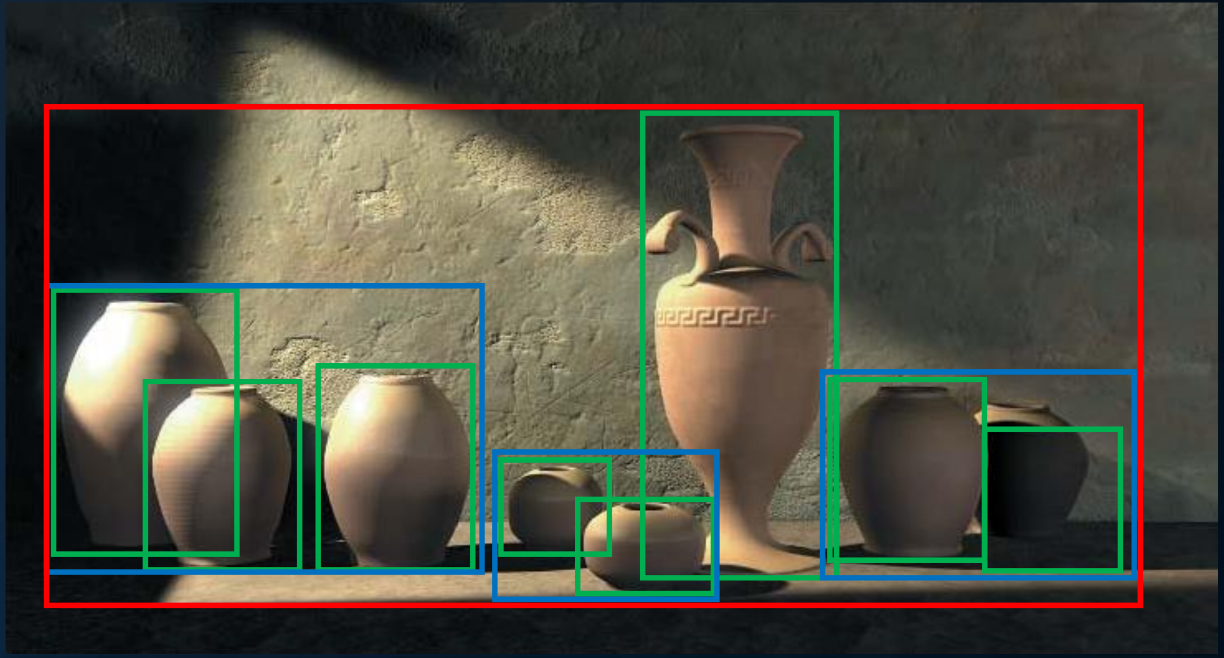
- Accelerate
- BVH
- Surface Area Heuristic
- Binning
- Fast Traversal

The Bounding Volume Hierarchy

BVH: tree structure, with:

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

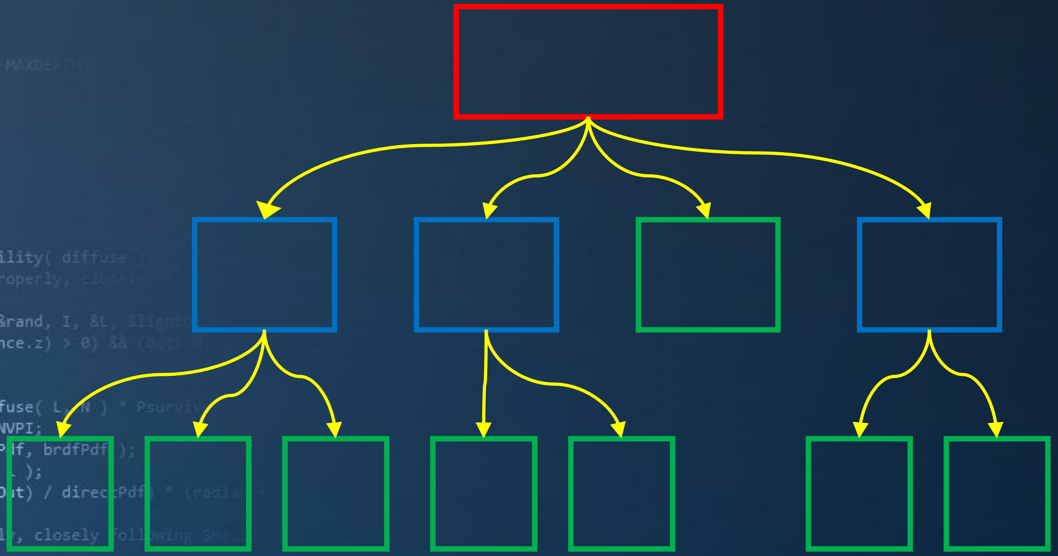




Automatic Construction of Bounding Volume Hierarchies

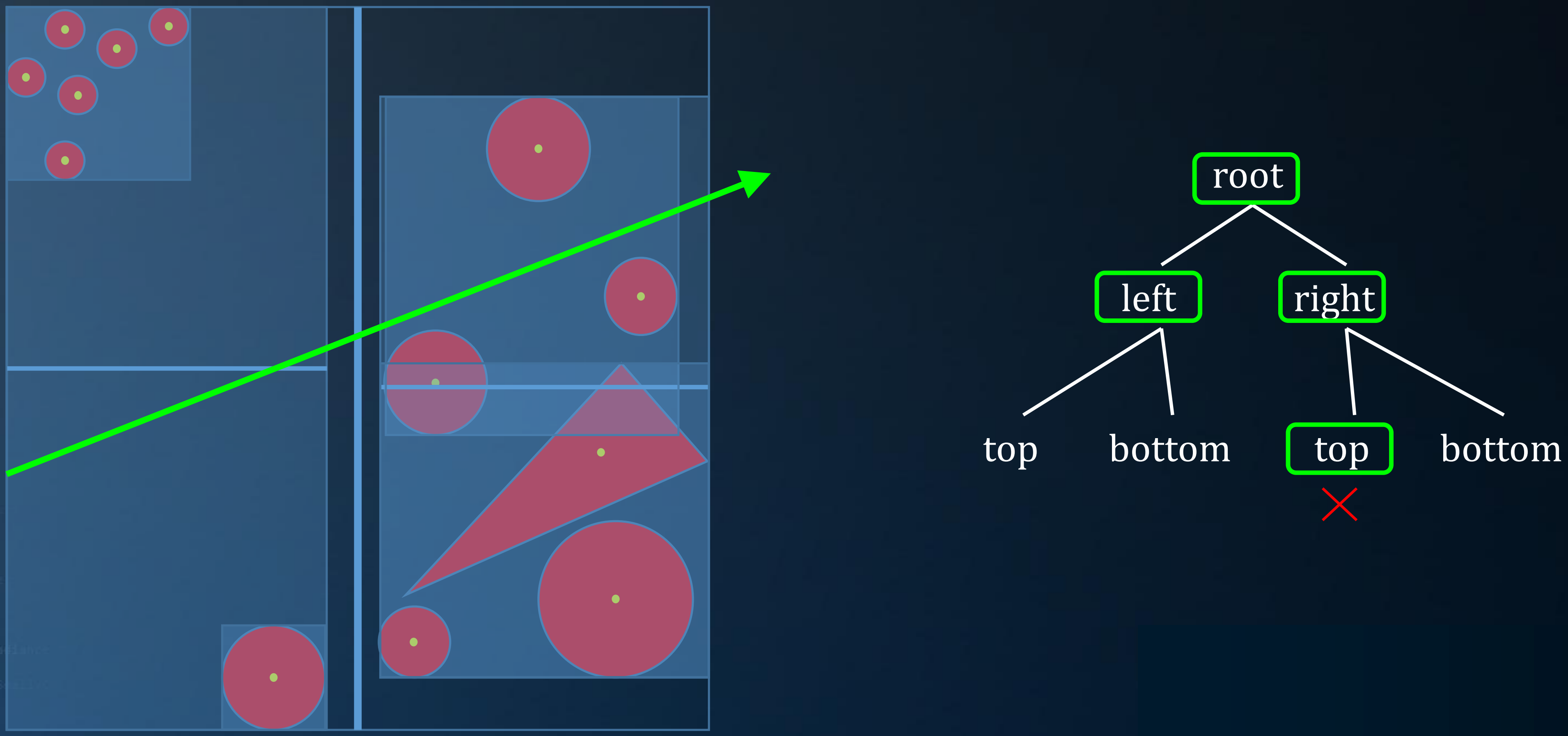
BVH: tree structure, with:

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



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

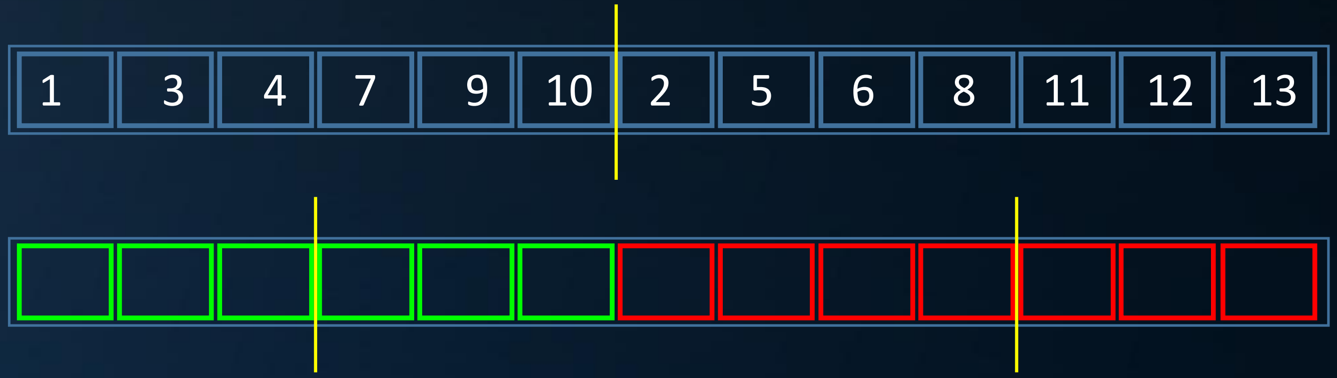
Automatic Construction of Bounding Volume Hierarchies



Automatic Construction of Bounding Volume Hierarchies

1	2	3	4	5	6	7	8	9	10	11	12	13

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

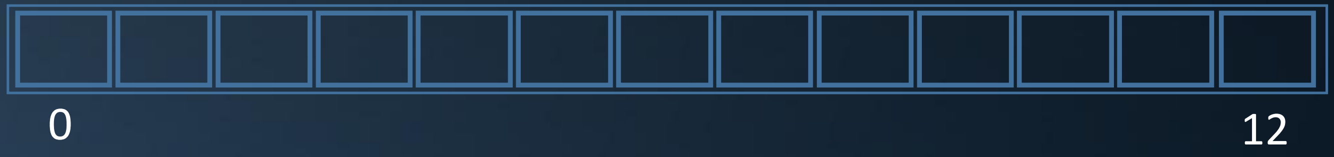


Note:

Step 3 can be done ‘in place’.

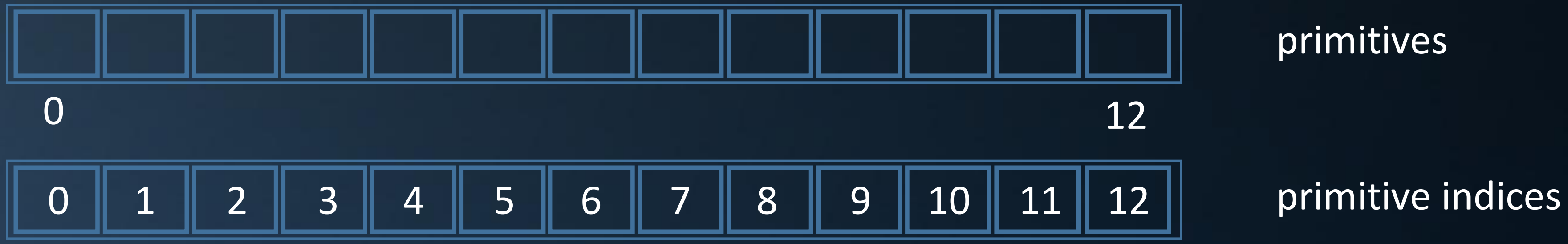
This process is identical to QuickSort: the split plane is The ‘pivot’.

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
};
```

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
};
```

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();
}
```

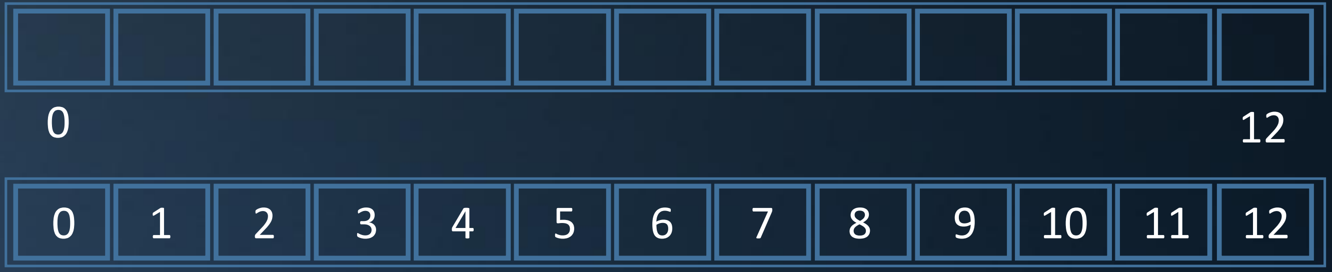
```
void BVHNode::Subdivide()
{
   if (count < 3) return;
   this.left = new BVHNode();
   this.right = new BVHNode();
   Partition();
   this.left->Subdivide();
   this.right->Subdivide();
   this.isLeaf = false;
}
```

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];
   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;
}
```

Automatic Construction of Bounding Volume Hierarchies



primitives

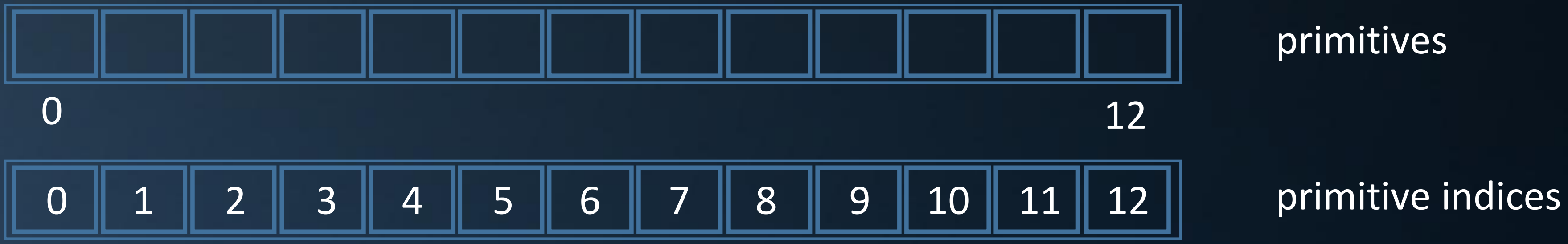
0 1 2 3 4 5 6 7 8 9 10 11 12

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
};
```

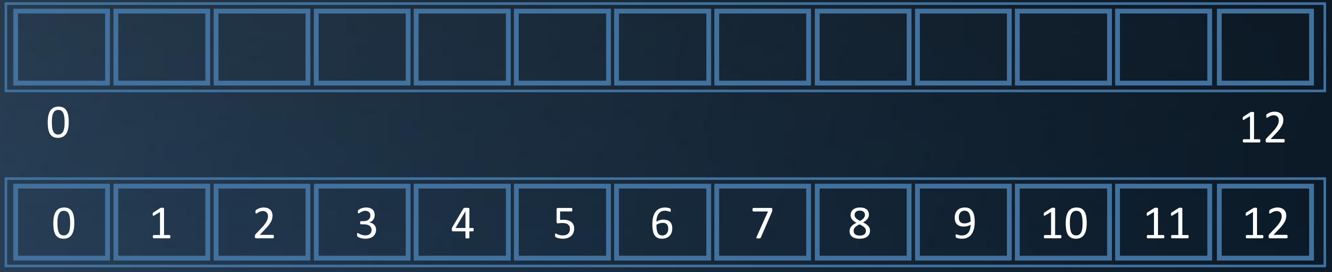
BVH nodes

Automatic Construction of Bounding Volume Hierarchies



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

Automatic Construction of Bounding Volume Hierarchies



primitives

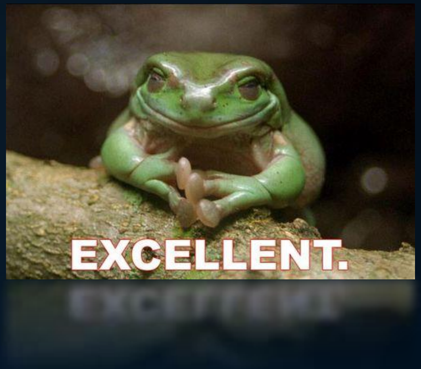
0

12

0	1	2	3	4	5	6	7	8	9	10	11	12

primitive indices

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



BVH nodes

TL;DR

Building a BVH:

- We build a BVH over *triangle indices*.
- The optimal BVH node structure is 32 bytes in size.
- Node ‘pointers’ are actually indices in a pre-allocated, aligned array.
- Node 1 is unused.

```
struct BVHNode
{
    float3 bmin;
    int leftFirst;
    float3 bmax;
    int count;
};
```

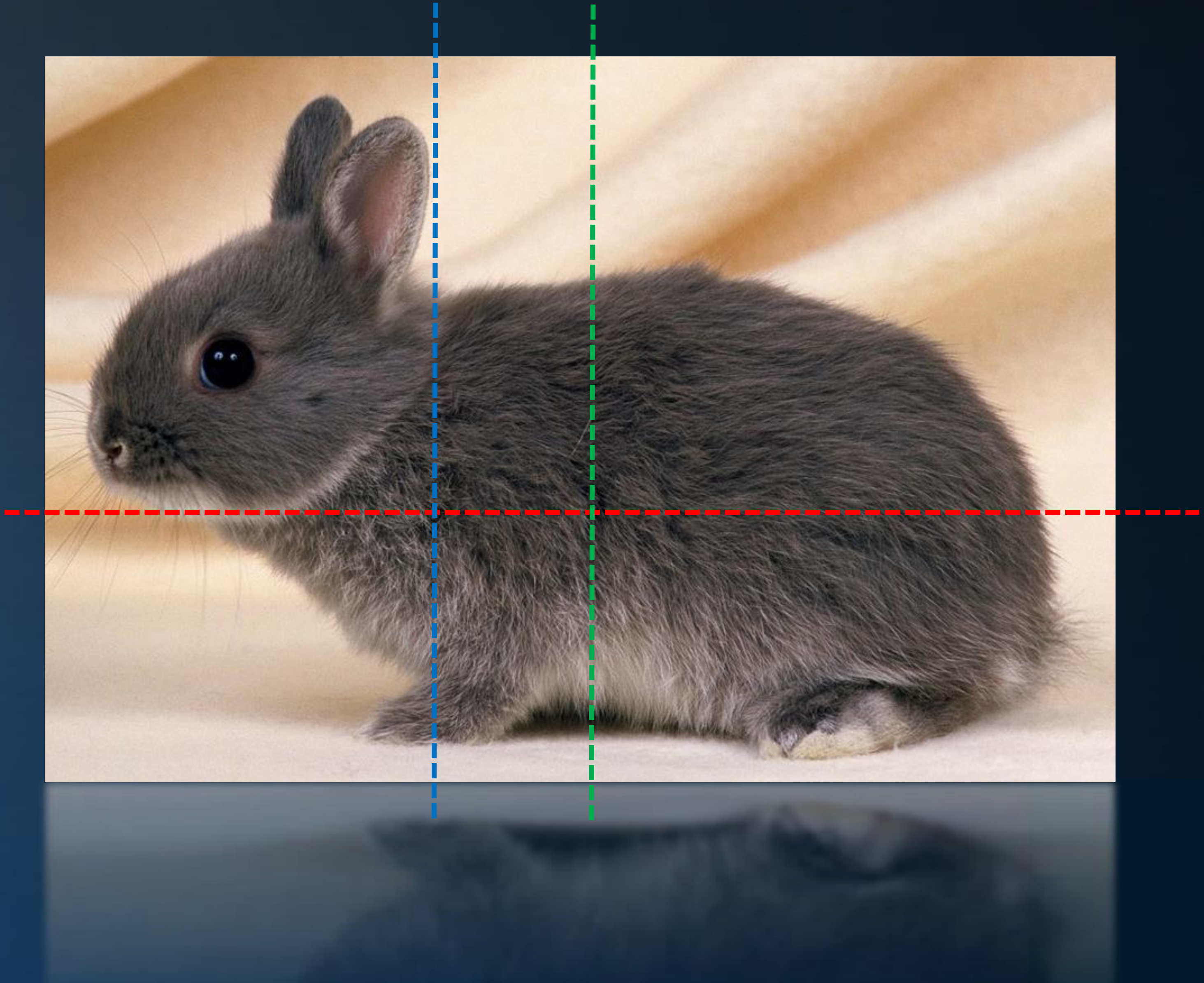
MILITARY POLICE
57 XB 57
ALL BRITISH MILITARY VISITORS TO EAST BERLIN REPORT TO SIDE WINDOW

Agenda:

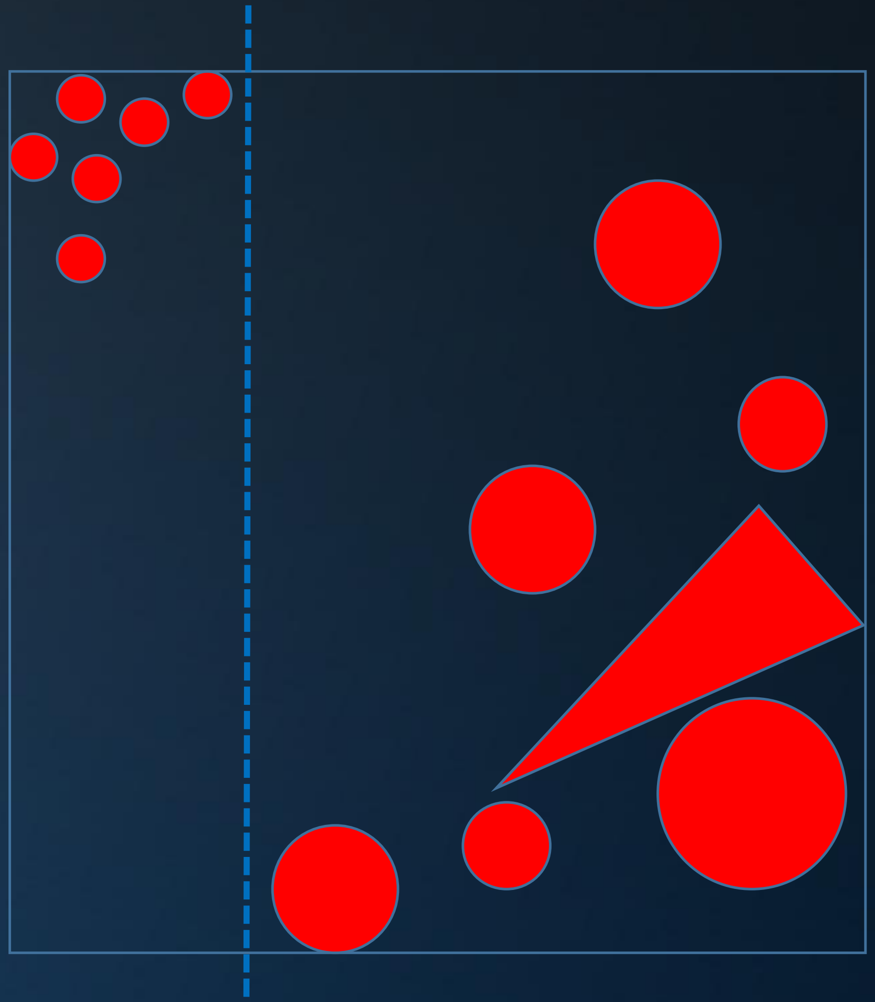
- Accelerate
- BVH
- Surface Area Heuristic
- Binning
- Fast Traversal

Ray Tracing for Games

```
rics
& (depth < MAXDEPTH)

c = inside ? 1 : 1.2
nt = nt / nc, ddn = d
os2t = 1.0f - nnt * nn
D, N );
0)

at a = nt - nc, b = nt + n
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
= true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely
df;
radiance = SampleLight( &rand, I, &L, &lightD
e.x + radiance.y + radiance.z) > 0) && (dot( N

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

andom walk - done properly, closely following Smallv
vive)

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



Ray Tracing for Games



```
rics
& (depth < MAXDEPTH)

c = inside ? 1 : 1.2
nt = nt / nc, ddn = 
os2t = 1.0f - nnt * nnt
D, N );
0)

at a = nt - nc, b = nt + nc
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
= true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely
df;
radiance = SampleLight( &rand, I, &L, &lightD
e.x + radiance.y + radiance.z) > 0) && (dot( N

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

andom walk - done properly, closely following Smal
vive)

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

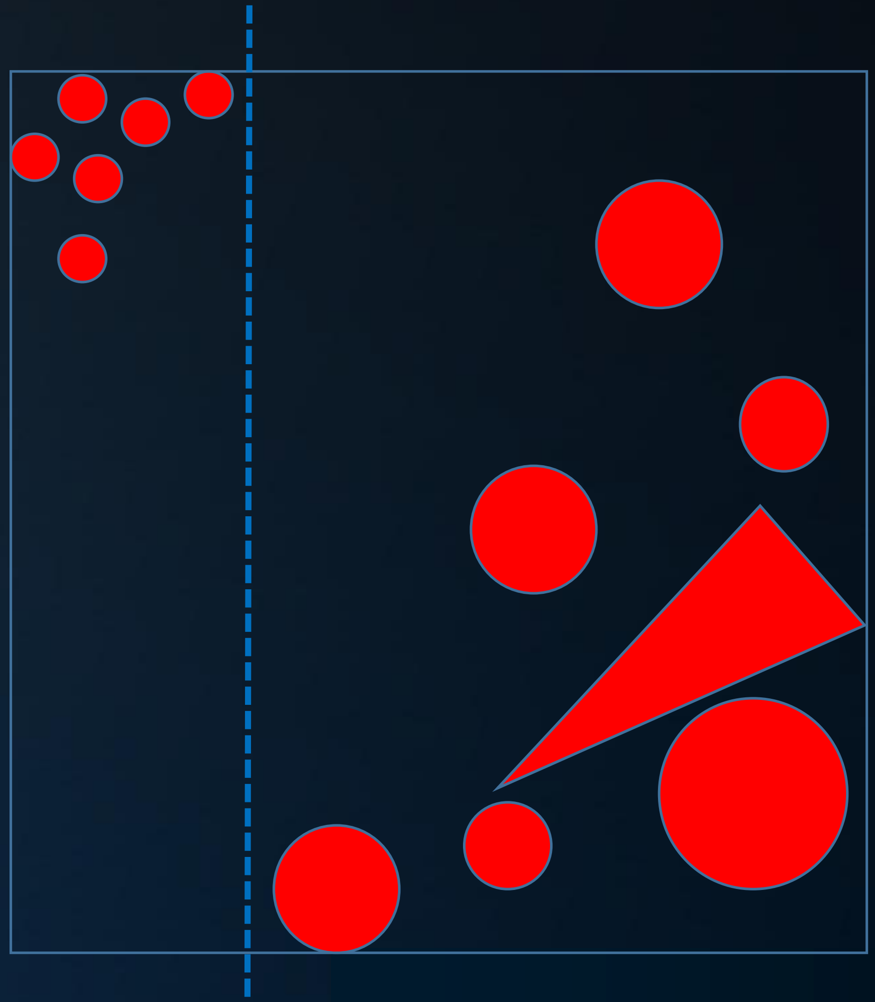
What Are We Trying To Solve?

A BVH is used to reduce the number of ray/primitive intersections.

But: it introduces new intersections.

The ideal BVH minimizes:

- # of ray / primitive intersections
- # of ray / node intersections.



Optimal Split Plane Position

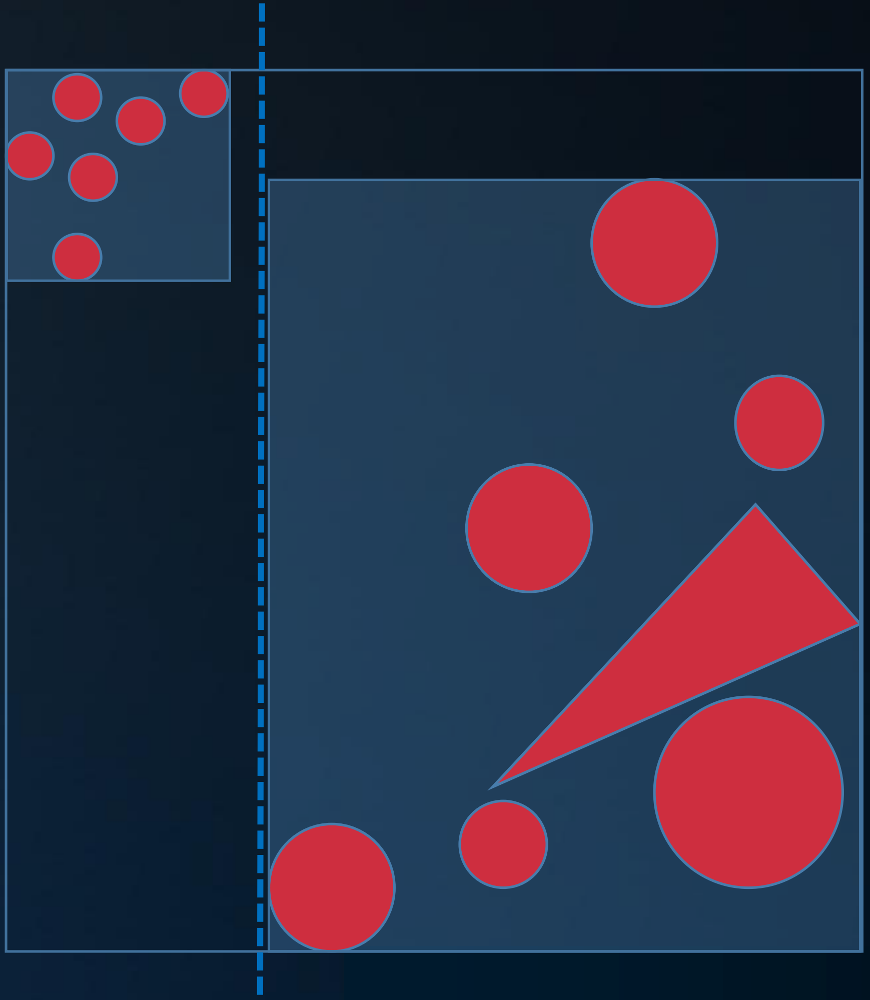
The ideal split minimizes the *expected cost* of a ray intersecting the resulting nodes.

This expected cost is based on:

- Number of primitives that will have to be intersected
- Probability of this happening

The cost of a split is thus:

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



Optimal Split Plane Position

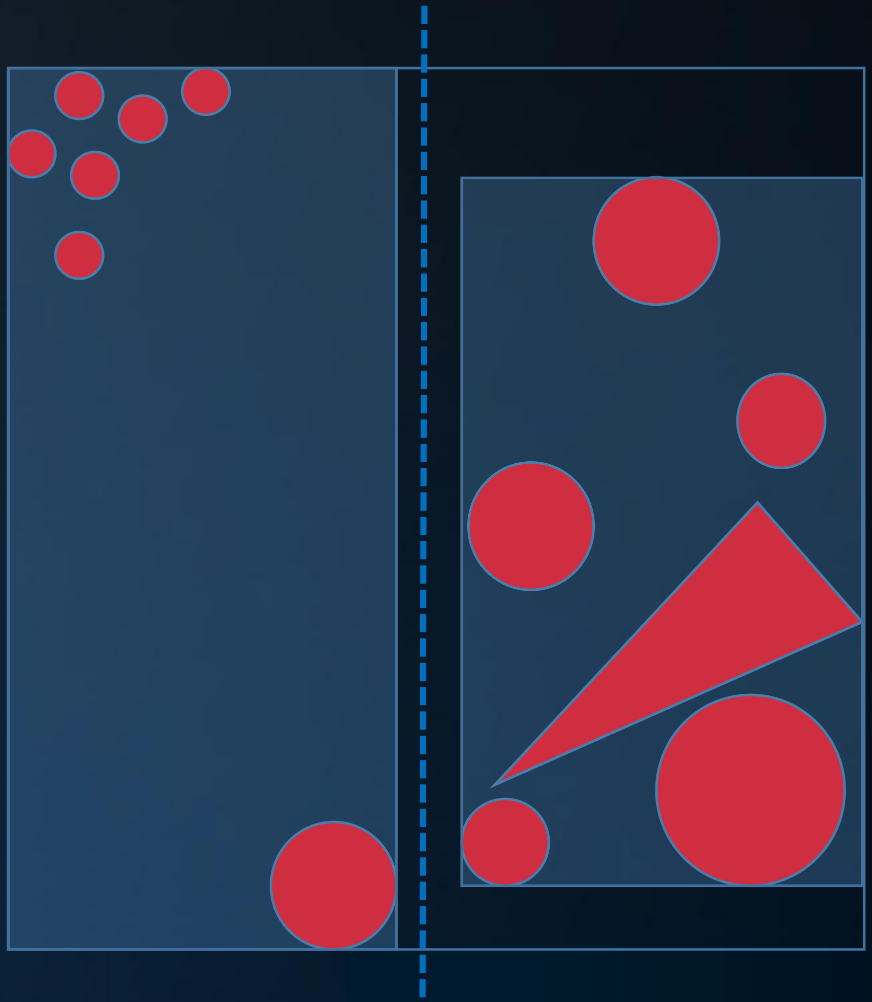
The ideal split minimizes the *expected cost* of a ray intersecting the resulting nodes.

This expected cost is based on:

- Number of primitives that will have to be intersected
- Probability of this happening

The cost of a split is thus:

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



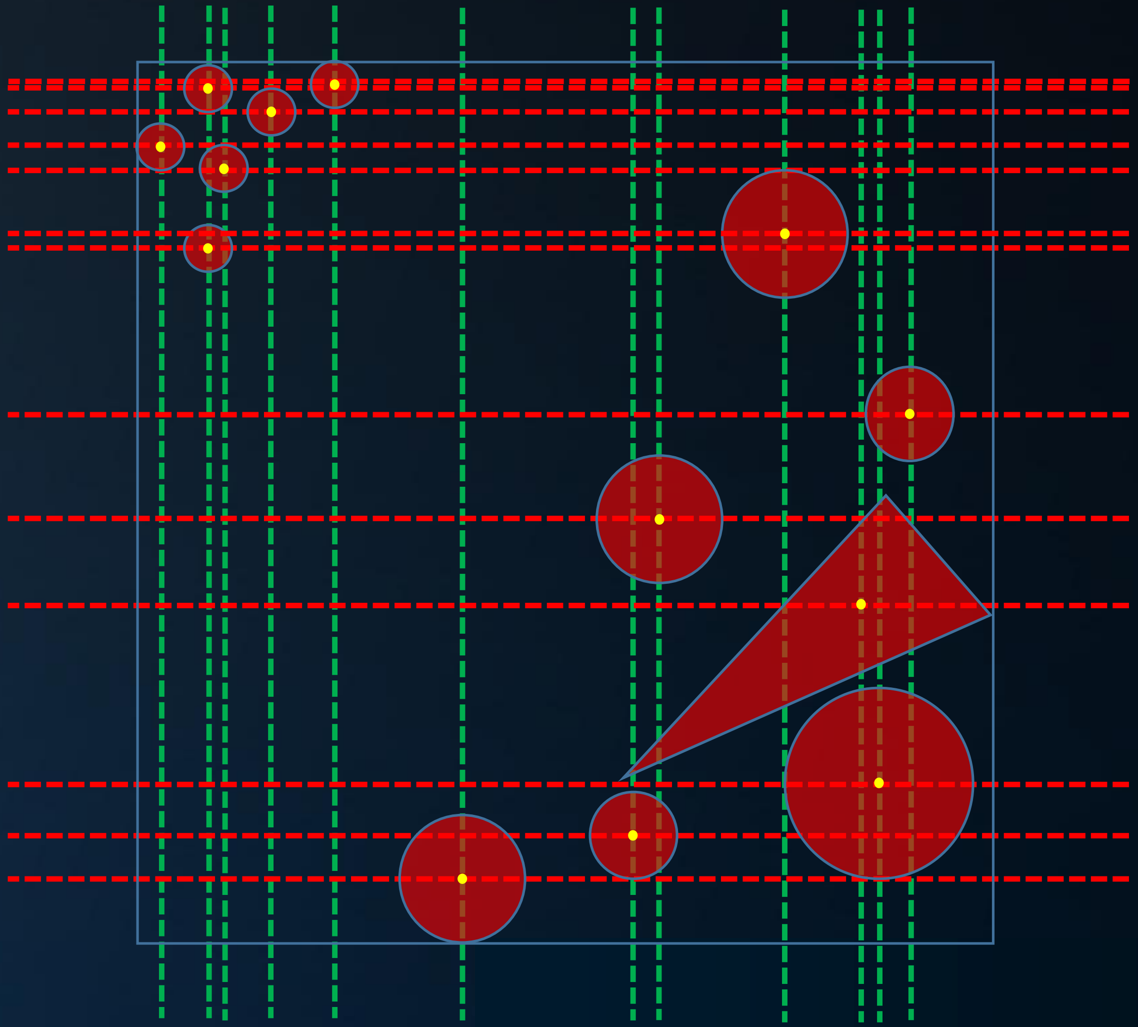
Optimal Split Plane Position

Which positions do we consider?

Object subdivision may happen over x, y or z axis.

The cost function is constant between primitive centroids.

➔ For N primitives: $3(N-1)$ possible locations



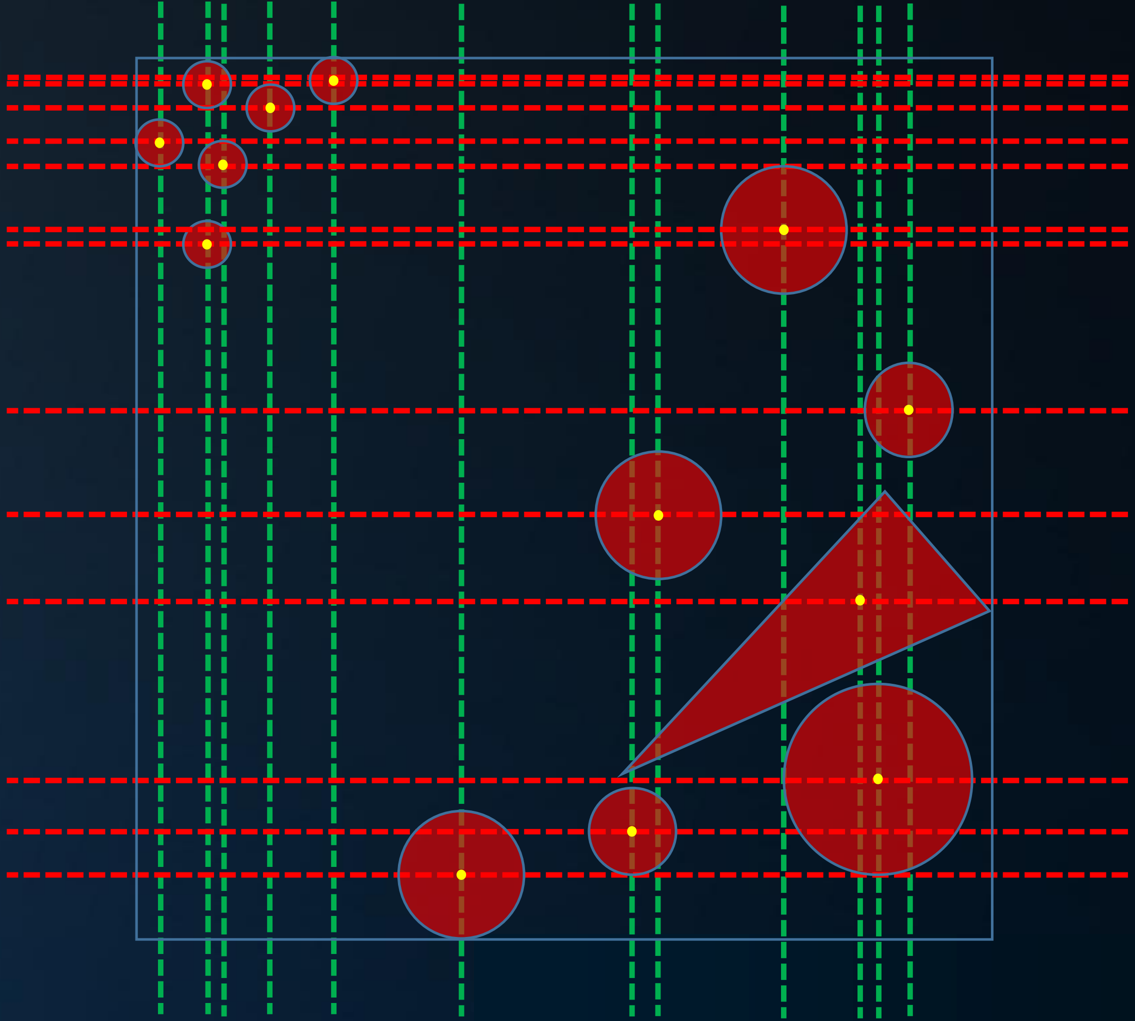
SAH and Termination

A split is 'not worth it' if it doesn't yield a cost lower than the cost of the parent node, i.e.:

$$A_{left} * N_{left} + A_{right} * N_{right} \geq A * N$$

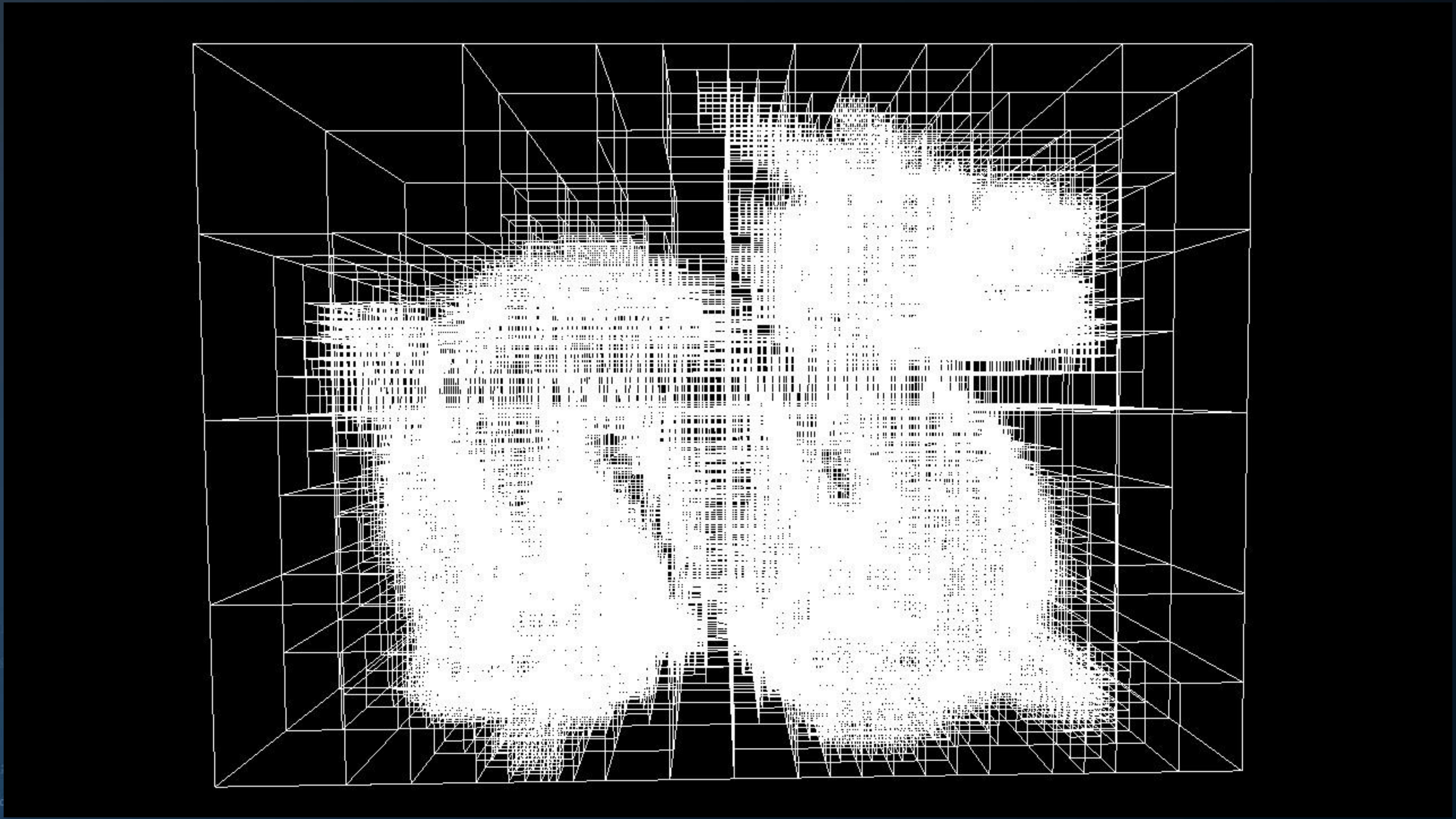
This provides us with a natural and optimal termination criterion.

(and it solves the problem of the Bad Artist)

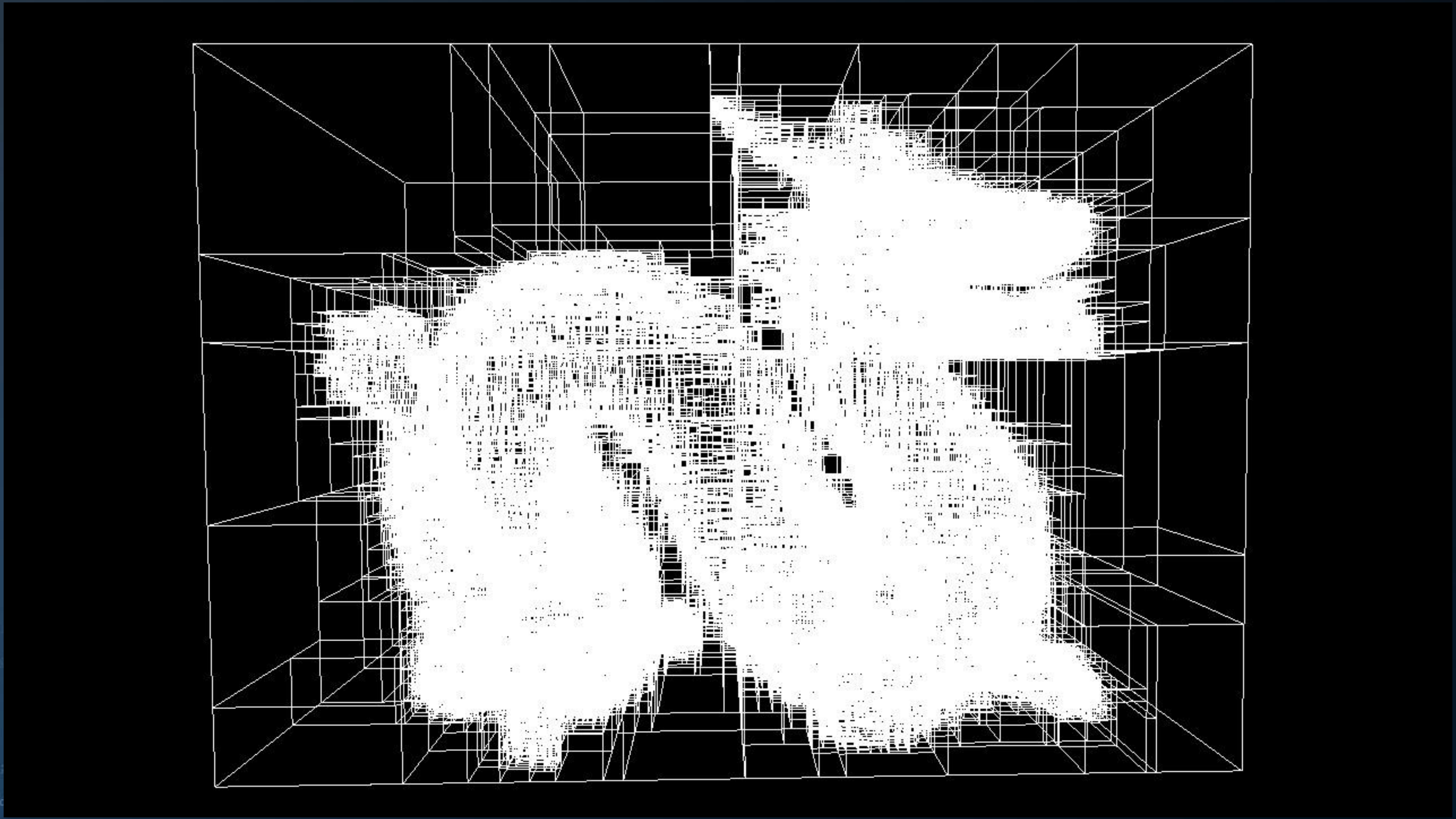


Ray Tracing for Games

```
rics
& (depth < MAXDEPTH))

c = inside ? 1 : 1.2f
nt = nt / nc, ddn = d
os2t = 1.0f - nnt * nnt
D, N );
0)

at a = nt - nc, b = nt + nc
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
 = true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse
 estimation - doing it properly, close
df;
radiance = SampleLight( &rand, I, &L, &
e.x + radiance.y + radiance.z) > 0) &&

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

andom walk - done properly, closely following
vive)

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

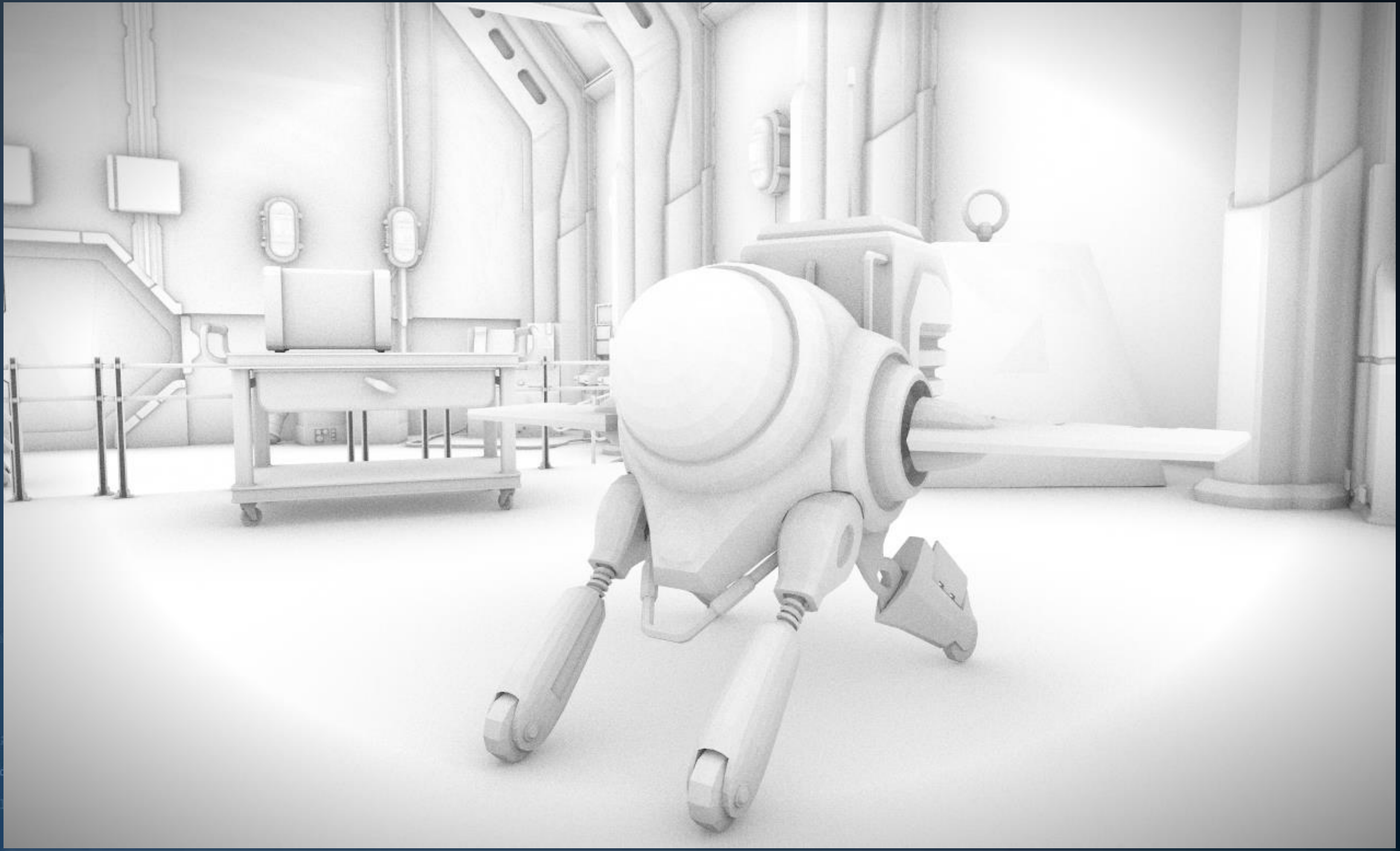


Median Split



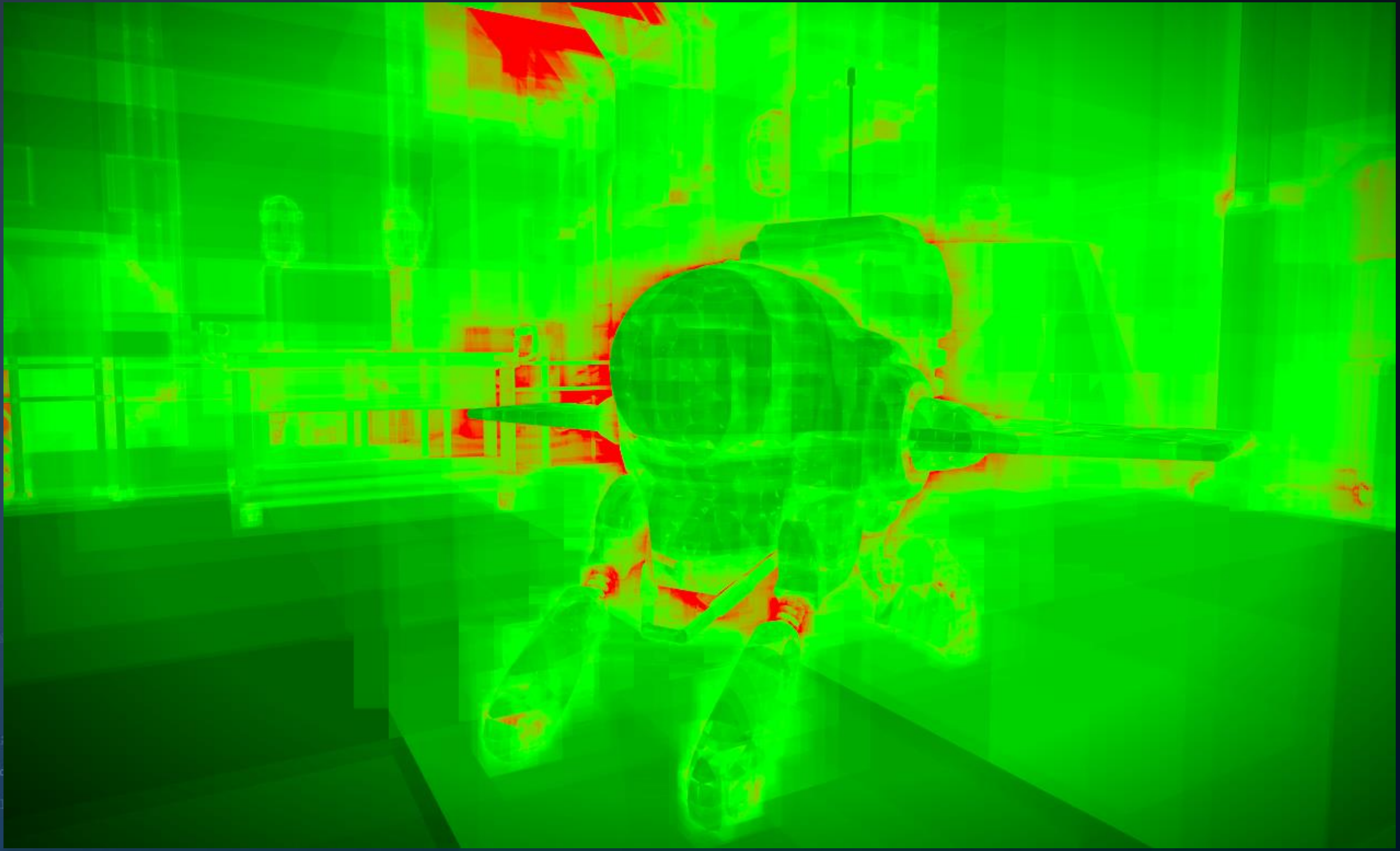
Surface Area Heuristic

Ray Tracing for Games



```
rics
& (depth < MAXDEPTH)

t = inside ? 1 : 1.2
nt = nt / nc, ddn
os2t = 1.0f - nnt * nnt
D, N );
0)

at a = nt - nc, b = nt + nc
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
 = true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse
 estimation - doing it properly, close
df;
radiance = SampleLight( &rand, I, &L,
e.x + radiance.y + radiance.z) > 0) &&

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

andom walk - done properly, closely fol
vive)

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

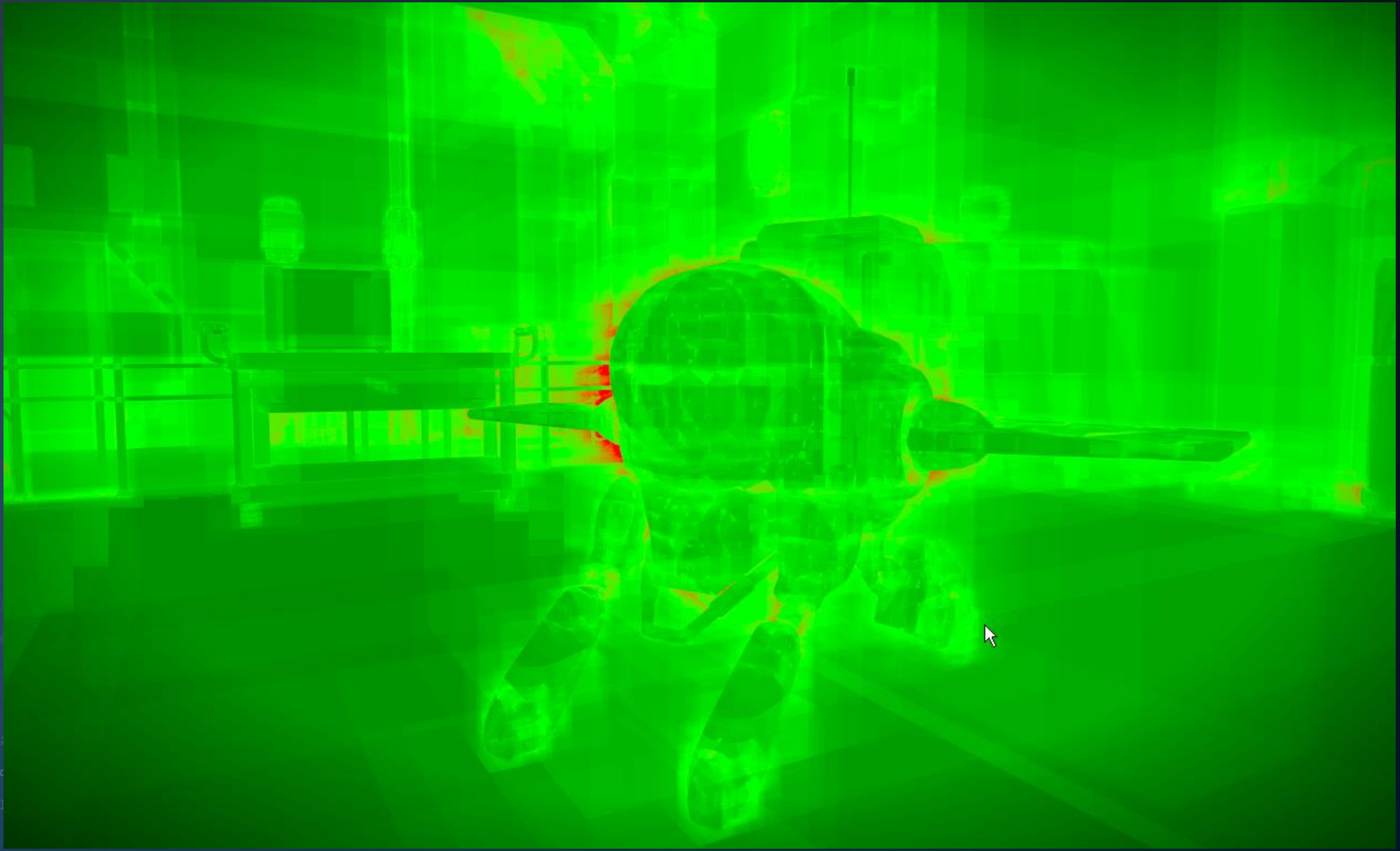
Ray Tracing for Games



```
rics
& (depth < MAXDEPTH))

c = inside ? 1
nt = nt / nc, ddn
os2t = 1.0f - nnt * nnt
D, N );
0)

at a = nt - nc, b = nt
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
 = true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse
 estimation - doing it properly, close
df;
radiance = SampleLight( &rand, I, &L,
e.x + radiance.y + radiance.z) > 0) &&

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

andom walk - done properly, closely fol
vive)

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

Ray Tracing for Games

```
rics
& (depth < MAXDEPTH))

: = inside ? 1 : 1.2f;
nt = nt / nc, ddn = dot(
os2t = 1.0f - nnt * nnt
D, N );
0)

at a = nt - nc, b = nt + nc
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
 = true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse
 estimation - doing it properly, close
df;
radiance = SampleLight( &rand, I, &L, &
e.x + radiance.y + radiance.z) > 0) &&

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

andom walk - done properly, closely fol
vive)

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



Agenda:

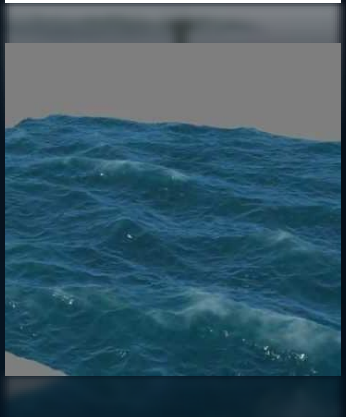
- Accelerate
- BVH
- Surface Area Heuristic
- Binning
- Fast Traversal

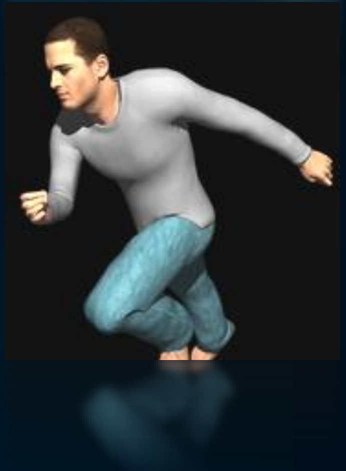
Rapid BVH Construction

Rebuilding a BVH requires $3NlogN$ split plane evaluations.

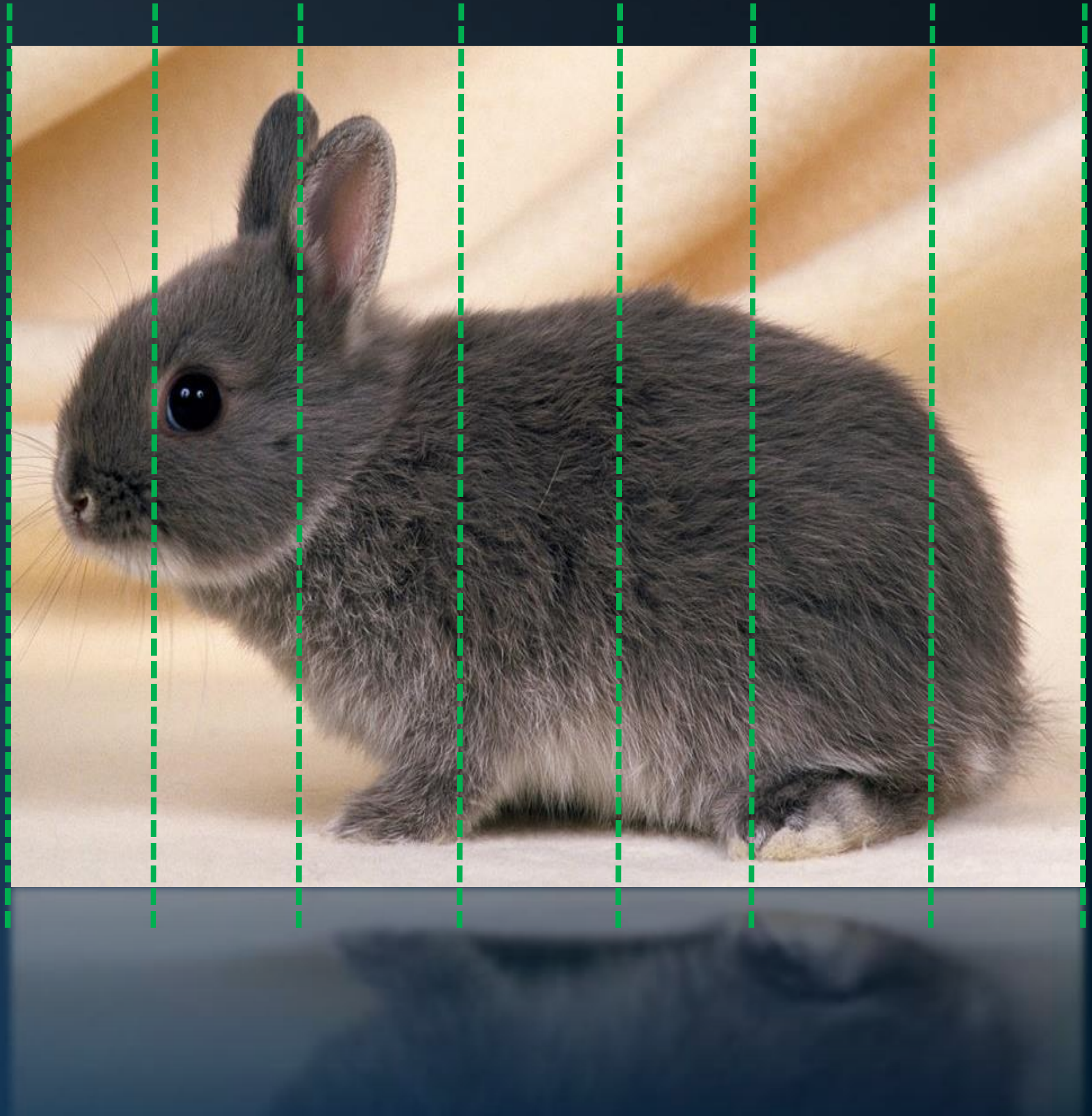
Speeding it up:

1. Do not use SAH (significantly lower quality BVH)
2. Do not evaluate all 3 axes (minor degradation of BVH quality)
3. Make split plane selection independent of N





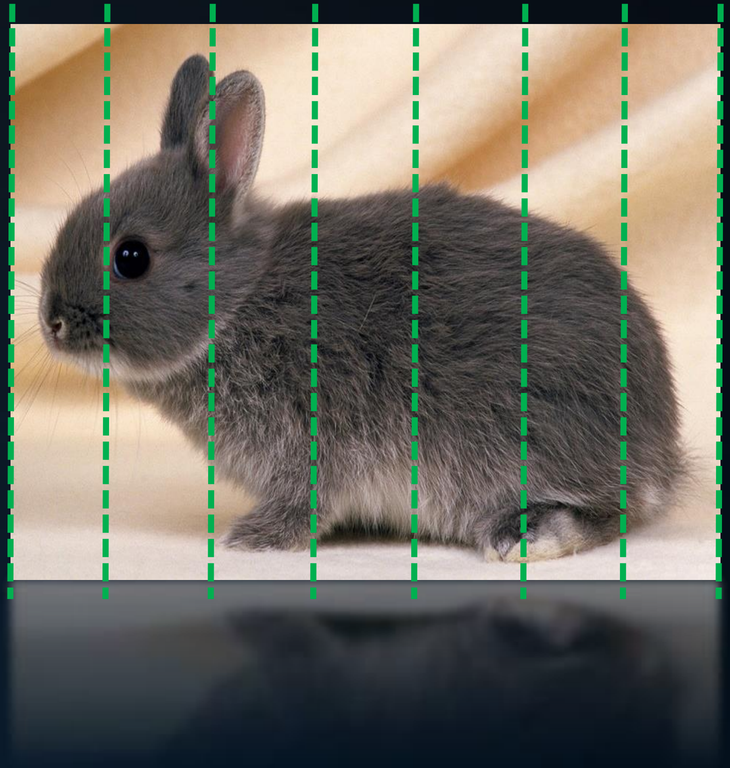
Ray Tracing for Games



```
rics
& (depth < MAXDEPTH))

c = inside ? 1 : 1.2f;
nt = nt / nc, ddn = dot(
os2t = 1.0f - nnt * nnt
D, N );
0)

at a = nt - nc, b = nt + nc
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
= true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely
df;
radiance = SampleLight( &rand, I, &L, &lightD
e.x + radiance.y + radiance.z) > 0) && (dot( N

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

andom walk - done properly, closely following Smal
vive)

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

Binned BVH Construction*

Binned construction:
Evaluate SAH at N discrete intervals.



*: On fast Construction of SAH-based Bounding Volume Hierarchies, Wald, 2007

Agenda:

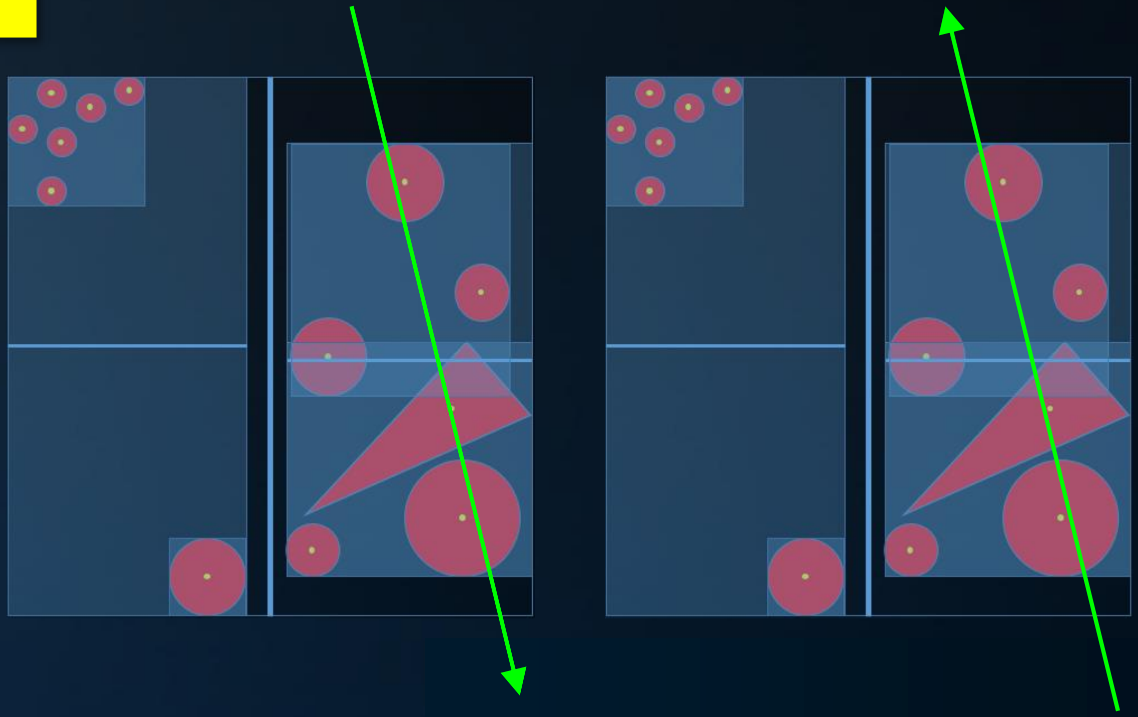
- Accelerate
- BVH
- Surface Area Heuristic
- Binning
- Fast Traversal

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 Traversal

Ordered traversal, option 1:

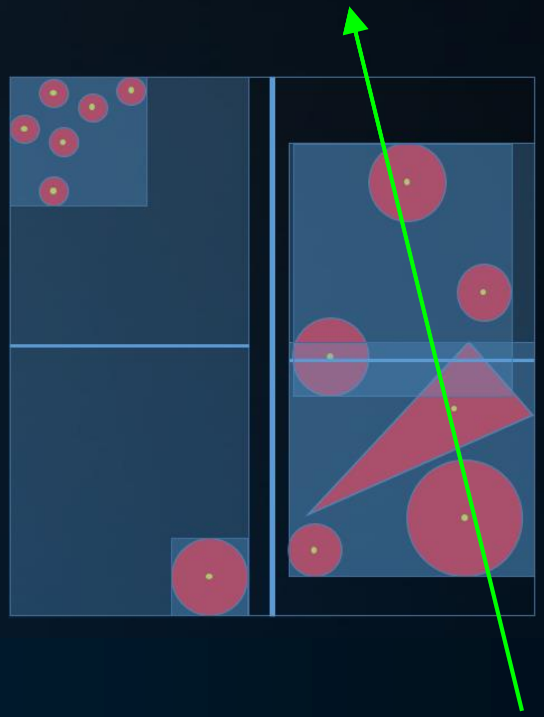
- Calculate distance to both child nodes
- Traverse the nearest child node first

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.



Ray/AABB Intersection

Vector code:

```
bool intersection( box b, ray r )
{
   __m128 t1 = _mm_mul_ps( _mm_sub_ps( node->bmin4, O4 ), rD4 );
   __m128 t2 = _mm_mul_ps( _mm_sub_ps( node->bmax4, O4 ), rD4 );
   __m128 vmax4 = _mm_max_ps( t1, t2 ), vmin4 = _mm_min_ps( t1, t2 )
   float* vmax = (float*)&vmax4, *vmin = (float*)&vmin4;
   float tmax = min(vmax[0], min(vmax[1], vmax[2]));
   float tmin = max(vmin[0], max(vmin[1], vmin[2]));
   return tmax >= tmin && tmax >= 0;
}
```

```
struct BVHNode
{
   AABB bounds;
   int leftFirst;
   int count;
};
```

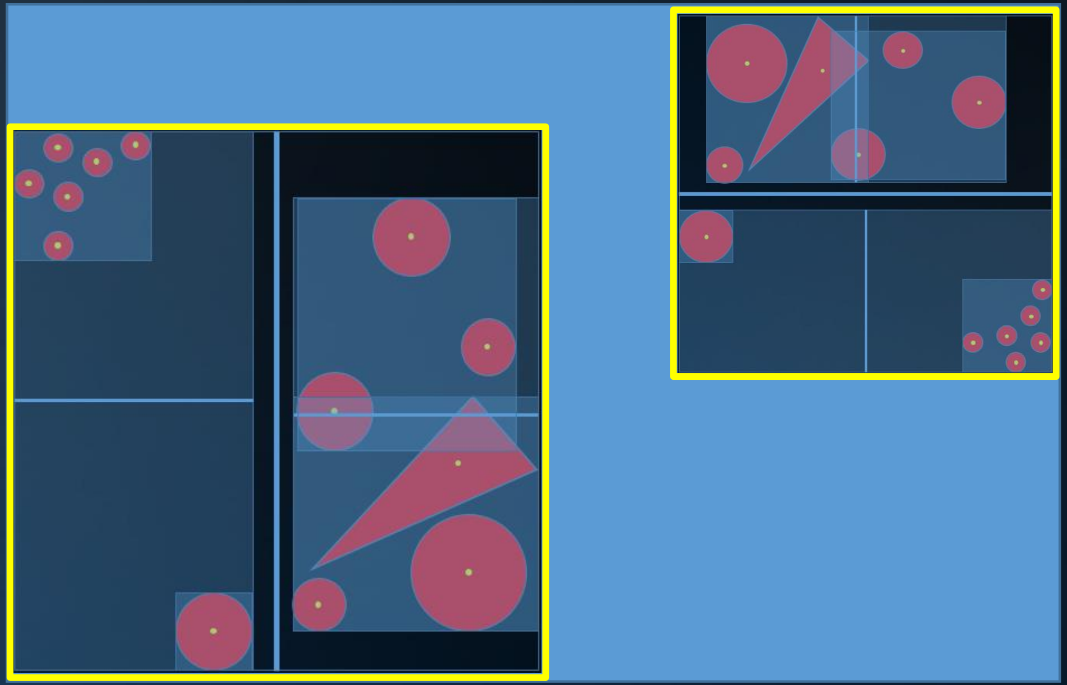
```
struct BVHNode
{
   float3 bmin;
   int leftFirst;
   float3 bmax;
   int count;
};
```

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

Agenda:

- Accelerate
- BVH
- Surface Area Heuristic
- Binning
- Fast Traversal

Combining BVHs

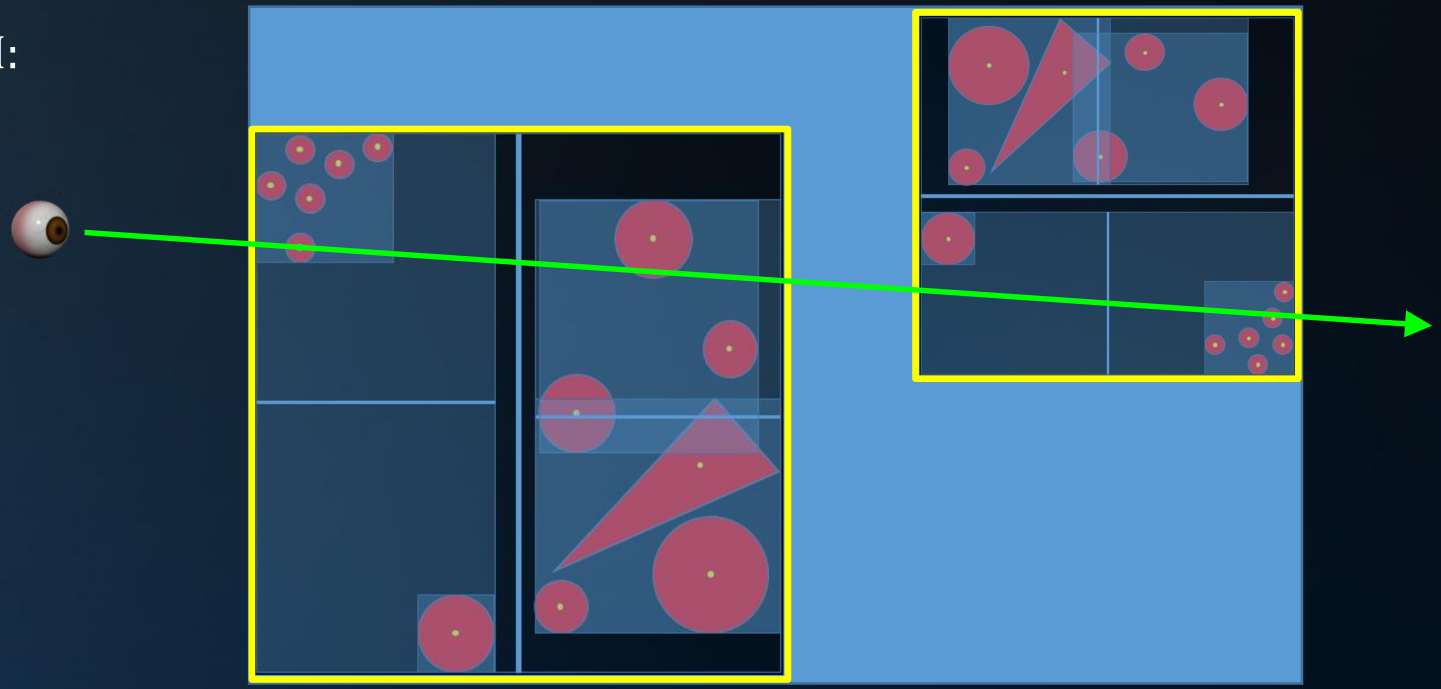


```
rics
& (depth < MAXDEPTH

: = inside ? 1 : 1.2
nt = nt / nc, ddn = d
os2t = 1.0f - nnt * nn
D, N );
0)

at a = nt - nc, b = nt + nc
at Tr = 1 - (R0 + (1 - R0)
Tr) R = (D * nnt - N * (ddn

E * diffuse;
= true;

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

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


MAXDEPTH)

survive = SurvivalProbability( diffuse )
estimation - doing it properly, closely
df;
radiance = SampleLight( &rand, I, &L, &lightD
e.x + radiance.y + radiance.z) > 0) && (dot( N

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

andom walk - done properly, closely following S
vive)

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

Rigid Motion

Applying rigid motion to a BVH:

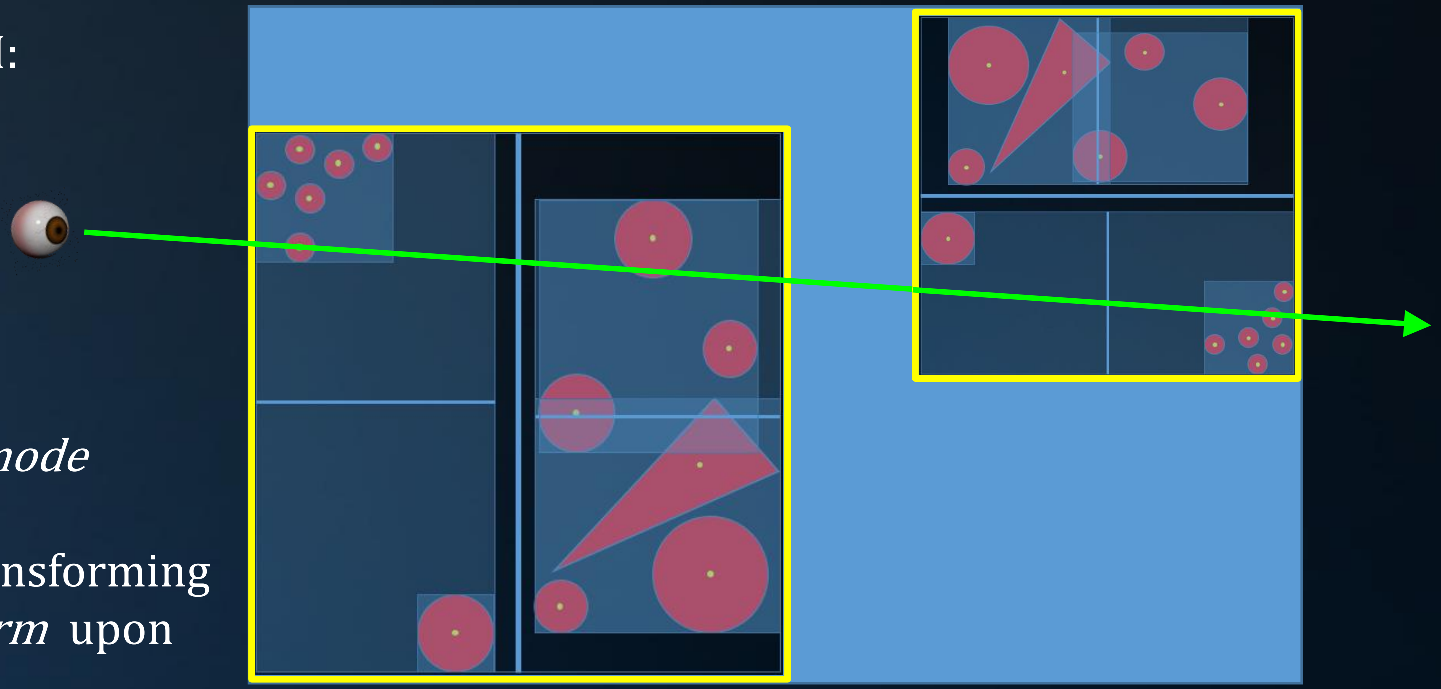
1. Refit the top-level BVH
2. Refit the affected BVH



Rigid Motion

Applying rigid motion to a BVH:

1. Refit the top-level BVH
2. Refit the affected BVH



or:

2. Transform the ray, not the node

Rigid motion is achieved by transforming the rays by the *inverse transform* upon entering the sub-BVH.

(this obviously does not only apply to translation)

End of PART 5.