

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0.25)
    {
        nt = nt / nc; ddn = ddn * ddn;
        r2t = 1.0f - nnt * nnt;
        D, N );
    }
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * r2t);
    (Tr) R = (D * nnt - N * (ddn
    E * diffuse;
    = true;
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, close);
    if;
    radiance = SampleLight( &rand, I, &align;
    e.x + radiance.y + radiance.z );
    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 Small;
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
```

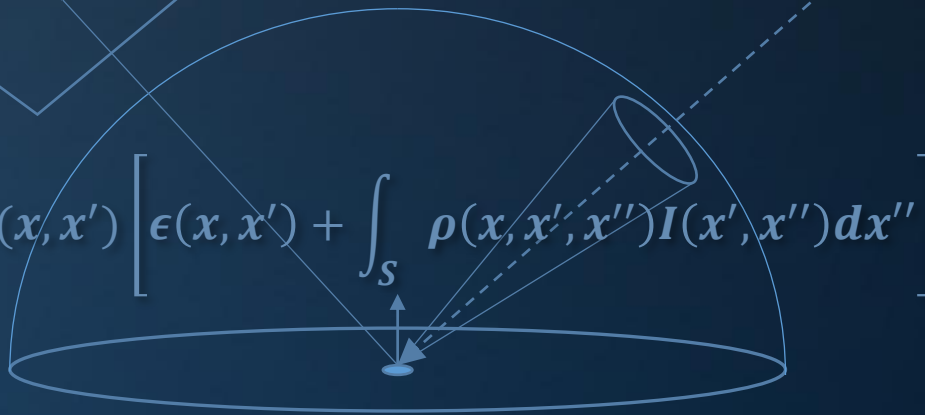


INFOMAGR – Advanced Graphics

Jacco Bikker - November 2021 - February 2022

Lecture 3 - “Acceleration Structures”

Welcome!



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



Today's Agenda:

- Problem Analysis
- Early Work
- BVH Up Close



Analysis

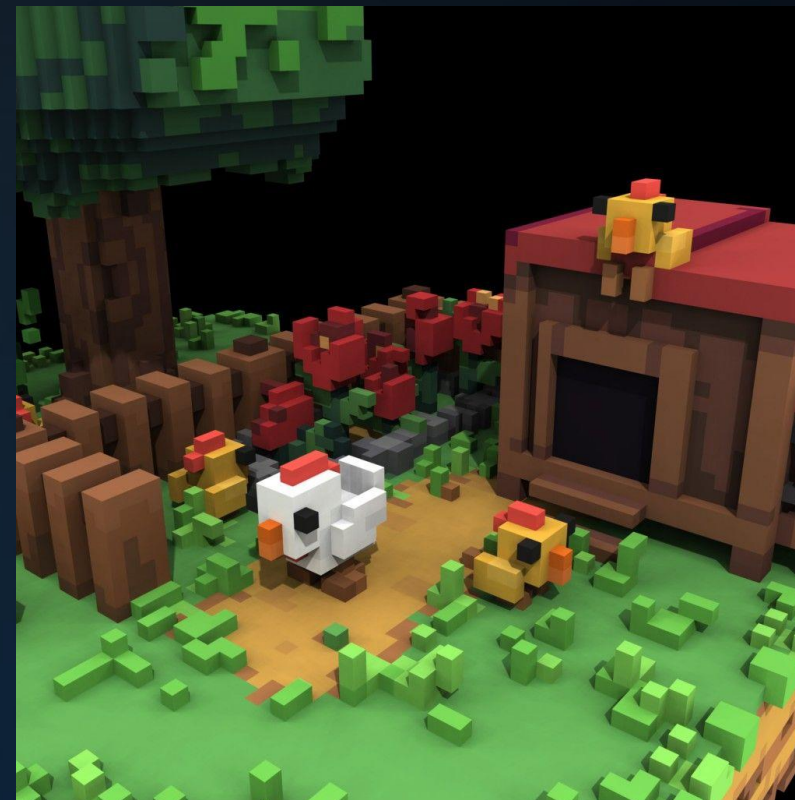
```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = dot(N, L);
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    else
    {
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * ddn);
        R = (D * nnt - N * (ddn > 0 ? 1 : -1));
        E * diffuse;
        = true;
        refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
    MAXDEPTH)
    survive = SurvivalProbability(
    estimation - doing it properly
    df;
    radiance = SampleLight( &r,
    e.x + radiance.y + radiance.z);
    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.x + radiance.y + radiance.z);
    random walk - done properly, closely following Small et al. (1999)
    survive)
    {
        at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
        survive;
        pdf;
        n = E * brdf * (dot( N, R ) / pdf);
        ion = true;
    }
}

```



“Cornell Box”



Voxel game



Analysis



Unreal 5 Tech Demo

```

w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Pdf;
at3 factor = diffuse * INVPI;
at weight = Mix2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiant
random walk - done properly, closely following S&T (1986)
vive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



Avengers Endgame



Analysis

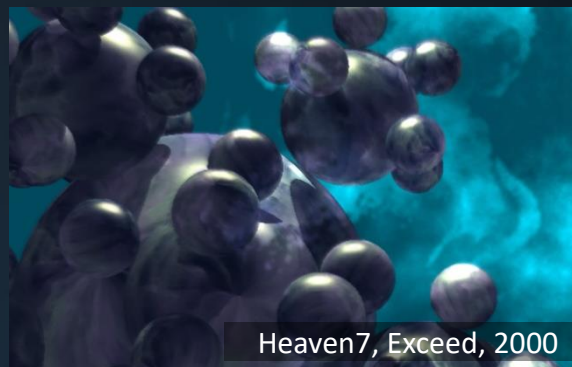
Characteristics

Rasterization:

- Games
- Fast
- Realistic
- Consumer hardware

Ray Tracing:

- Movies
- Slow
- Very Realistic
- Supercomputers



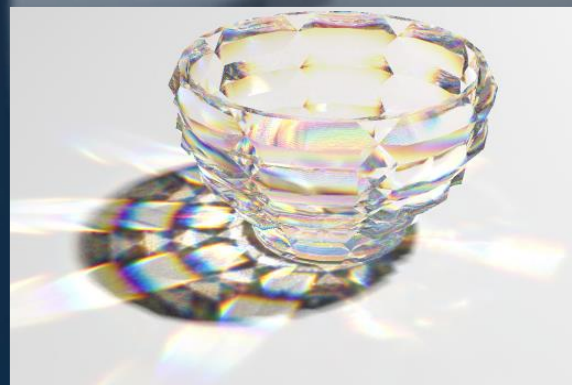
Heaven7, Exceed, 2000



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:

$$1M \times 5 \times 1k = 5 \cdot 10^9 \text{ ray/prim intersections.}$$

(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.



Today's Agenda:

- Problem Analysis
- Early Work
- BVH Up Close



Early Work

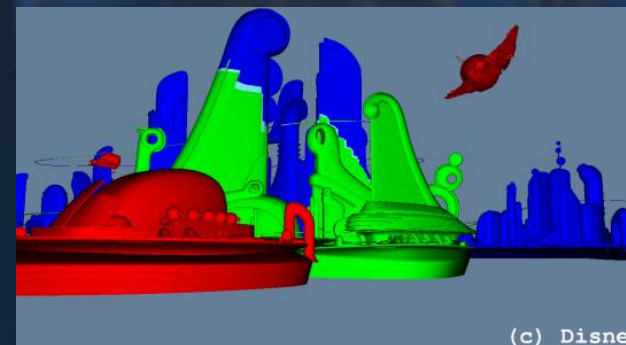
Complex Primitives

More expressive than a triangle:

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

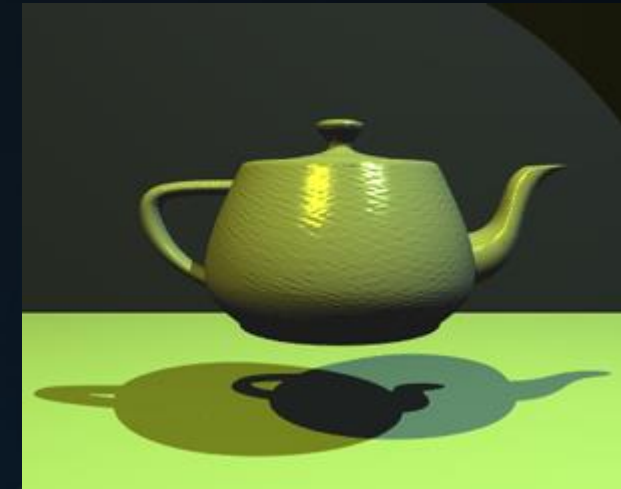


(c) Disney



(c) Disney

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

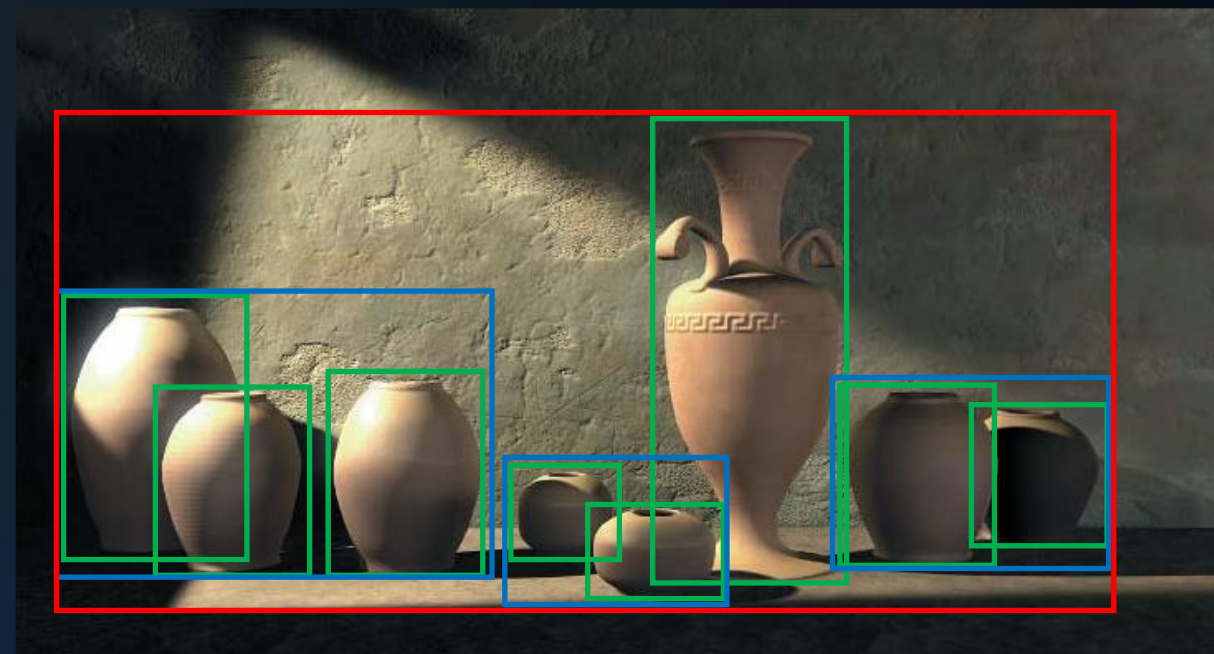
Rubin & Whitted*

“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

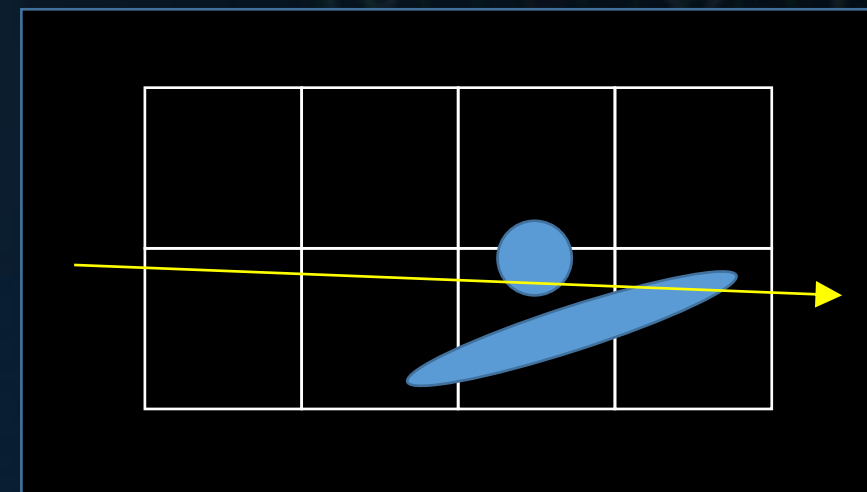
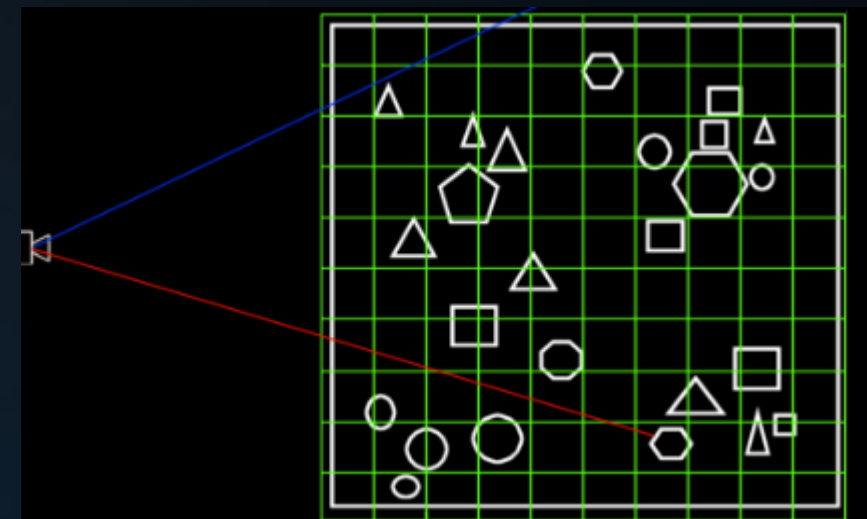
Amanatides & Woo*

“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”

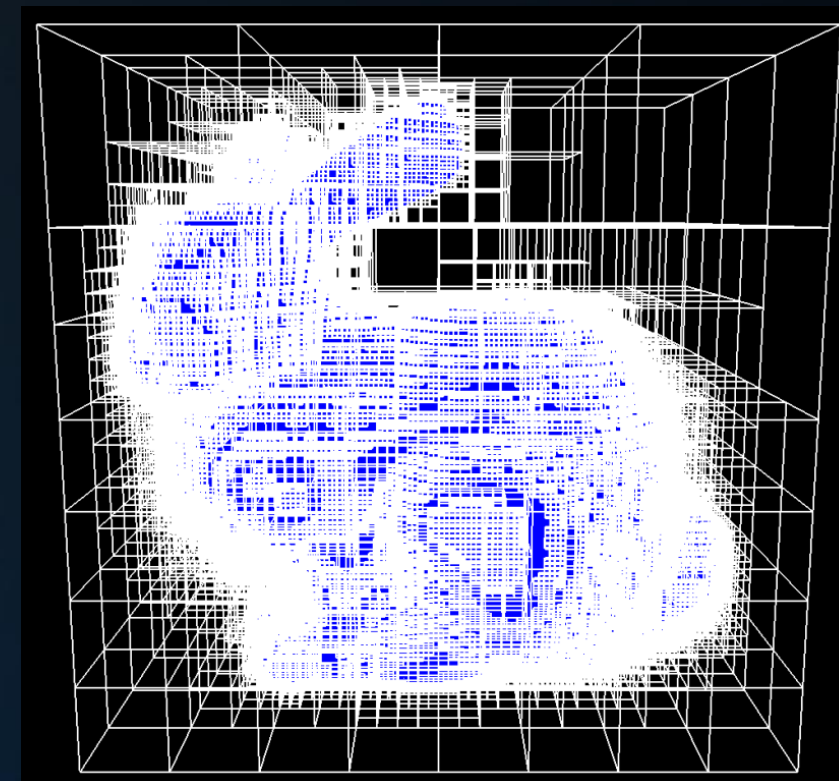
Like the grid, octrees can be automatically generated.

Advantages over grids:

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

Disadvantage compared to grids:

- Expensive traversal steps.



```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        pos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

at a = nt - nc; b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn * 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, &align,
e.x + radiance.y + radiance.z) > 0) && (acc
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 Section 2.4.1
vive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
    
```

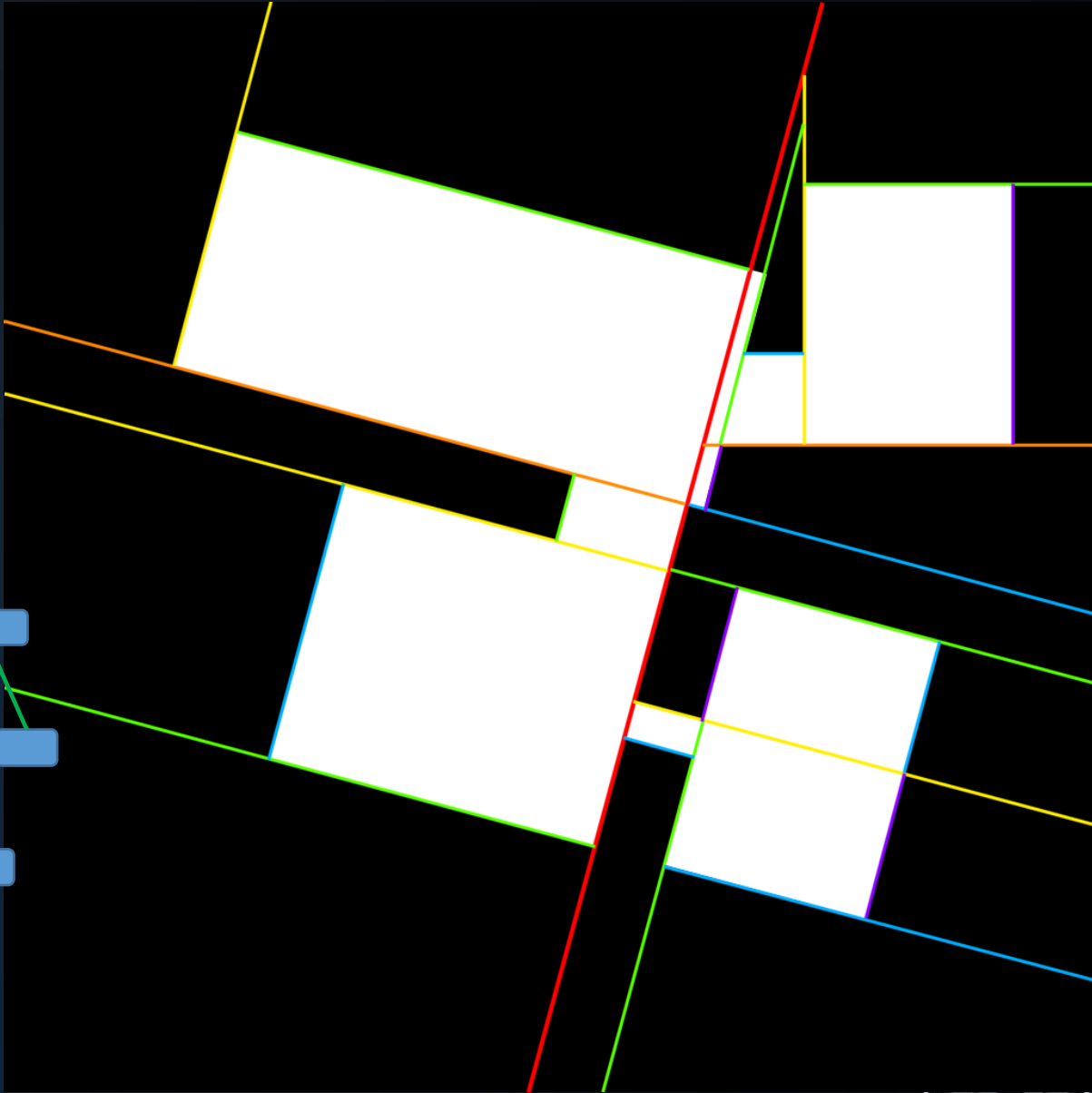
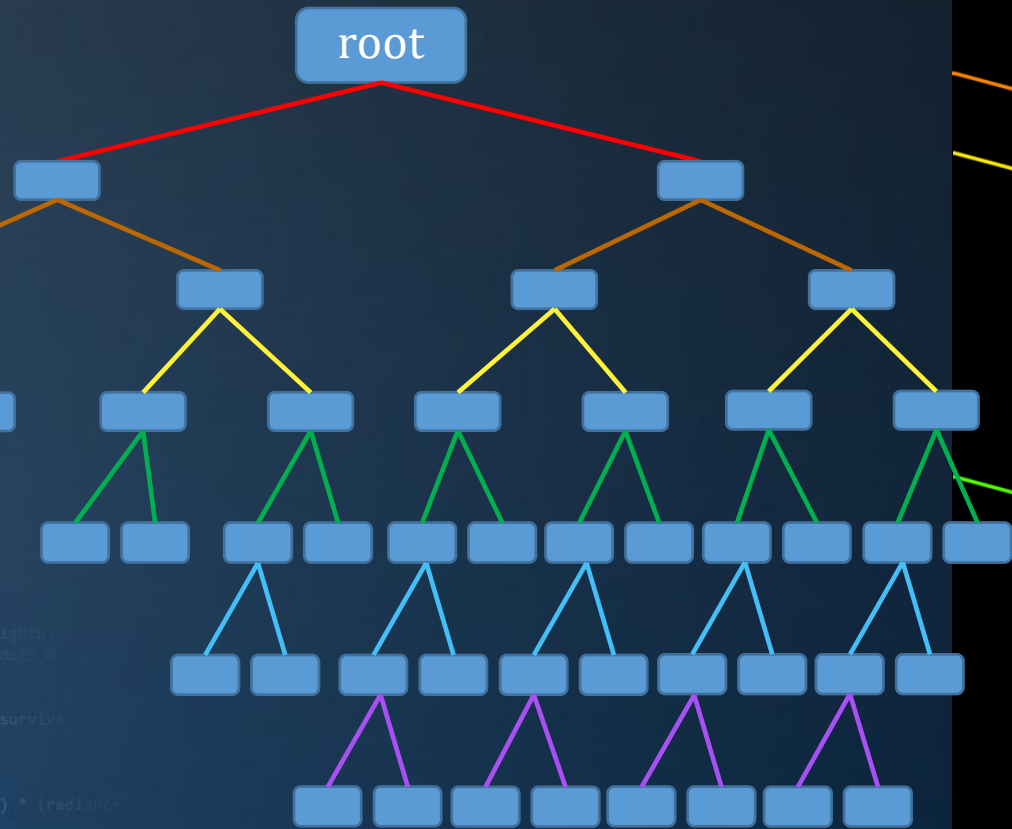
*: A. S. Glassner. Space Subdivision for Fast Ray Tracing. IEEE Computer Graphics and Applications, 4:15–22, 1984.



Early Work

BSP Trees

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn / nc;
        pos2t = 1.0f - nnt * ddn;
        D, N );
    }
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
    E * diffuse;
    = true;
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely
    df;
    radiance = SampleLight( &rand, I, &L, &align,
    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 SampleLight
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    ion = true;
}
```



Early Work

BSP Tree*

“Binary Space Partitioning”

Split planes are chosen from the geometry.

A good split plane:

- Results in equal amounts of polygons on both sides
- Splits as few polygons as possible

The BSP tends to suffer from numerical instability (splinter polygons).



*: 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”

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc, ddn = ddn / nc;
        pos2t = 1.0f - nnt * ddn;
        D, N );
    }
}

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

E * diffuse;
= true;

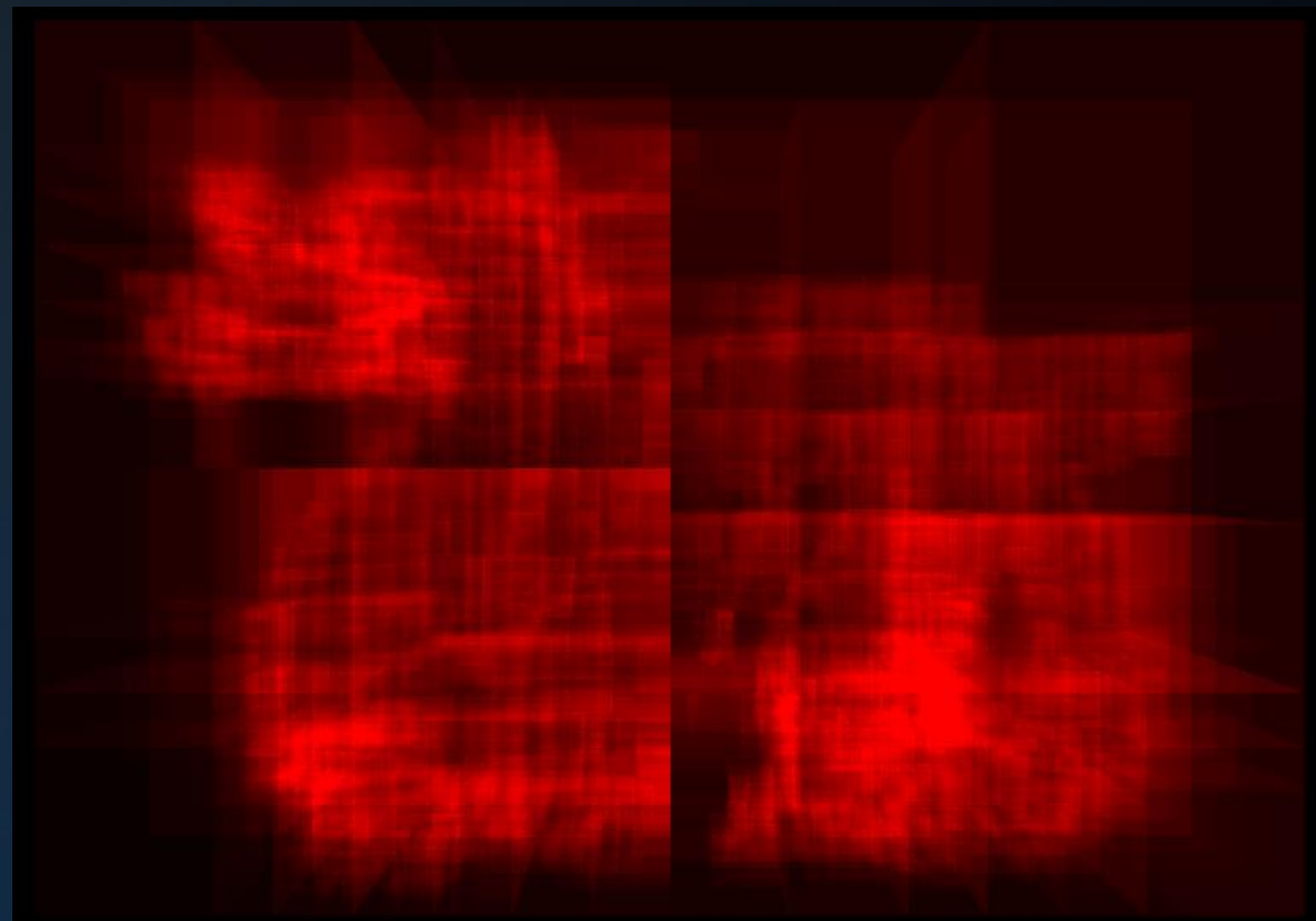
efl + refr)) && (depth < MAXDEPTH)
{
    D, N );
    refl * E * diffuse;
    = true;
}

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
df;
radiance = SampleLight( &rand, I, &L, &alignPdf );
e.x + radiance.y + radiance.z > 0) && (depth < MAXDEPTH)
{
    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
vive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



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



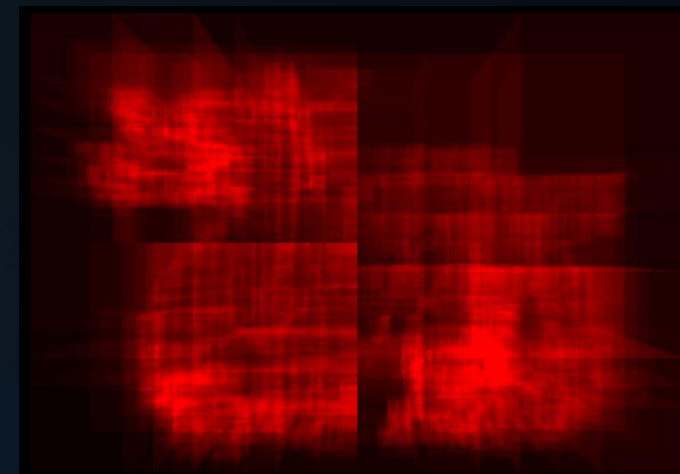
Early Work

kD-Tree Construction*

Given a scene S consisting of N primitives:

A kd-tree over S is a binary tree that recursively subdivides the space covered by S .

- The root corresponds to the axis aligned bounding box (AABB) of S ;
- 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.



```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn / nc;
        ps2t = 1.0f - nnt * ddn;
        D, N );
    }
}

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 followi
if;
radiance = SampleLight( &rand, I, &L, &lightP
e.x + radiance.y + radiance.z) > 0) && (depth <
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 followi
vive)

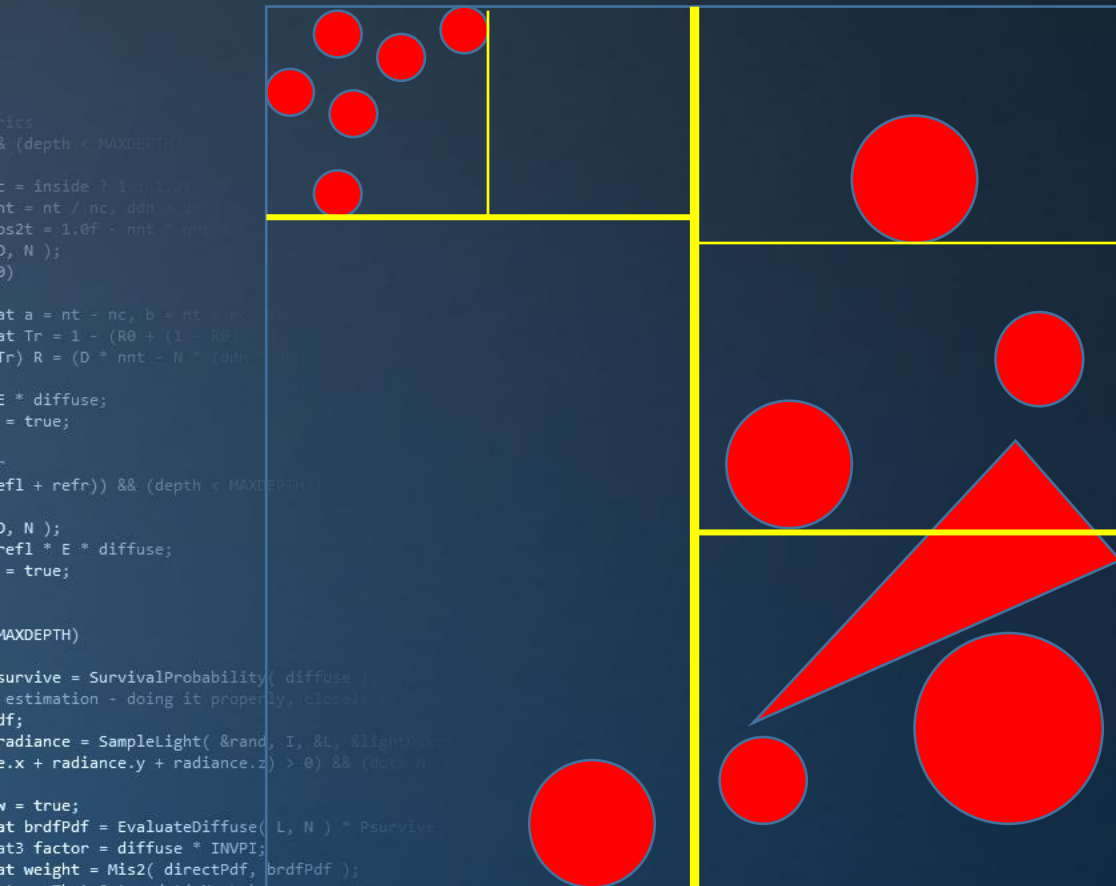
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

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



Early Work



```

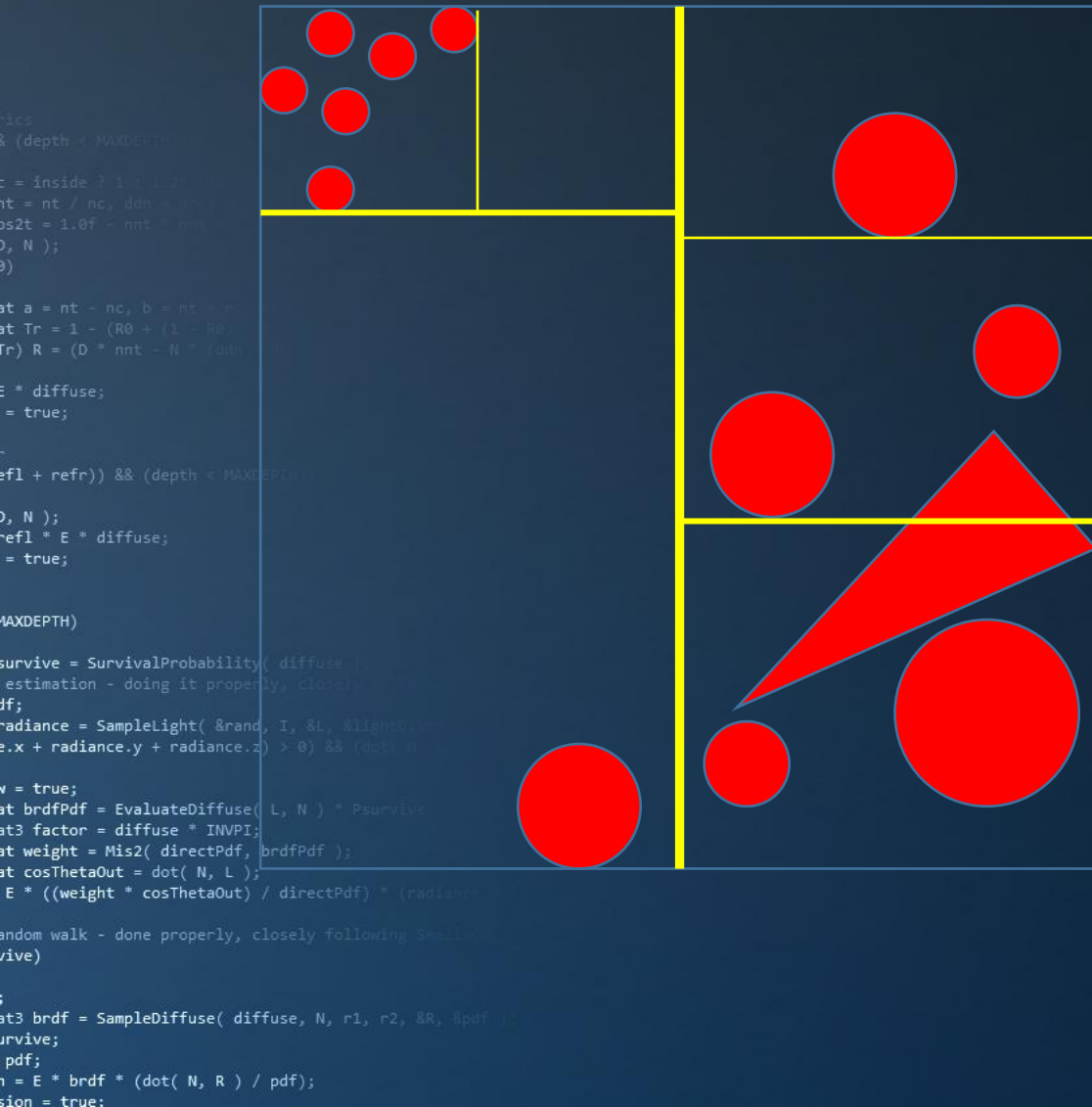
function Build( triangles  $T$ , voxel  $V$  )
{
    if (Terminate(  $T$ ,  $V$  )) return new LeafNode(  $T$  )
    Plane  $p$  = FindPlane(  $T$ ,  $V$  )
    Voxel  $V_L, V_R$  = Split  $V$  with  $p$ 
    triangles  $T_L = \{ t \in T \mid (t \cap V_L) \neq 0 \}$ 
    triangles  $T_R = \{ t \in T \mid (t \cap V_R) \neq 0 \}$ 
    return new InteriorNode(
         $p$ ,
        Build(  $T_L$ ,  $V_L$  ),
        Build(  $T_R$ ,  $V_R$  )
    )
}
  
```

```

Function BuildKdTree( triangles  $T$  )
{
    Voxel  $V = \text{bounds}(T)$ 
    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

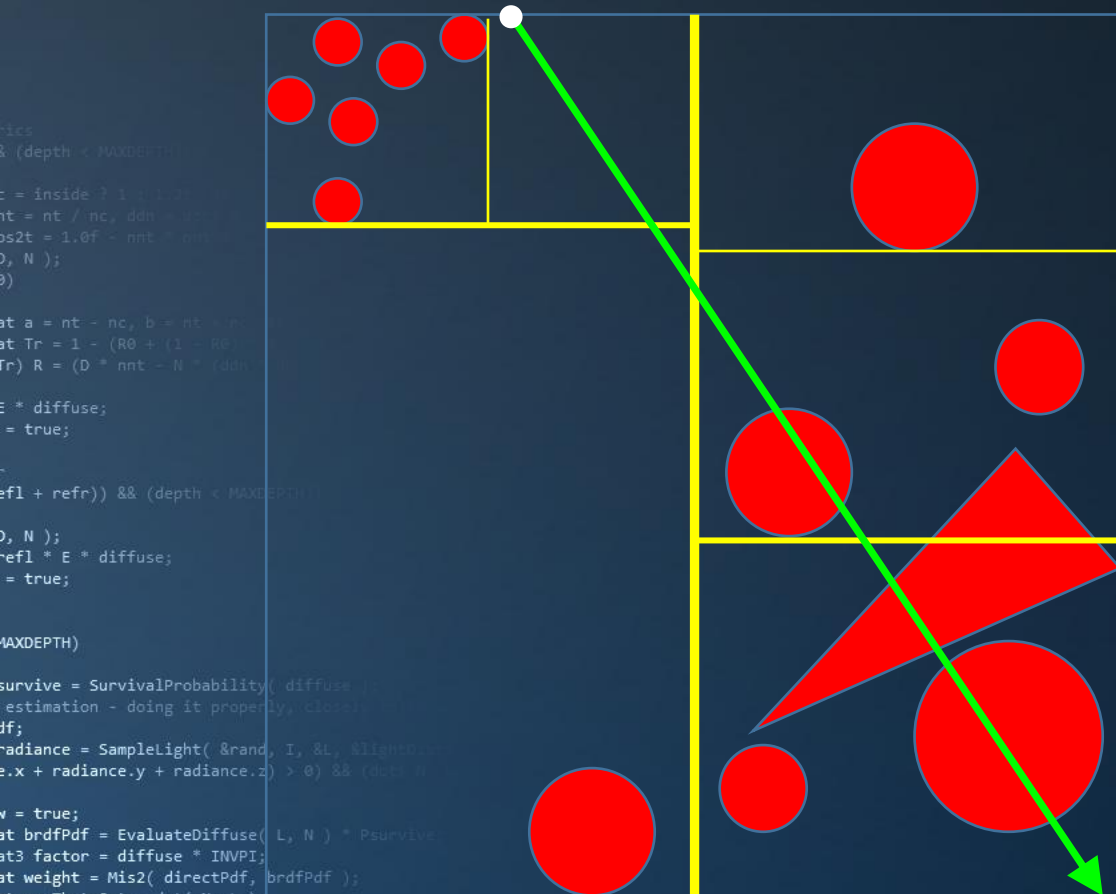
empty space reduces probability of having to intersect primitives

- Optimal split plane position / axis

good solutions exist – will be discussed later.



Early Work



Traversal*

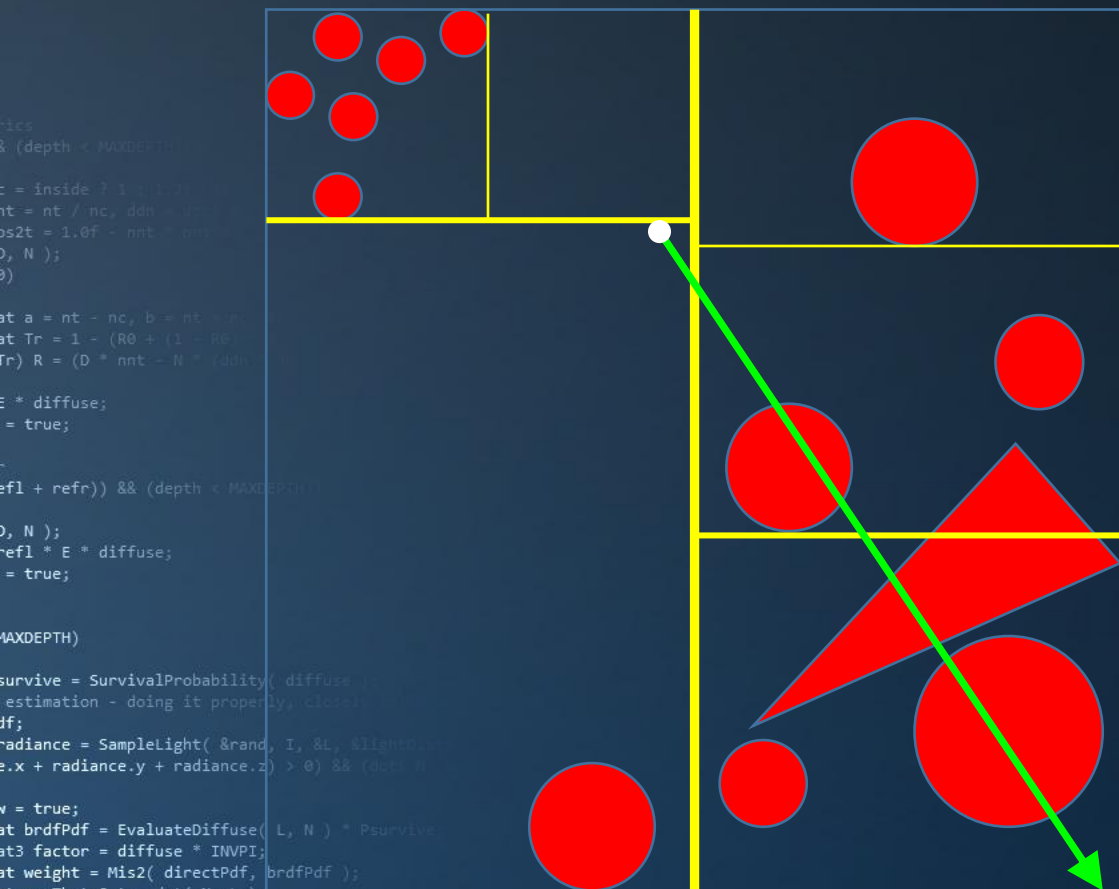
1. Find the point P where the ray enters the voxel
 2. Determine which leaf node contains this point
 3. Intersect the ray with the primitives in the leaf
- If intersections are found:

- Determine the closest intersection
- If the intersection is inside the voxel: done

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



Early Work



Traversal*

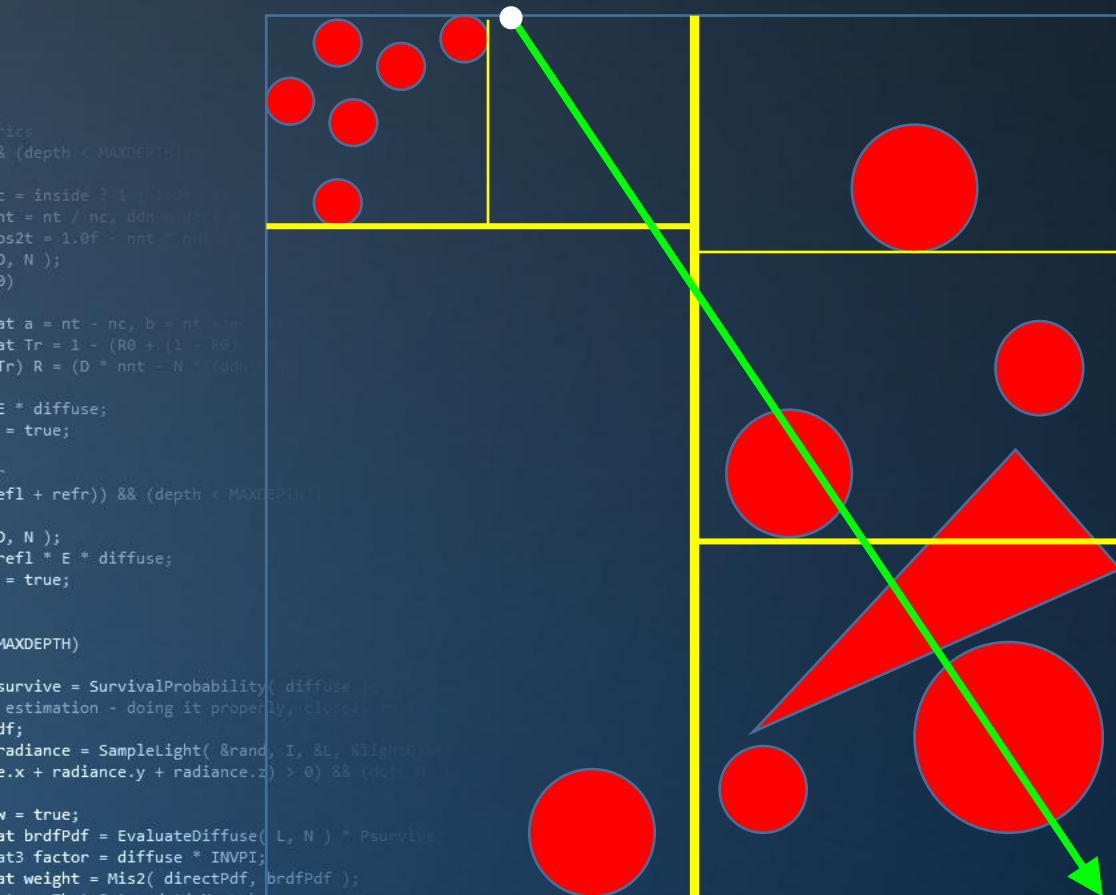
1. Find the point P where the ray enters the voxel
2. Determine which leaf node contains this point
3. Intersect the ray with the primitives in the leaf
- If intersections are found:
 - Determine the closest intersection
 - If the intersection is inside the voxel: done
4. Determine the point B where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

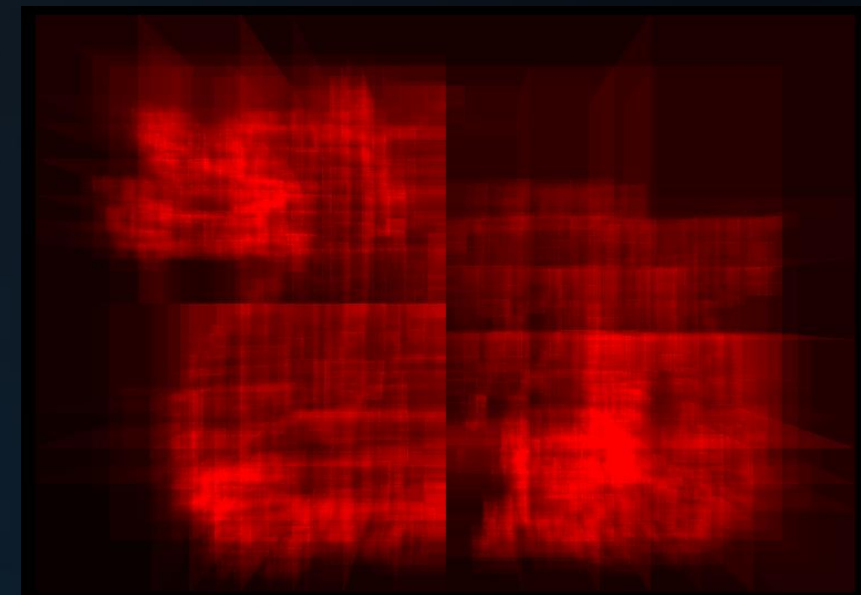
Traversing a kD-tree is done in a strict order.

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

```

ics
& (depth < MAXDEPTH)
{
    if ( ! inside ) return 0;
    int nt = nt / nc, ddn = ddn / nc;
    float ps2t = 1.0f - nnt * ddn;
    float D, N );
    0)
    {
        float a = nt - nc, b = nt * nc;
        float Tr = 1 - (R0 + (1 - R0) * a);
        float R = (D * nnt - N * (ddn * a));
        if (R < 0) R = 0;
        E * diffuse;
        = true;
        -
        refl + refr)) && (depth < MAXDEPTH)
        {
            D, N );
            refl * E * diffuse;
            = true;
        }
    }
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following Small's
    if;
    radiance = SampleLight( &rand, I, &L, &light );
    e.x + radiance.y + radiance.z) > 0) && (depth < MAXDEPTH)
    {
        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 Small's
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    ion = true;
}

```



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

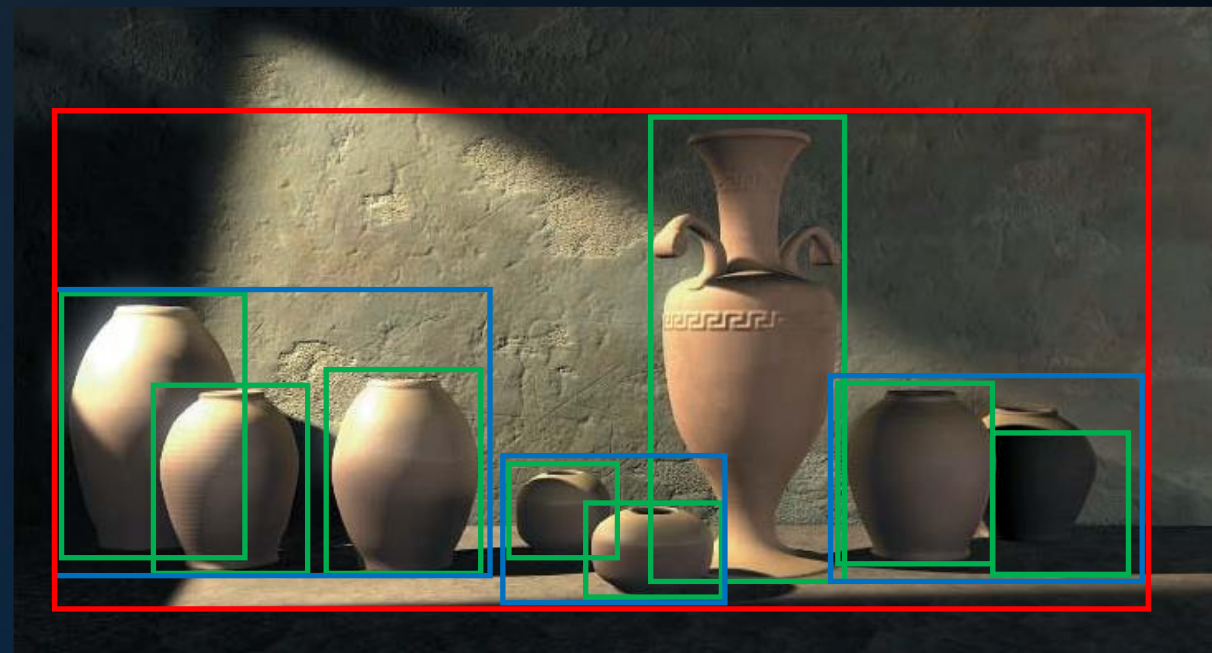
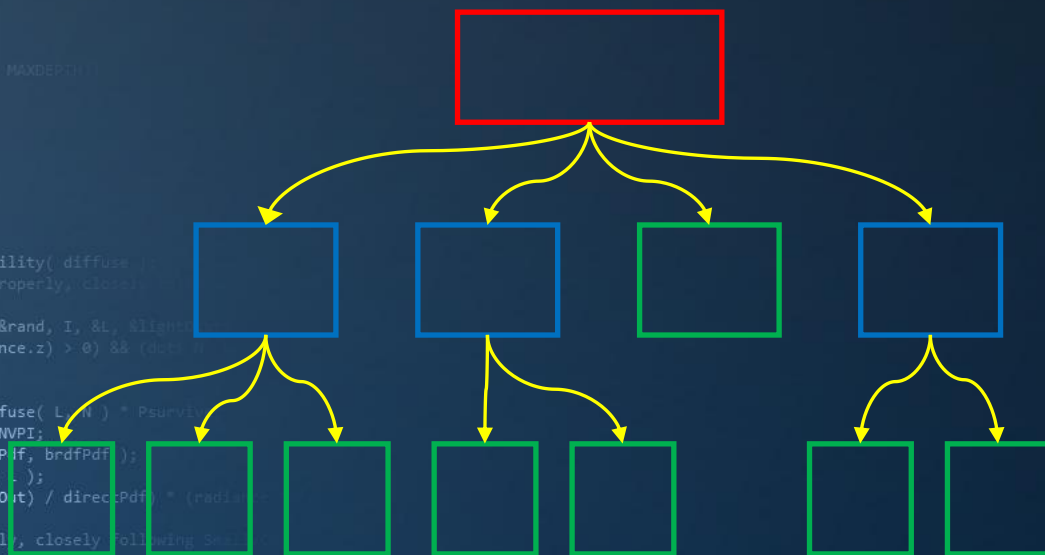
Automatic Construction of Bounding Volume Hierarchies

BVH: tree structure, with:

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

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn / nc;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    {
        at a = nt - nc, b = nt * nc;
        at Tr = 1 - (R0 + (1 - R0) * ddn);
        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
    if;
    radiance = SampleLight( &rand, I, &L, &align;
    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, . );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
    random walk - done properly, closely following
    survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
  
```

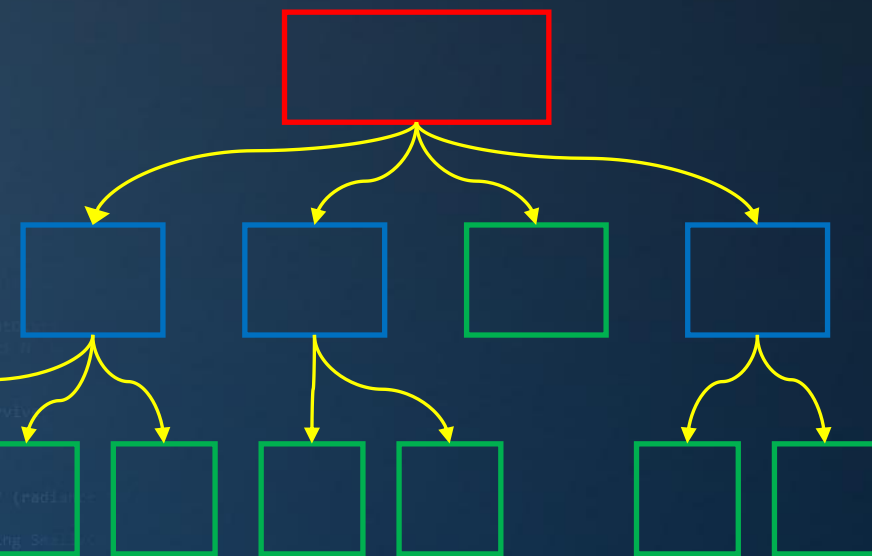


BVH

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;
};
```

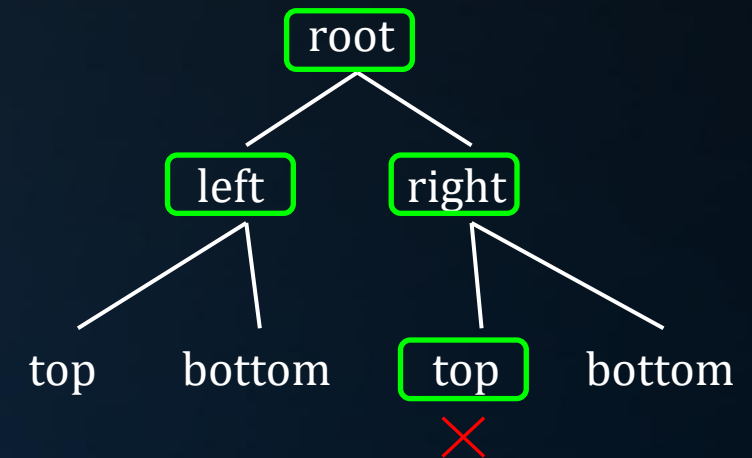
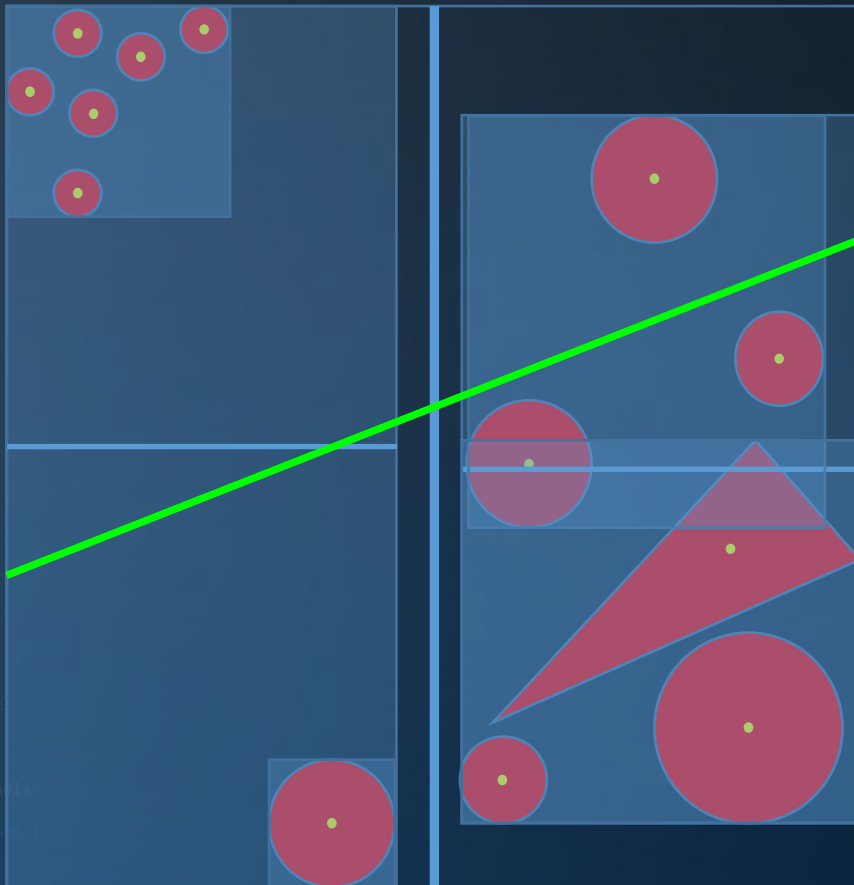


BVH

Automatic Construction of Bounding Volume Hierarchies

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0.25)
    {
        nt = nt / nc; ddn = ddn * 0.5;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) * ddn);
    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, &lightP;
    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
    vive)
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```



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’.

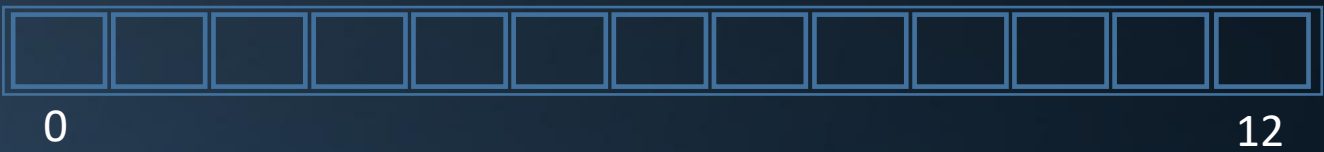


This process is identical to QuickSort: the split plane is The ‘pivot’.



BVH

Automatic Construction of Bounding Volume Hierarchies

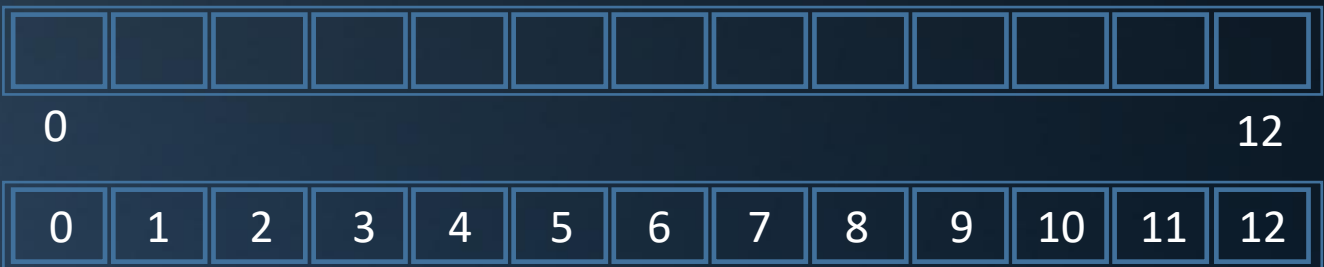


```
struct BVHNode
{
    AABB bounds;           // 24 bytes
    bool isLeaf;           // 4 bytes
    BVHNode* left, *right; // 8 or 16 bytes
    Primitive** primList;  // ? bytes
};
```



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

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn / nc;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * a);
    (Tr) R = (D * nnt - N * (ddn *
    E * diffuse;
    = true;
    -
    efl + refr)) && (depth < MAXDEPTH)
    D, N );
    efl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if;
    radiance = SampleLight( &rand, I, &L, &align, &
    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 Section
    ve)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    ion = true;

```

```

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();

```

```

}

```

```

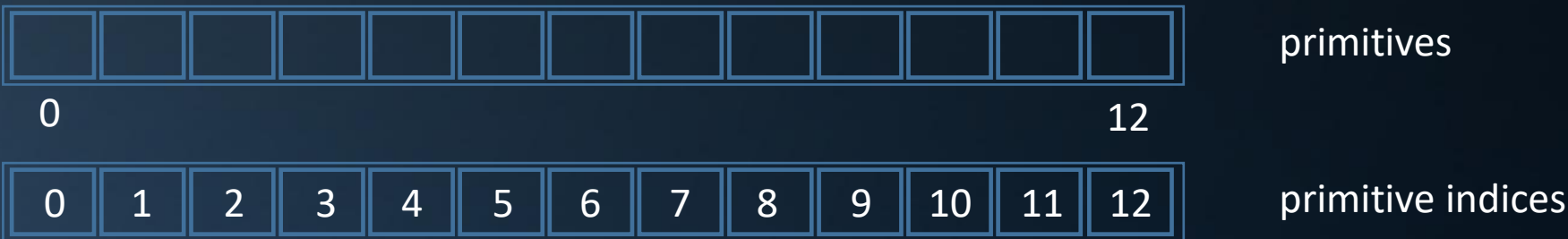
void BVHNode::Subdivide()
{
    if (count < 3) return;
    this->left = &pool[poolPtr++];
    this->right = &pool[poolPtr++];
    Partition();
    this->left->Subdivide();
    this->right->Subdivide();
    this->isLeaf = false;
}

```



BVH

Automatic Construction of Bounding Volume Hierarchies

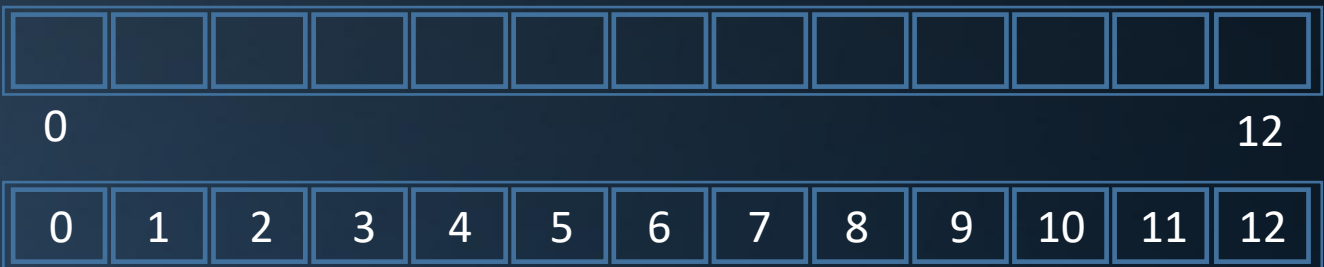


```
struct BVHNode
{
    AABB bounds;           // 24 bytes
    bool isLeaf;           // 4 bytes
    int left, right;       // 8 bytes
    int first, count;      // 8 bytes, total 44 bytes
};
```



BVH

Automatic Construction of Bounding Volume Hierarchies



primitives

primitive indices

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

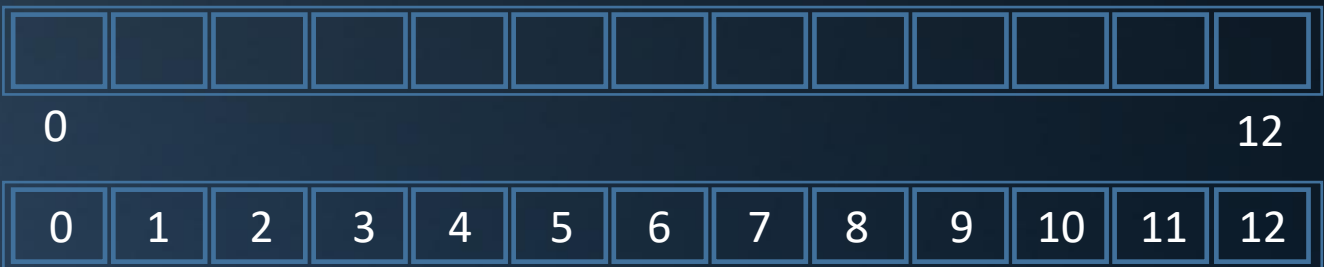


BVH nodes



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'.

```

ics
& (depth < MAXDEPTH)
{
    float t = inside ? 1.0f : 0.0f;
    float nt = nt / nc; ddn = ddn * nt;
    float cos2t = 1.0f - nnt * nnt;
    float D, N );
    float a = nt - nc; b = nt * nc;
    float Tr = 1 - (R0 + (1 - R0) * t);
    float R = (D * nnt - N * (ddn *
    float E * diffuse;
    float = true;
    float refl + refr)) && (depth < MAXDEPTH)
    float D, N );
    float refl * E * diffuse;
    float = true;
    float MAXDEPTH)
    float survive = SurvivalProbability( diffuse );
    float estimation - doing it properly, closely following
    float if;
    float radiance = SampleLight( &rand, I, &L, &light );
    float e.x + radiance.y + radiance.z) > 0) && (depth <
    float w = true;
    float at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    float at3 factor = diffuse * INVPI;
    float at weight = Mis2( directPdf, brdfPdf );
    float at cosThetaOut = dot( N, L );
    float E * ((weight * cosThetaOut) / directPdf) * (radiance
    float random walk - done properly, closely following Small's
    float vive)
    float at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    float survive;
    float pdf;
    float n = E * brdf * (dot( N, R ) / pdf);
    float sion = true;

```



BVH

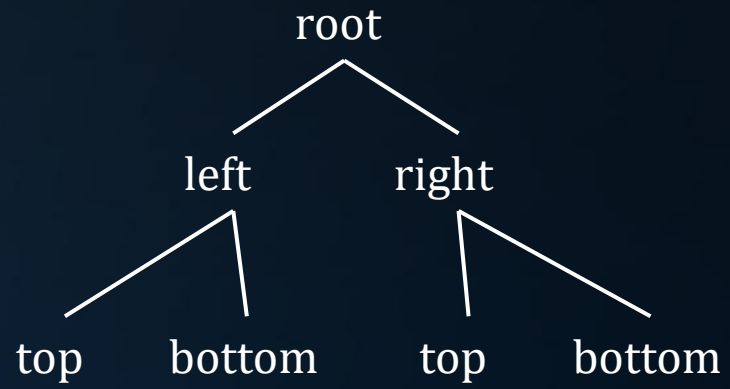
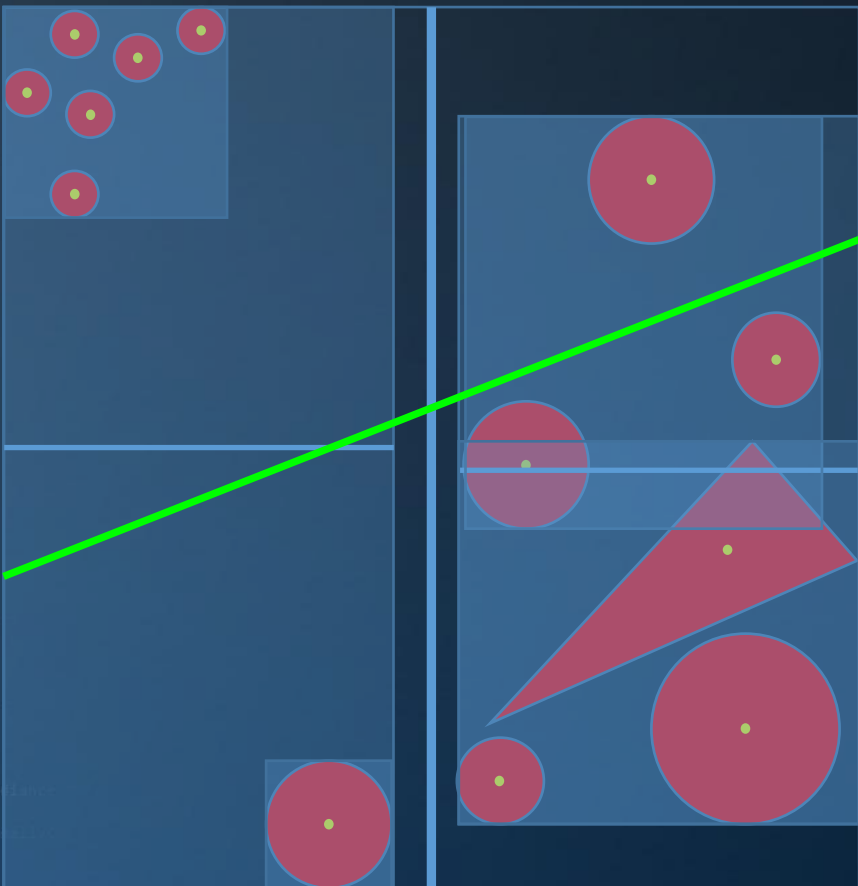
BVH Traversal

```
...ics
&& (depth < MAXDEPTH)
{
    t = inside ? 1 : -1;
    nt = nt / nc; ddn = ddn * t;
    cos2t = 1.0f - nnt * nnt;
    D, N );
    )
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * t);
    Tr) R = (D * nnt - N * (ddn > 0 ? 1 : -1));

    E * diffuse;
    = true;

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

    MAXDEPTH)
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    df;
    radiance = SampleLight( &rand, I, &L, &lightP,
    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
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    ion = true;
```



BVH

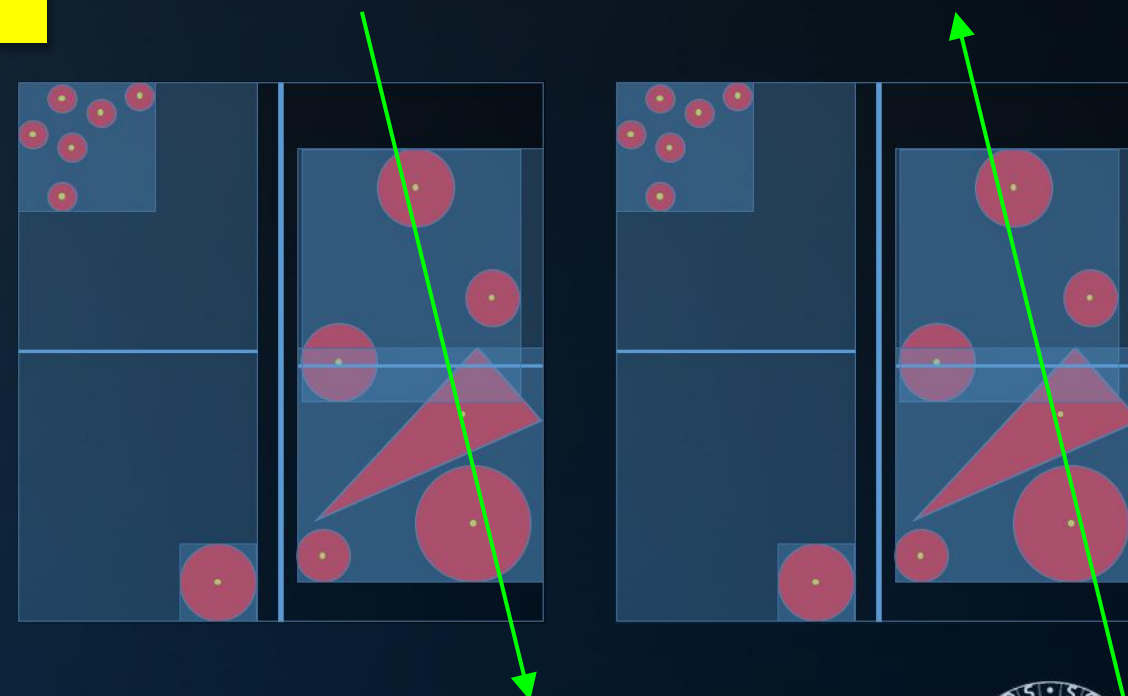
BVH Traversal

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:

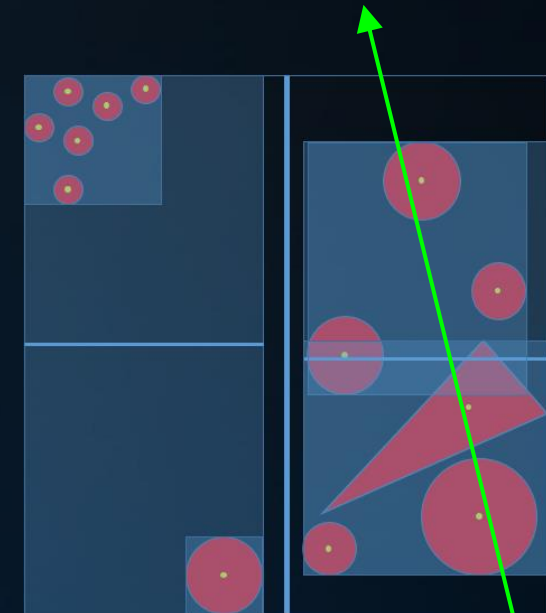
- Calculate distance to both child nodes
- Traverse the nearest child node first

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.

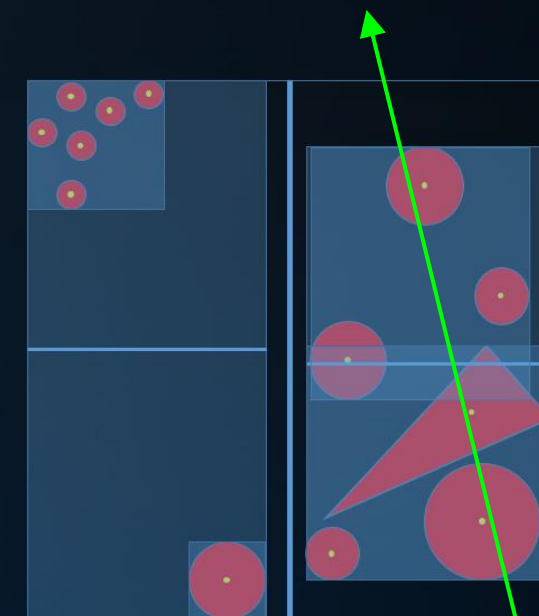
- Nodes may overlap.

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 2021 - February 2022

END of “Acceleration Structures”

next lecture: “Light Transport”

