

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0.25)
    {
        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) * pos2t);
    (Tr) R = (D * nnt - N * (ddn * pos2t));
    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, N, 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's
    (survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

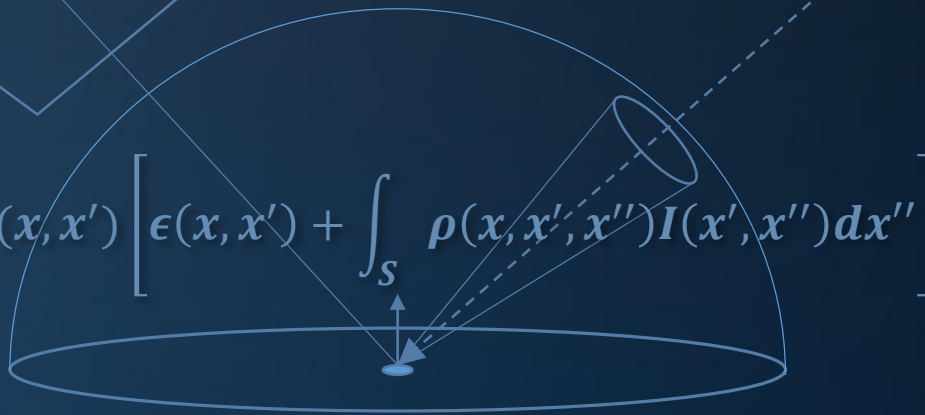


INFOMAGR – Advanced Graphics

Jacco Bikker - November 2019 - February 2020

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:

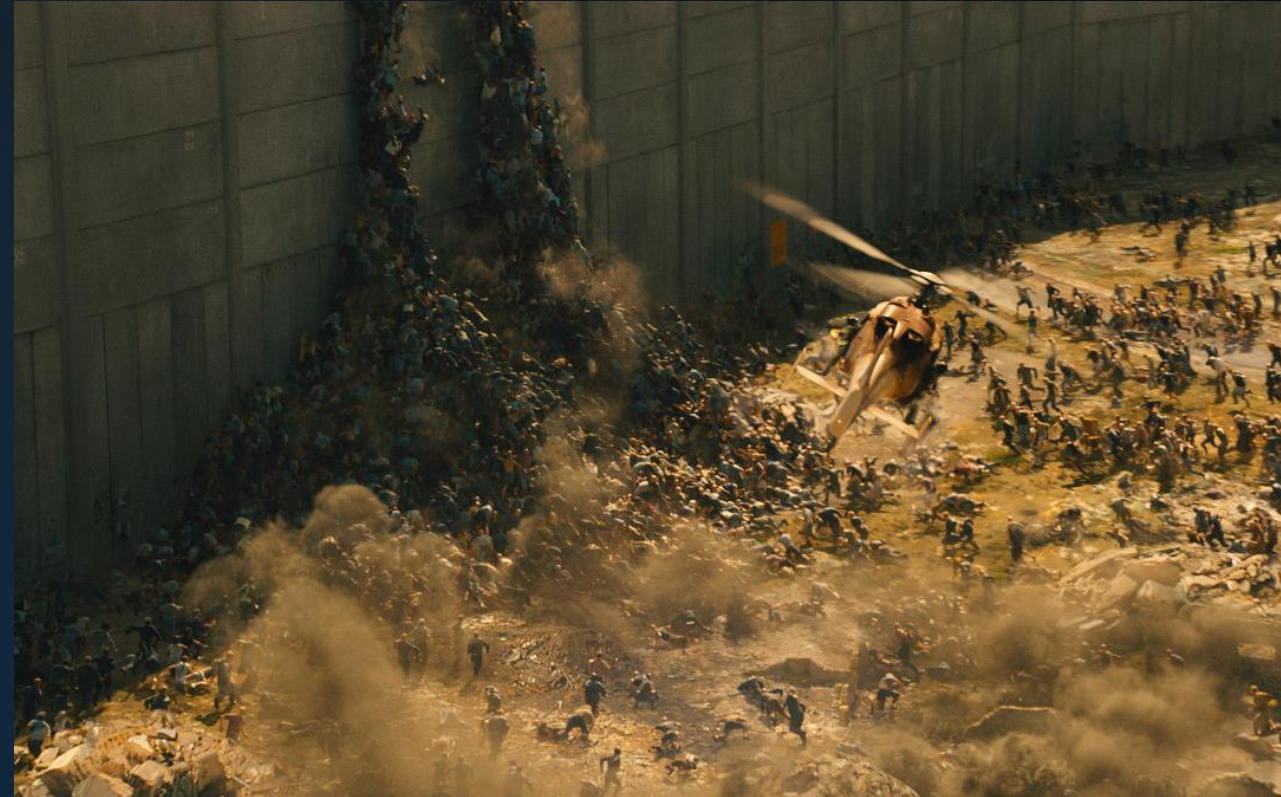
- Problem Analysis
- Early Work
- BVH Up Close



Analysis



Just Cause 3
Avalanche Studios, 2015



World War Z
Paramount Pictures, 2013



Analysis

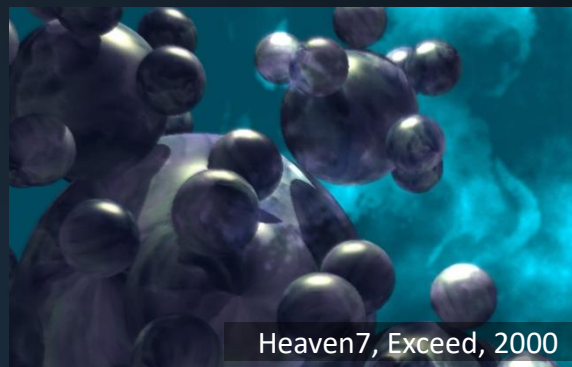
Characteristics

Rasterization:

- Games
- Fast
- Realistic
- Consumer hardware

Ray Tracing:

- Movies
- Slow
- Very Realistic
- Supercomputers



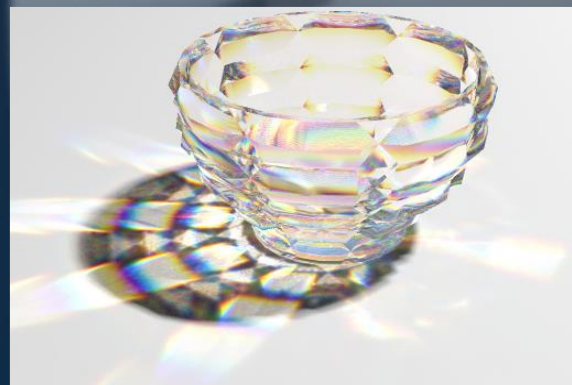
Heaven7, Exceed, 2000



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:

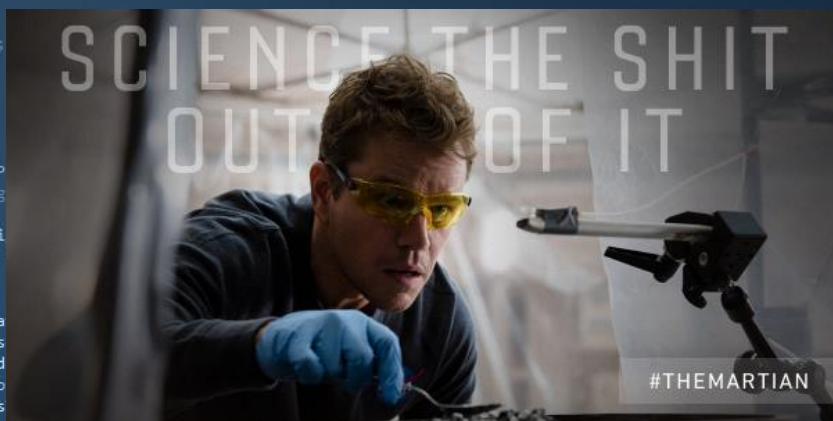
- Pixels
- Primitives
- Light sources
- Path segments

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.

```

ics
& (depth < MAXDEPTH)
{
    if ( ! inside ) return 0;
    nt = nt / nc; ddn = ddn * ddn;
    cos2t = 1.0f - nnt * ddn;
    if ( cos2t < 0 ) cos2t = 0;
    D, N );
    if ( ! cos2t ) return 0;
    float a = nt - nc, b = nt + nc;
    float r0 = 0, r1 = 1, r2 = 1;
    float Tr = 1 - (R0 + (1 - R0) * cos2t);
    float R = (D * nnt - N * (ddn * cos2t));
    if ( R < Tr )
    {
        E * diffuse;
        = true;
    }
    else
    {
        refl + refr)) && (depth < MAXDEPTH)
    {
        D, N );
        refl * E * diffuse;
        = true;
    }
}

MAXDEPTH)
{
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, closely following
    if ( ! survive ) return 0;
    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
    survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}

```



Today's Agenda:

- Problem Analysis
- Early Work
- BVH Up Close



Early Work

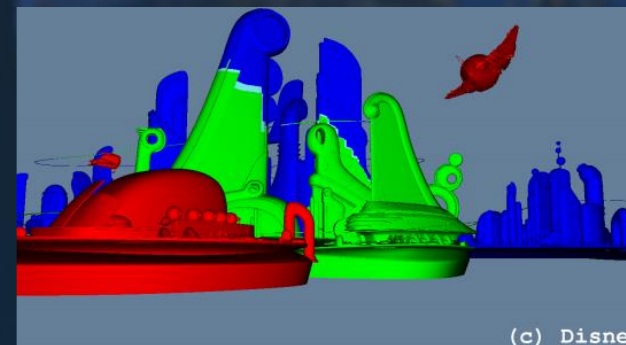
Complex Primitives

More expressive than a triangle:

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

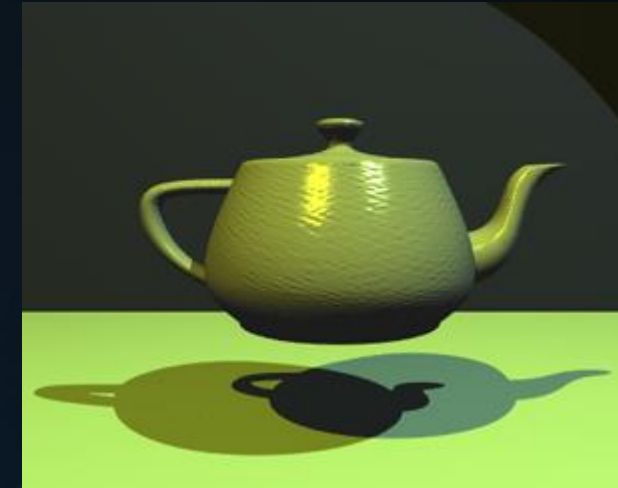


(c) Disney



(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

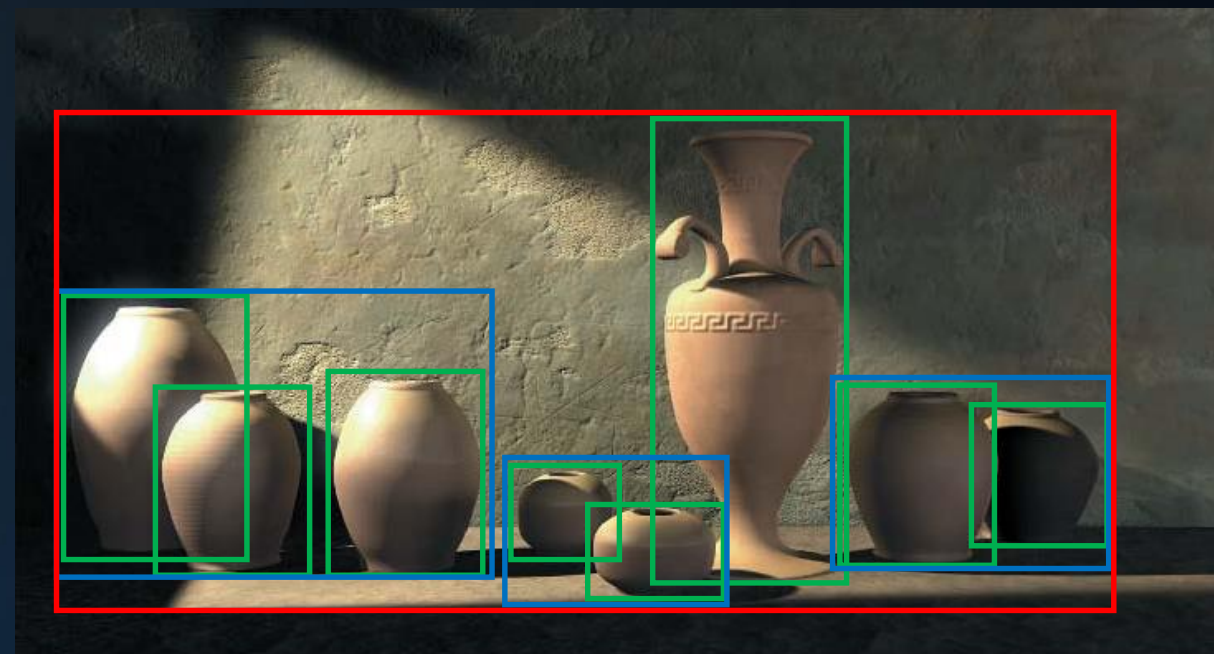
Rubin & Whitted*

“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

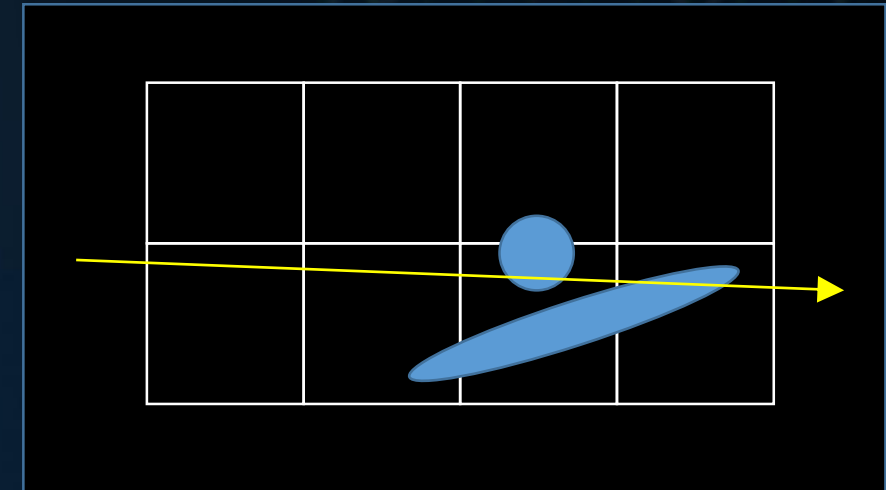
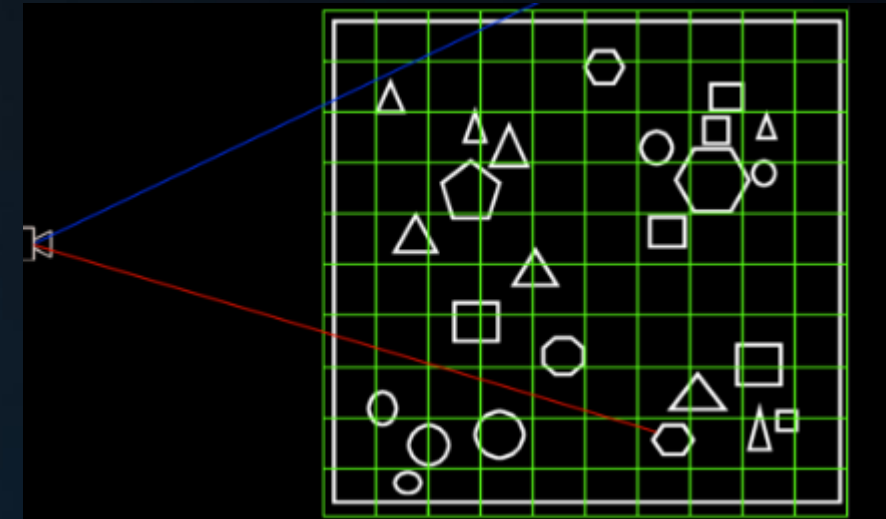
Amanatides & Woo*

“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”

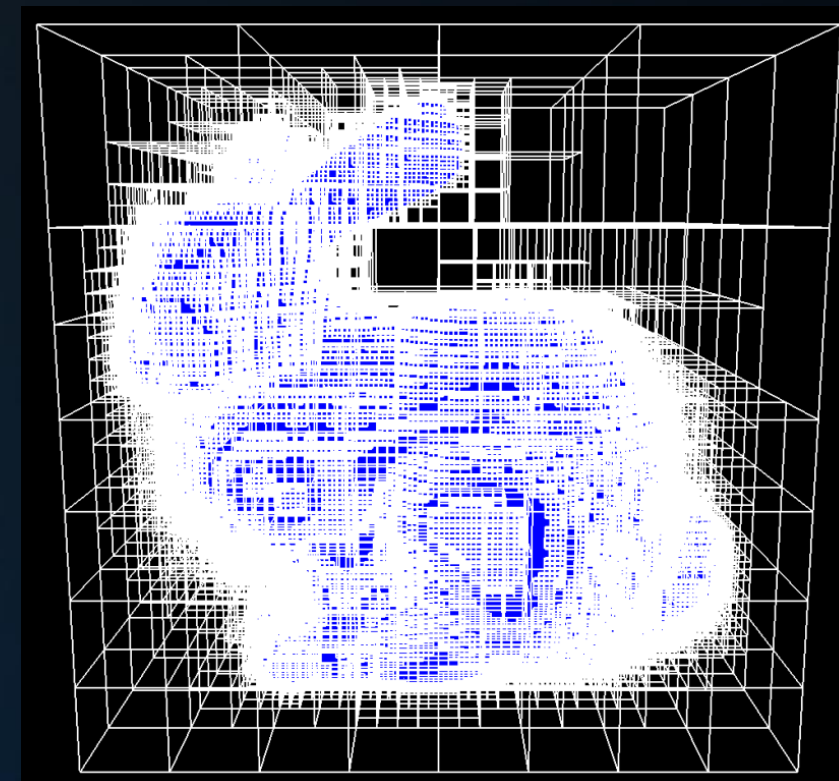
Like the grid, octrees can be automatically generated.

Advantages over grids:

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

Disadvantage compared to grids:

- Expensive traversal steps.



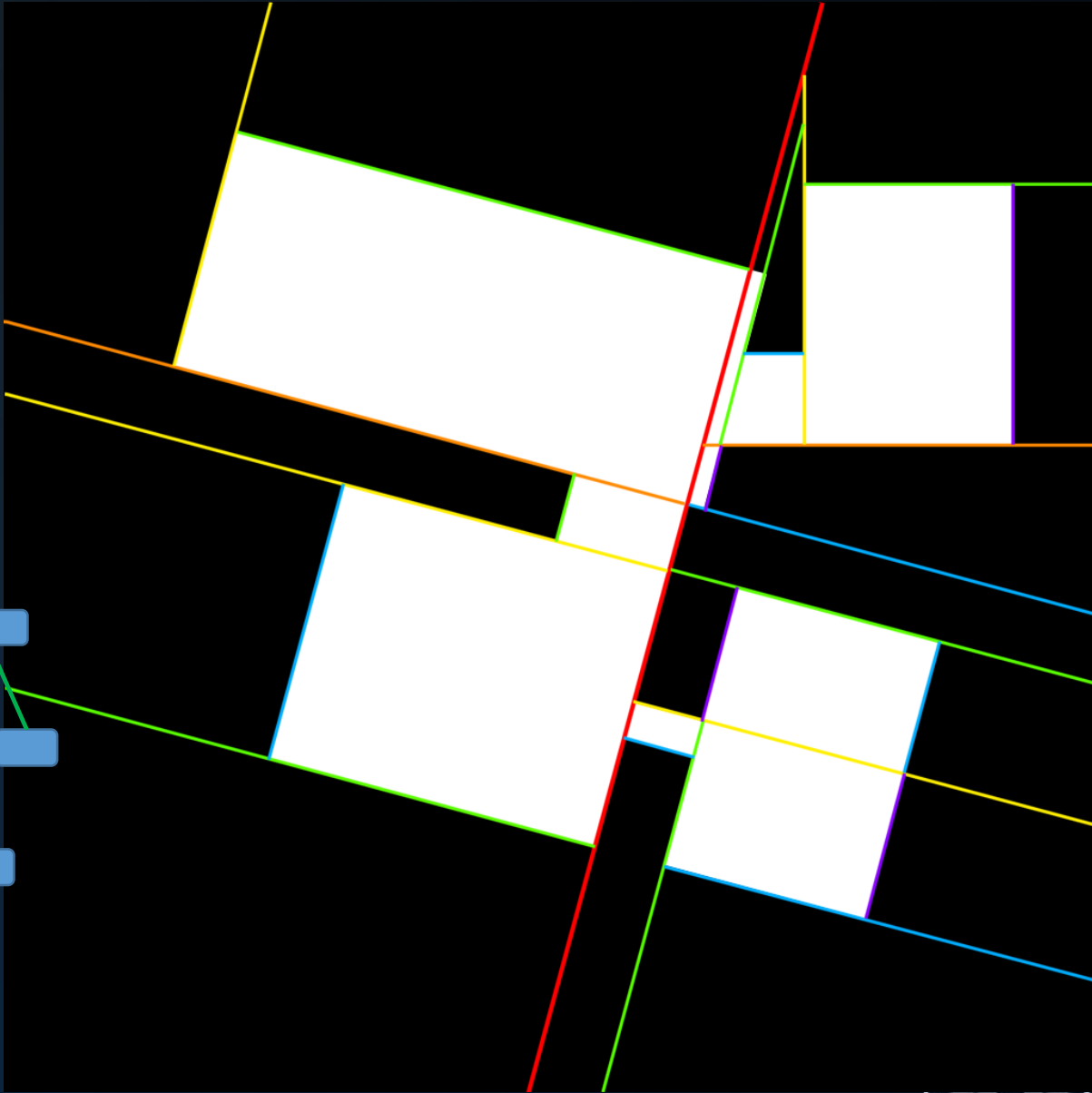
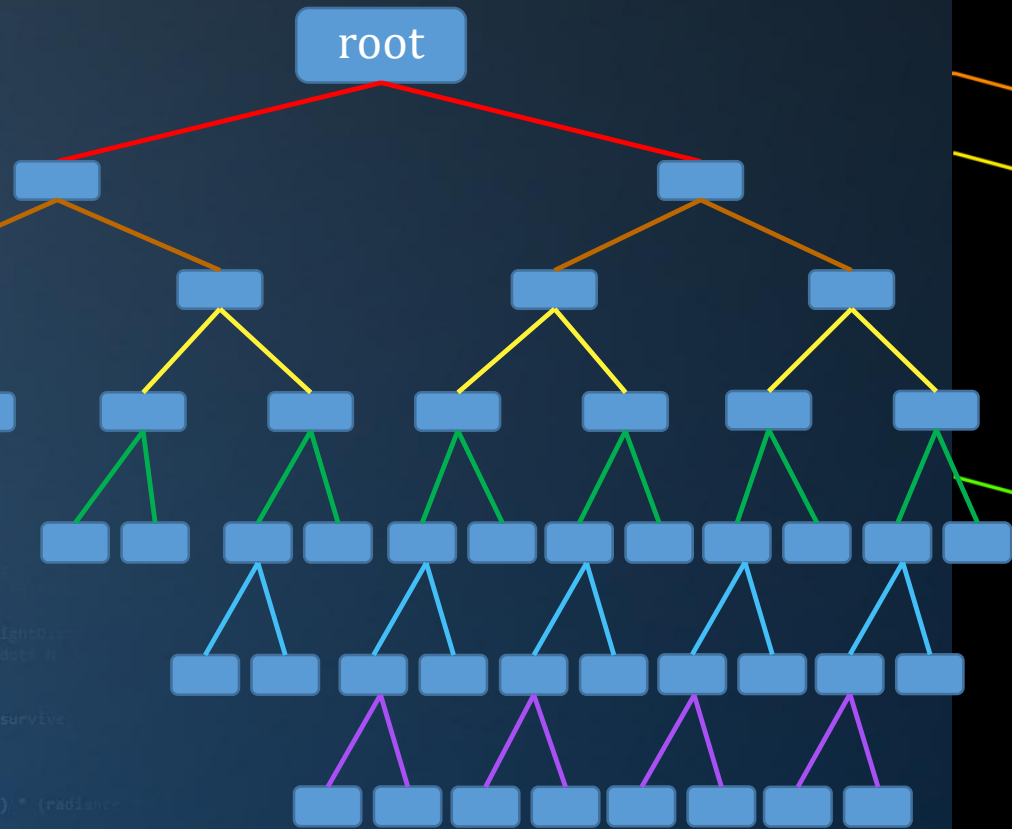
*: A. S. Glassner. Space Subdivision for Fast Ray Tracing. IEEE Computer Graphics and Applications, 4:15–22, 1984.



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));

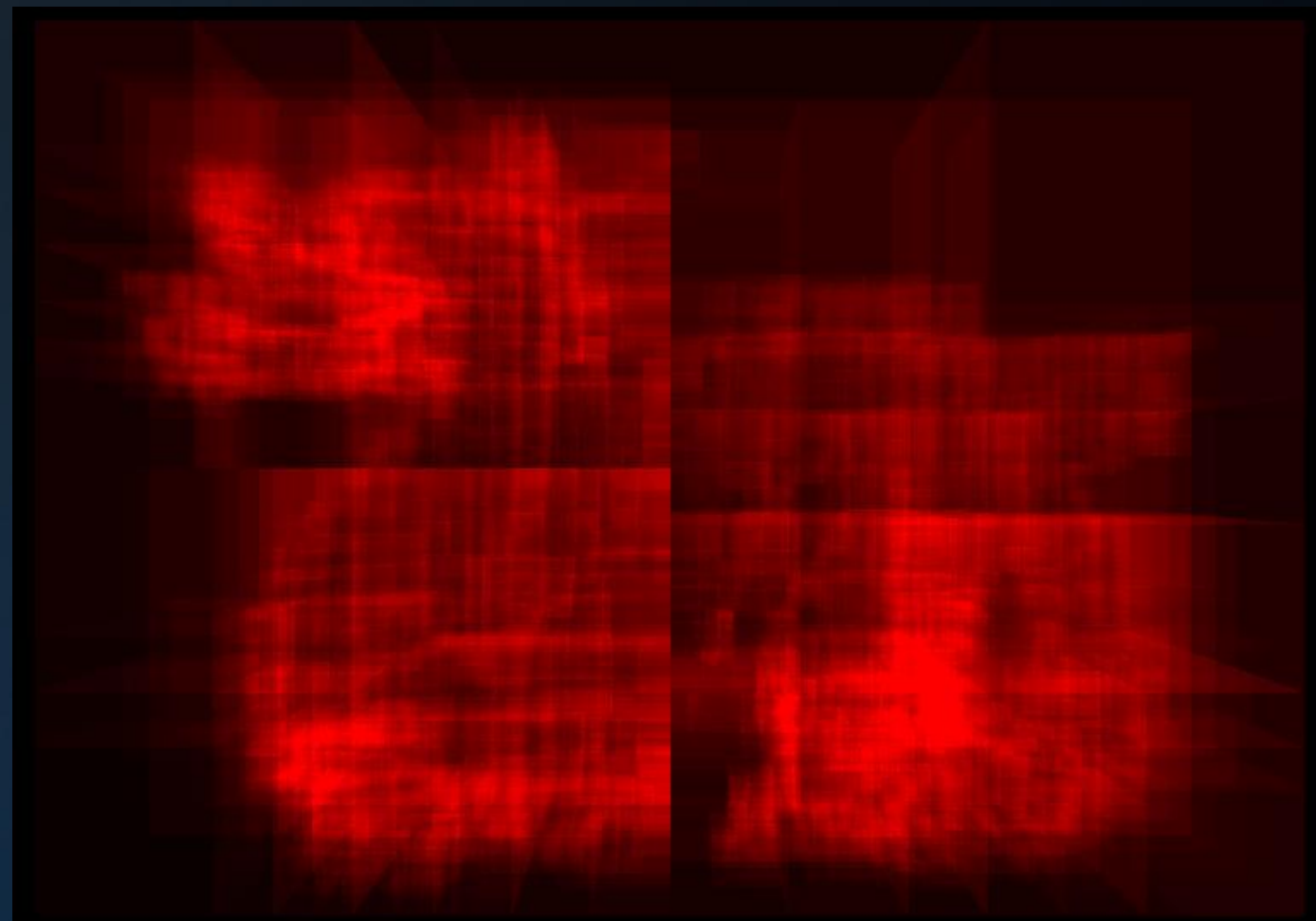
E * diffuse;
= true;

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
(survive)
{
    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.



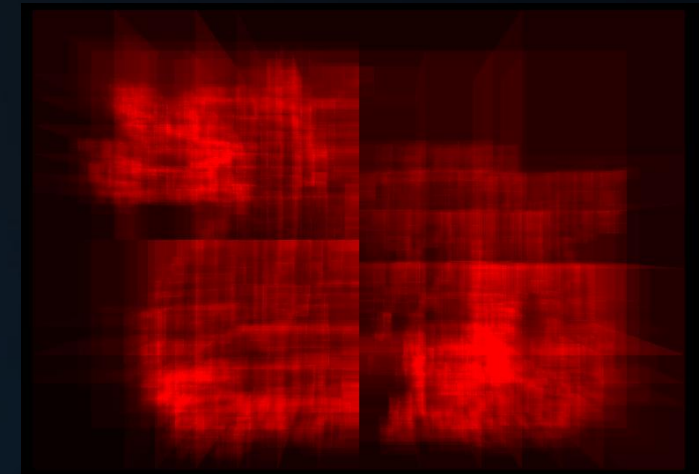
Early Work

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)

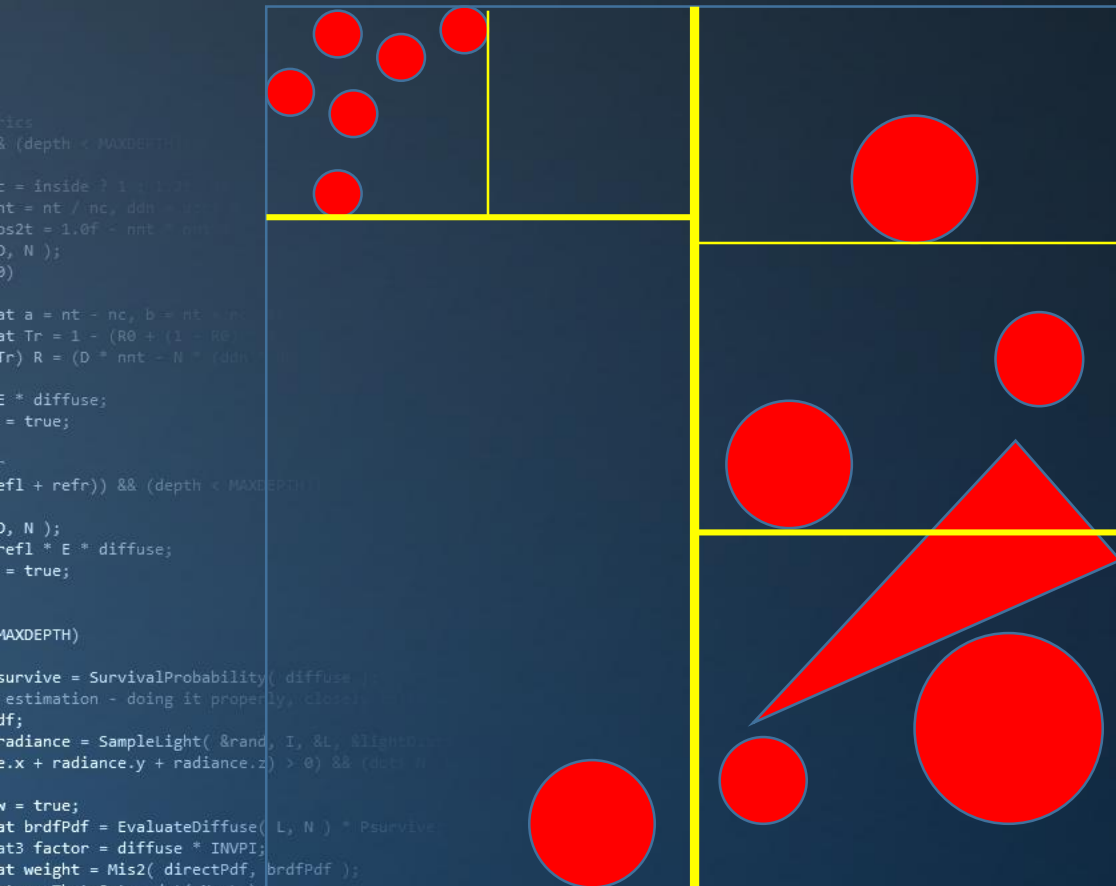
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

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



Early Work

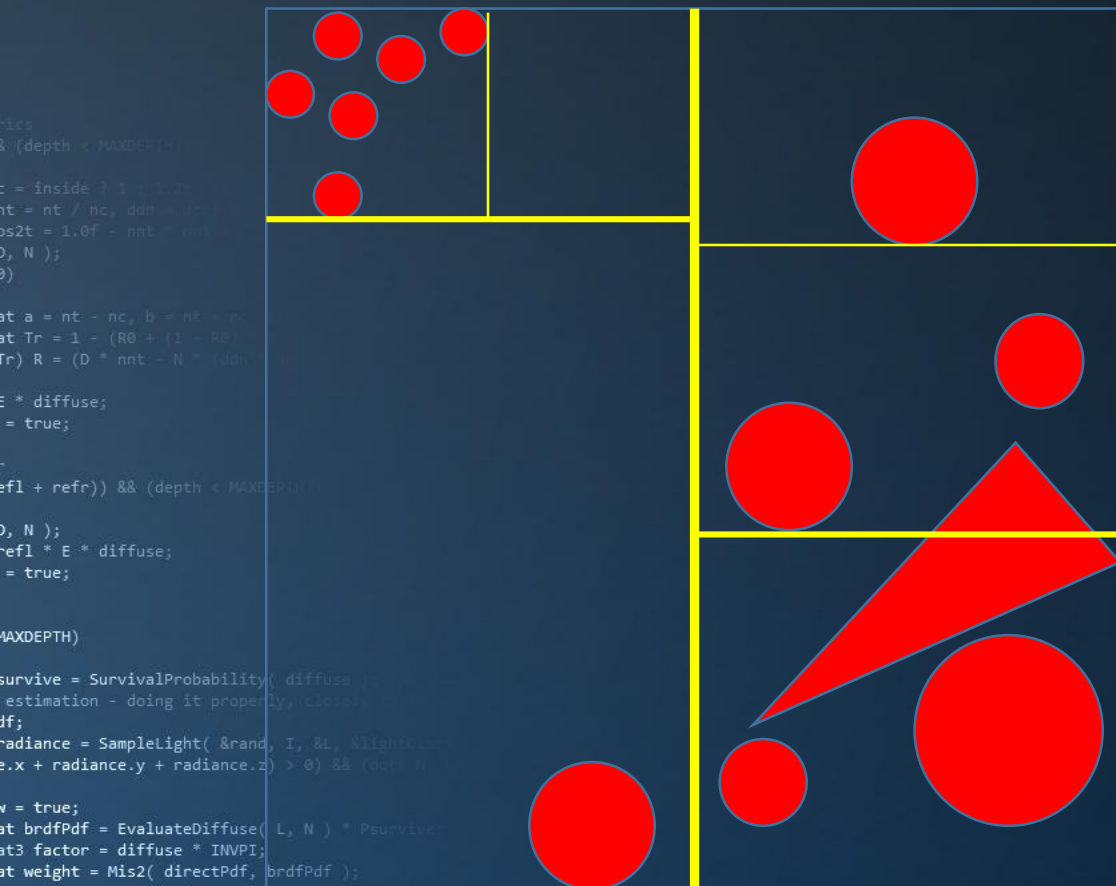


```
function Build( triangles  $T$ , voxel  $V$  )
{
    if (Terminate(  $T$ ,  $V$  )) return new LeafNode(  $T$  )
     $p$  = FindPlane(  $T$ ,  $V$  )
     $V_L, V_R$  = Split  $V$  with  $p$ 
     $T_L = \{t \in T \mid (t \cap V_L) \neq \emptyset\}$ 
     $T_R = \{t \in T \mid (t \cap V_R) \neq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

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

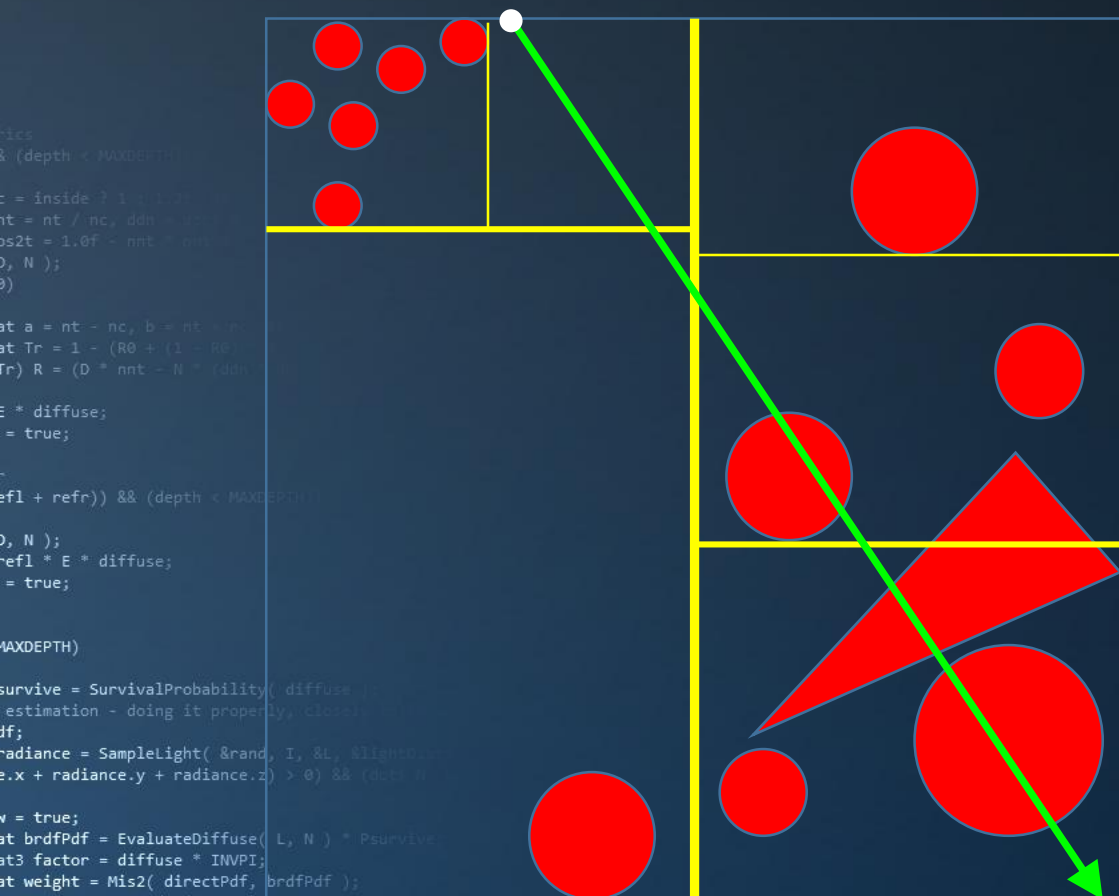
empty space reduces probability of having to intersect primitives

- Optimal split plane position / axis

good solutions exist – will be discussed later.



Early Work



Traversal*

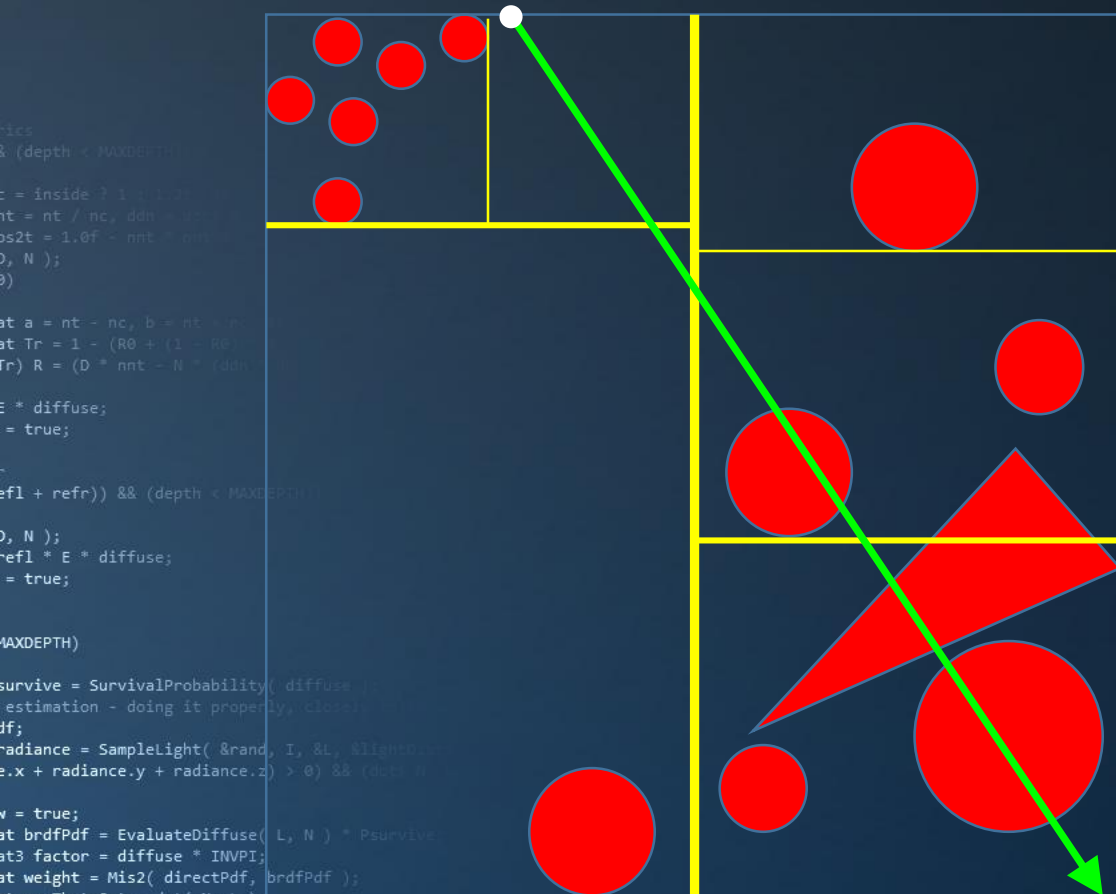
1. Find the point P where the ray enters the voxel
2. Determine which leaf node contains this point
3. Intersect the ray with the primitives in the leaf
- If intersections are found:
 - Determine the closest intersection
 - If the intersection is inside the voxel: done
4. Determine the point B where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

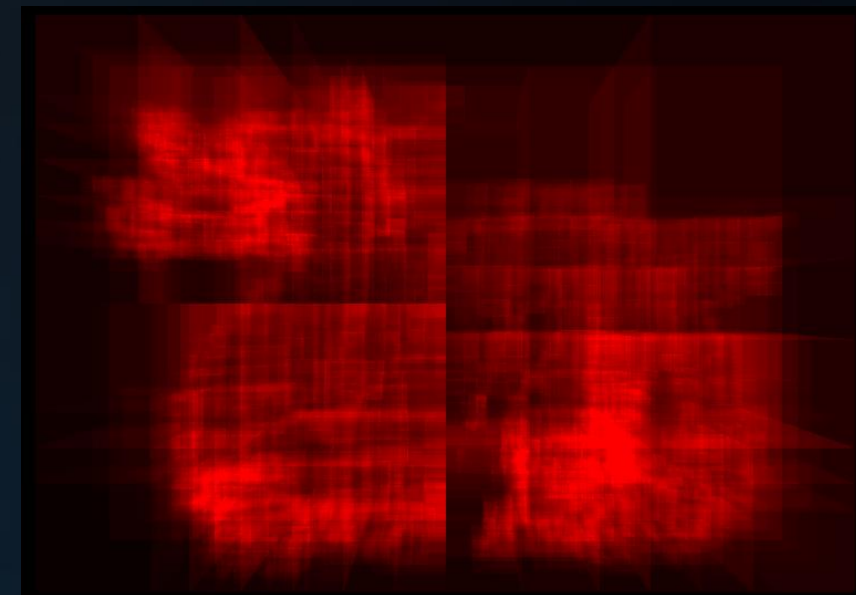
Traversing a kD-tree is done in a strict order.

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 );
    if ( D < 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 )
        {
            E * diffuse;
            = true;
        }
        else
        {
            refl + refr)) && (depth < MAXDEPTH)
            {
                D, N );
                refl * E * diffuse;
                = true;
            }
        }
    }
    if ( depth < MAXDEPTH )
    {
        survive = SurvivalProbability( diffuse );
        estimation - doing it properly, closely following Small's
        if ( survive )
        {
            radiance = SampleLight( &rand, I, &L, &align );
            radiance.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.x + radiance.y + radiance.z);
            }
        }
        random walk - done properly, closely following Small's
        survive)
    {
        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

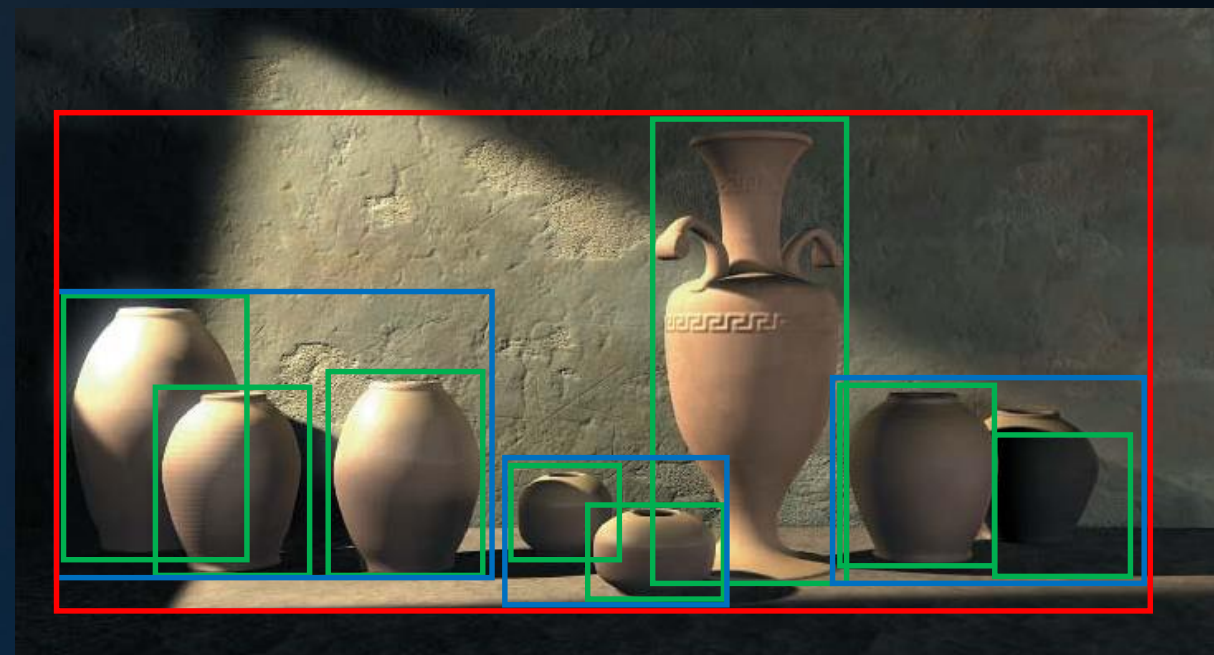
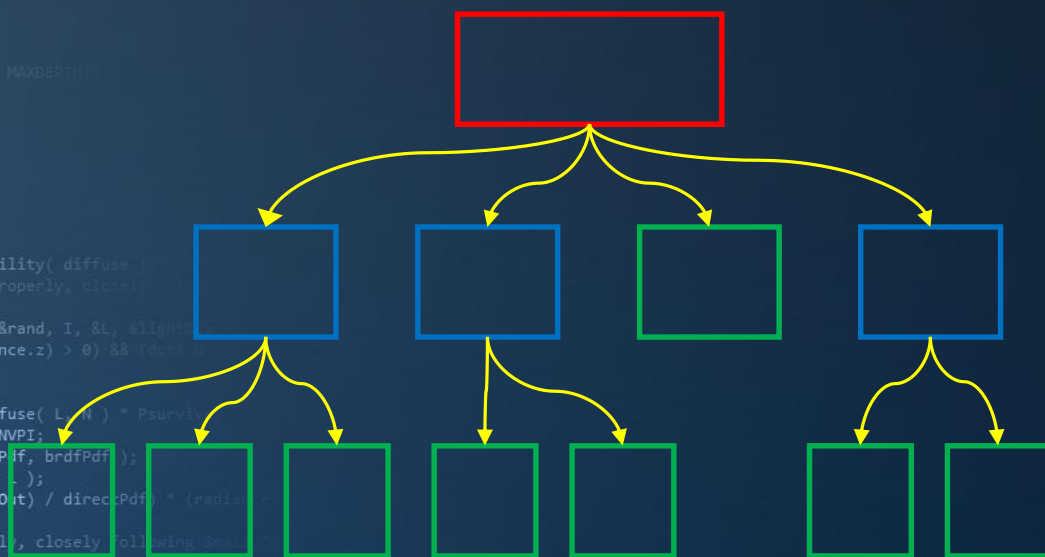
Automatic Construction of Bounding Volume Hierarchies

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;
        if (cos2t < 0) cos2t = 0;
        D, N );
        R = (D * nnt - N * (ddn * cos2t));
        E * diffuse;
        = true;
        defl + refr)) && (depth < MAXDEPTH)
        {
            D, N );
            refl * E * diffuse;
            = true;
        }
    }
    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;
  
```

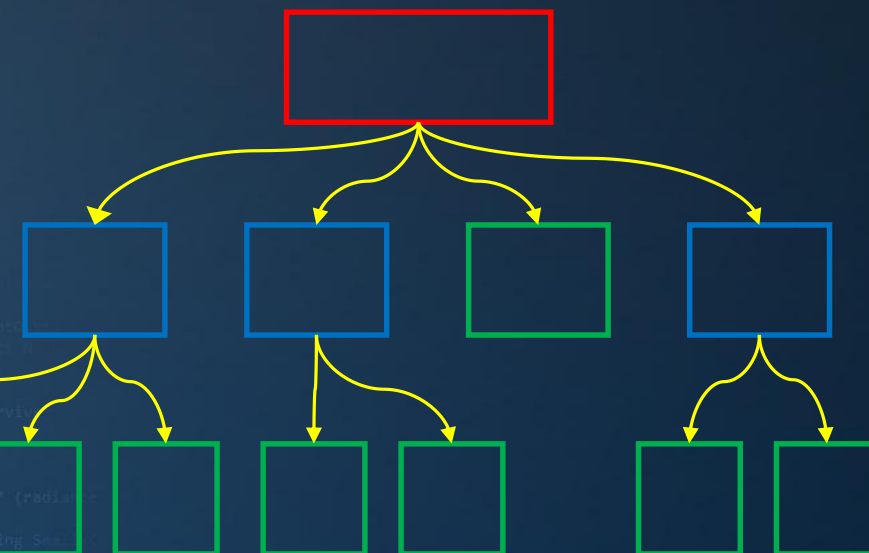


BVH

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;
};
```

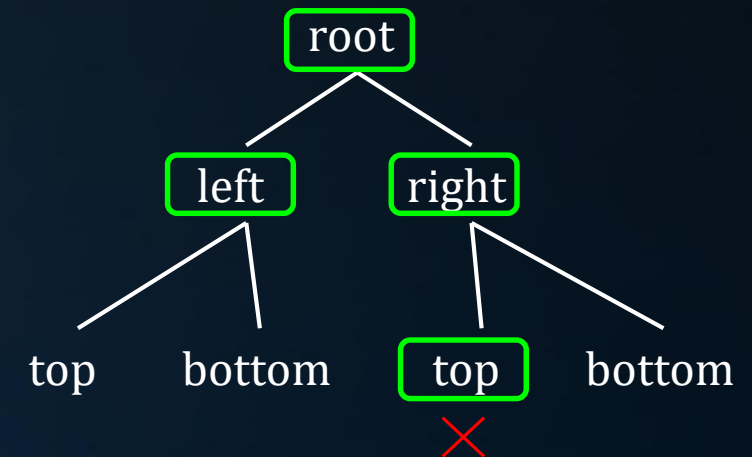
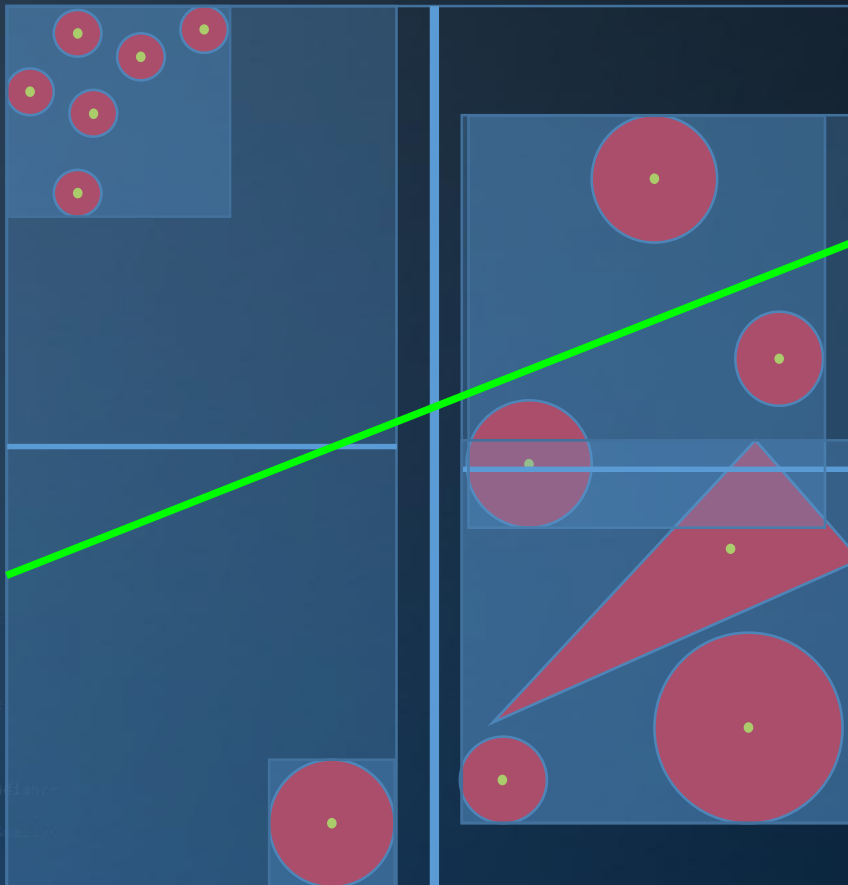


BVH

Automatic Construction of Bounding Volume Hierarchies

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0.25)
    {
        nt = nt / nc; ddn = ddn * 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 * 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, &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
    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



- 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

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0.25)
    {
        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) * a);
at Tr) R = (D * nnt - N * (ddn * nc));

```

```

E * diffuse;
= true;

```

```

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

```

```

D, N );
refl * 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) && (oc
};

```

```

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
ive)

```

```

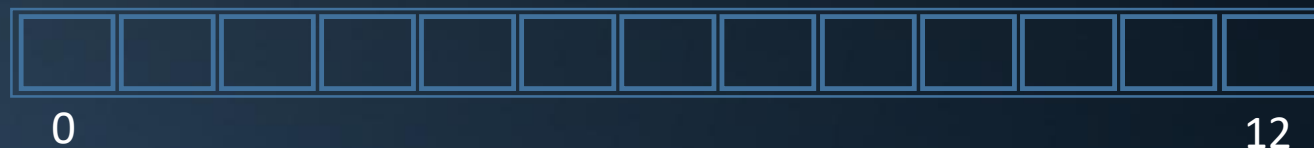
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;

```

```

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

```



```
struct BVHNode
```

```
{
```

```
    AABB bounds; // 24 bytes
```

```
    bool isLeaf; // 4 bytes
```

```
    BVHNode* left, *right; // 8 or 16 bytes
```

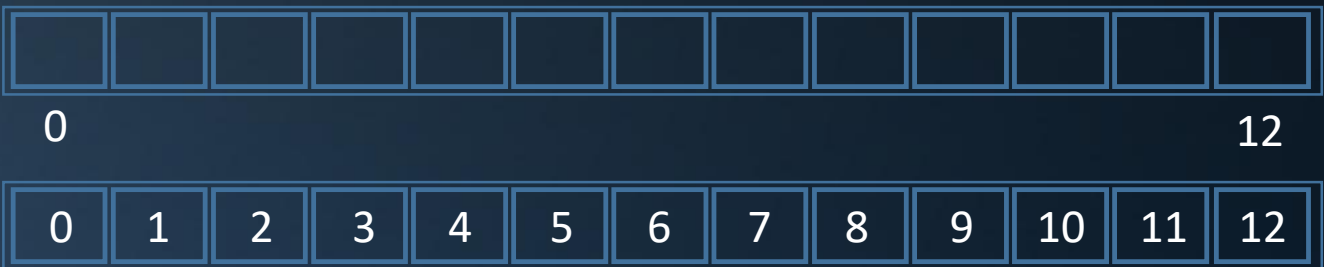
```
    Primitive** primList; // ? bytes
```

```
};
```



BVH

Automatic Construction of Bounding Volume Hierarchies



primitives

primitive indices

```
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 );
    refl * 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 5.4.2
    vive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = 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();

```

```

}

```

```

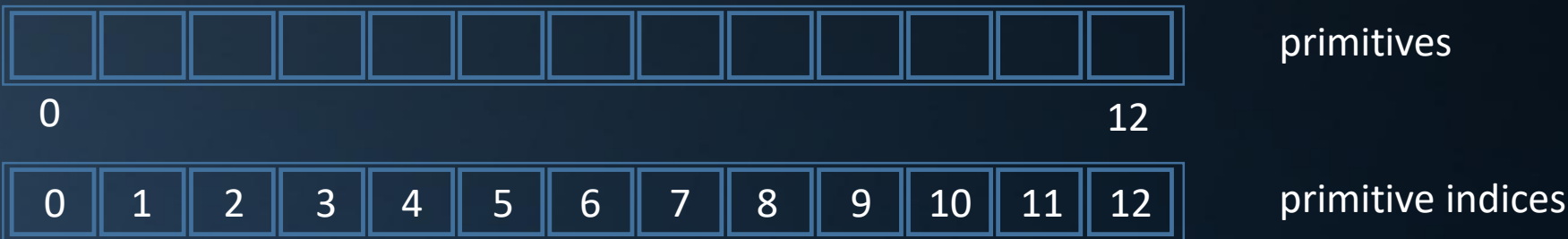
void BVHNode::Subdivide()
{
    if (count < 3) return;
    this->left = pool[poolPtr++];
    this->right = pool[poolPtr++];
    Partition();
    this->left->Subdivide();
    this->right->Subdivide();
    this->isLeaf = false;
}

```



BVH

Automatic Construction of Bounding Volume Hierarchies

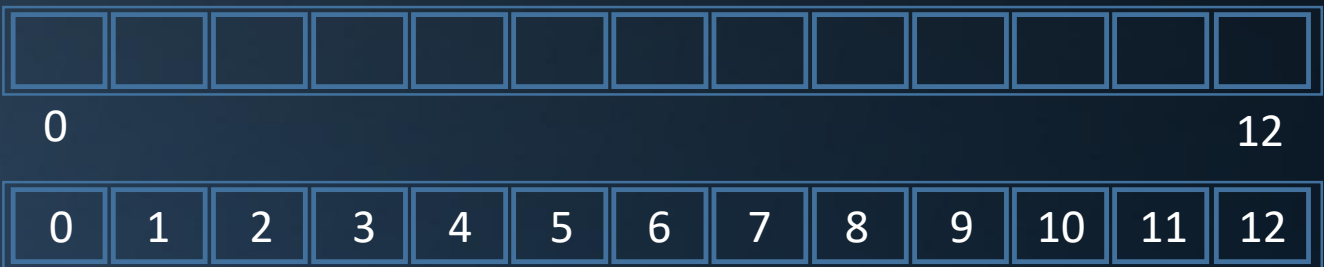


```
struct BVHNode
{
    AABB bounds;           // 24 bytes
    bool isLeaf;           // 4 bytes
    int left, right;       // 8 bytes
    int first, count;      // 8 bytes, total 44 bytes
};
```



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
};
```

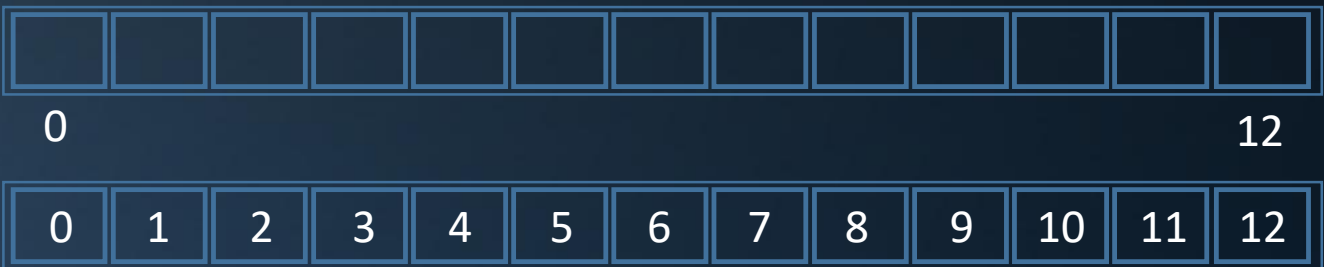


BVH nodes



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'.



BVH

BVH Traversal

```
void Sphere::intersect(const Ray & r, float & t, Vec & p, Vec & n) const {
    float u = 1.0f - r.d.x*r.d.x - r.d.y*r.d.y - r.d.z*r.d.z;
    float v = t*t*u;
    Vec p1 = r.o + r.d*t;
    Vec n1 = p1 - p;
    float n1n1 = n1.x*n1.x + n1.y*n1.y + n1.z*n1.z;
    float c = 1.0f - n1n1/v;
    Vec n2 = n1*c;
    float n2n2 = n2.x*n2.x + n2.y*n2.y + n2.z*n2.z;
    float ddn = n2n2/v;
    Vec n3 = n2*ddn;
    Vec p2 = p1 - n3;
    Vec n4 = p2 - p;
    float n4n4 = n4.x*n4.x + n4.y*n4.y + n4.z*n4.z;
    float f = 1.0f - n4n4/v;
    Vec n5 = n4*f;
    float n5n5 = n5.x*n5.x + n5.y*n5.y + n5.z*n5.z;
    float ddn2 = ddn*n5n5;
    if (ddn2 < 0.0f) return;
    Vec n6 = n5*ddn;
    Vec p3 = p1 - n6;
    Vec n7 = p3 - p;
    float n7n7 = n7.x*n7.x + n7.y*n7.y + n7.z*n7.z;
    float f2 = 1.0f - n7n7/v;
    Vec n8 = n7*f2;
    float n8n8 = n8.x*n8.x + n8.y*n8.y + n8.z*n8.z;
    float ddn3 = ddn*n8n8;
    if (ddn3 < 0.0f) return;
    Vec n9 = n8*ddn;
    Vec p4 = p1 - n9;
    Vec n10 = p4 - p;
    float n10n10 = n10.x*n10.x + n10.y*n10.y + n10.z*n10.z;
    float f3 = 1.0f - n10n10/v;
    Vec n11 = n10*f3;
    float n11n11 = n11.x*n11.x + n11.y*n11.y + n11.z*n11.z;
    float ddn4 = ddn*n11n11;
    if (ddn4 < 0.0f) return;
    Vec n12 = n11*ddn;
    Vec p5 = p1 - n12;
    Vec n13 = p5 - p;
    float n13n13 = n13.x*n13.x + n13.y*n13.y + n13.z*n13.z;
    float f4 = 1.0f - n13n13/v;
    Vec n14 = n13*f4;
    float n14n14 = n14.x*n14.x + n14.y*n14.y + n14.z*n14.z;
    float ddn5 = ddn*n14n14;
    if (ddn5 < 0.0f) return;
    Vec n15 = n14*ddn;
    Vec p6 = p1 - n15;
    Vec n16 = p6 - p;
    float n16n16 = n16.x*n16.x + n16.y*n16.y + n16.z*n16.z;
    float f5 = 1.0f - n16n16/v;
    Vec n17 = n16*f5;
    float n17n17 = n17.x*n17.x + n17.y*n17.y + n17.z*n17.z;
    float ddn6 = ddn*n17n17;
    if (ddn6 < 0.0f) return;
    Vec n18 = n17*ddn;
    Vec p7 = p1 - n18;
    Vec n19 = p7 - p;
    float n19n19 = n19.x*n19.x + n19.y*n19.y + n19.z*n19.z;
    float f6 = 1.0f - n19n19/v;
    Vec n20 = n19*f6;
    float n20n20 = n20.x*n20.x + n20.y*n20.y + n20.z*n20.z;
    float ddn7 = ddn*n20n20;
    if (ddn7 < 0.0f) return;
    Vec n21 = n20*ddn;
    Vec p8 = p1 - n21;
    Vec n22 = p8 - p;
    float n22n22 = n22.x*n22.x + n22.y*n22.y + n22.z*n22.z;
    float f7 = 1.0f - n22n22/v;
    Vec n23 = n22*f7;
    float n23n23 = n23.x*n23.x + n23.y*n23.y + n23.z*n23.z;
    float ddn8 = ddn*n23n23;
    if (ddn8 < 0.0f) return;
    Vec n24 = n23*ddn;
    Vec p9 = p1 - n24;
    Vec n25 = p9 - p;
    float n25n25 = n25.x*n25.x + n25.y*n25.y + n25.z*n25.z;
    float f8 = 1.0f - n25n25/v;
    Vec n26 = n25*f8;
    float n26n26 = n26.x*n26.x + n26.y*n26.y + n26.z*n26.z;
    float ddn9 = ddn*n26n26;
    if (ddn9 < 0.0f) return;
    Vec n27 = n26*ddn;
    Vec p10 = p1 - n27;
    Vec n28 = p10 - p;
    float n28n28 = n28.x*n28.x + n28.y*n28.y + n28.z*n28.z;
    float f9 = 1.0f - n28n28/v;
    Vec n29 = n28*f9;
    float n29n29 = n29.x*n29.x + n29.y*n29.y + n29.z*n29.z;
    float ddn10 = ddn*n29n29;
    if (ddn10 < 0.0f) return;
    Vec n30 = n29*ddn;
    Vec p11 = p1 - n30;
    Vec n31 = p11 - p;
    float n31n31 = n31.x*n31.x + n31.y*n31.y + n31.z*n31.z;
    float f10 = 1.0f - n31n31/v;
    Vec n32 = n31*f10;
    float n32n32 = n32.x*n32.x + n32.y*n32.y + n32.z*n32.z;
    float ddn11 = ddn*n32n32;
    if (ddn11 < 0.0f) return;
    Vec n33 = n32*ddn;
    Vec p12 = p1 - n33;
    Vec n34 = p12 - p;
    float n34n34 = n34.x*n34.x + n34.y*n34.y + n34.z*n34.z;
    float f11 = 1.0f - n34n34/v;
    Vec n35 = n34*f11;
    float n35n35 = n35.x*n35.x + n35.y*n35.y + n35.z*n35.z;
    float ddn12 = ddn*n35n35;
    if (ddn12 < 0.0f) return;
    Vec n36 = n35*ddn;
    Vec p13 = p1 - n36;
    Vec n37 = p13 - p;
    float n37n37 = n37.x*n37.x + n37.y*n37.y + n37.z*n37.z;
    float f12 = 1.0f - n37n37/v;
    Vec n38 = n37*f12;
    float n38n38 = n38.x*n38.x + n38.y*n38.y + n38.z*n38.z;
    float ddn13 = ddn*n38n38;
    if (ddn13 < 0.0f) return;
    Vec n39 = n38*ddn;
    Vec p14 = p1 - n39;
    Vec n40 = p14 - p;
    float n40n40 = n40.x*n40.x + n40.y*n40.y + n40.z*n40.z;
    float f13 = 1.0f - n40n40/v;
    Vec n41 = n40*f13;
    float n41n41 = n41.x*n41.x + n41.y*n41.y + n41.z*n41.z;
    float ddn14 = ddn*n41n41;
    if (ddn14 < 0.0f) return;
    Vec n42 = n41*ddn;
    Vec p15 = p1 - n42;
    Vec n43 = p15 - p;
    float n43n43 = n43.x*n43.x + n43.y*n43.y + n43.z*n43.z;
    float f14 = 1.0f - n43n43/v;
    Vec n44 = n43*f14;
    float n44n44 = n44.x*n44.x + n44.y*n44.y + n44.z*n44.z;
    float ddn15 = ddn*n44n44;
    if (ddn15 < 0.0f) return;
    Vec n45 = n44*ddn;
    Vec p16 = p1 - n45;
    Vec n46 = p16 - p;
    float n46n46 = n46.x*n46.x + n46.y*n46.y + n46.z*n46.z;
    float f15 = 1.0f - n46n46/v;
    Vec n47 = n46*f15;
    float n47n47 = n47.x*n47.x + n47.y*n47.y + n47.z*n47.z;
    float ddn16 = ddn*n47n47;
    if (ddn16 < 0.0f) return;
    Vec n48 = n47*ddn;
    Vec p17 = p1 - n48;
    Vec n49 = p17 - p;
    float n49n49 = n49.x*n49.x + n49.y*n49.y + n49.z*n49.z;
    float f16 = 1.0f - n49n49/v;
    Vec n50 = n49*f16;
    float n50n50 = n50.x*n50.x + n50.y*n50.y + n50.z*n50.z;
    float ddn17 = ddn*n50n50;
    if (ddn17 < 0.0f) return;
    Vec n51 = n50*ddn;
    Vec p18 = p1 - n51;
    Vec n52 = p18 - p;
    float n52n52 = n52.x*n52.x + n52.y*n52.y + n52.z*n52.z;
    float f17 = 1.0f - n52n52/v;
    Vec n53 = n52*f17;
    float n53n53 = n53.x*n53.x + n53.y*n53.y + n53.z*n53.z;
    float ddn18 = ddn*n53n53;
    if (ddn18 < 0.0f) return;
    Vec n54 = n53*ddn;
    Vec p19 = p1 - n54;
    Vec n55 = p19 - p;
    float n55n55 = n55.x*n55.x + n55.y*n55.y + n55.z*n55.z;
    float f18 = 1.0f - n55n55/v;
    Vec n56 = n55*f18;
    float n56n56 = n56.x*n56.x + n56.y*n56.y + n56.z*n56.z;
    float ddn19 = ddn*n56n56;
    if (ddn19 < 0.0f) return;
    Vec n57 = n56*ddn;
    Vec p20 = p1 - n57;
    Vec n58 = p20 - p;
    float n58n58 = n58.x*n58.x + n58.y*n58.y + n58.z*n58.z;
    float f19 = 1.0f - n58n58/v;
    Vec n59 = n58*f19;
    float n59n59 = n59.x*n59.x + n59.y*n59.y + n59.z*n59.z;
    float ddn20 = ddn*n59n59;
    if (ddn20 < 0.0f) return;
    Vec n60 = n59*ddn;
    Vec p21 = p1 - n60;
    Vec n61 = p21 - p;
    float n61n61 = n61.x*n61.x + n61.y*n61.y + n61.z*n61.z;
    float f20 = 1.0f - n61n61/v;
    Vec n62 = n61*f20;
    float n62n62 = n62.x*n62.x + n62.y*n62.y + n62.z*n62.z;
    float ddn21 = ddn*n62n62;
    if (ddn21 < 0.0f) return;
    Vec n63 = n62*ddn;
    Vec p22 = p1 - n63;
    Vec n64 = p22 - p;
    float n64n64 = n64.x*n64.x + n64.y*n64.y + n64.z*n64.z;
    float f21 = 1.0f - n64n64/v;
    Vec n65 = n64*f21;
    float n65n65 = n65.x*n65.x + n65.y*n65.y + n65.z*n65.z;
    float ddn22 = ddn*n65n65;
    if (ddn22 < 0.0f) return;
    Vec n66 = n65*ddn;
    Vec p23 = p1 - n66;
    Vec n67 = p23 - p;
    float n67n67 = n67.x*n67.x + n67.y*n67.y + n67.z*n67.z;
    float f22 = 1.0f - n67n67/v;
    Vec n68 = n67*f22;
    float n68n68 = n68.x*n68.x + n68.y*n68.y + n68.z*n68.z;
    float ddn23 = ddn*n68n68;
    if (ddn23 < 0.0f) return;
    Vec n69 = n68*ddn;
    Vec p24 = p1 - n69;
    Vec n70 = p24 - p;
    float n70n70 = n70.x*n70.x + n70.y*n70.y + n70.z*n70.z;
    float f23 = 1.0f - n70n70/v;
    Vec n71 = n70*f23;
    float n71n71 = n71.x*n71.x + n71.y*n71.y + n71.z*n71.z;
    float ddn24 = ddn*n71n71;
    if (ddn24 < 0.0f) return;
    Vec n72 = n71*ddn;
    Vec p25 = p1 - n72;
    Vec n73 = p25 - p;
    float n73n73 = n73.x*n73.x + n73.y*n73.y + n73.z*n73.z;
    float f24 = 1.0f - n73n73/v;
    Vec n74 = n73*f24;
    float n74n74 = n74.x*n74.x + n74.y*n74.y + n74.z*n74.z;
    float ddn25 = ddn*n74n74;
    if (ddn25 < 0.0f) return;
    Vec n75 = n74*ddn;
    Vec p26 = p1 - n75;
    Vec n76 = p26 - p;
    float n76n76 = n76.x*n76.x + n76.y*n76.y + n76.z*n76.z;
    float f25 = 1.0f - n76n76/v;
    Vec n77 = n76*f25;
    float n77n77 = n77.x*n77.x + n77.y*n77.y + n77.z*n77.z;
    float ddn26 = ddn*n77n77;
    if (ddn26 < 0.0f) return;
    Vec n78 = n77*ddn;
    Vec p27 = p1 - n78;
    Vec n79 = p27 - p;
    float n79n79 = n79.x*n79.x + n79.y*n79.y + n79.z*n79.z;
    float f26 = 1.0f - n79n79/v;
    Vec n80 = n79*f26;
    float n80n80 = n80.x*n80.x + n80.y*n80.y + n80.z*n80.z;
    float ddn27 = ddn*n80n80;
    if (ddn27 < 0.0f) return;
    Vec n81 = n80*ddn;
    Vec p28 = p1 - n81;
    Vec n82 = p28 - p;
    float n82n82 = n82.x*n82.x + n82.y*n82.y + n82.z*n82.z;
    float f27 = 1.0f - n82n82/v;
    Vec n83 = n82*f27;
    float n83n83 = n83.x*n83.x + n83.y*n83.y + n83.z*n83.z;
    float ddn28 = ddn*n83n83;
    if (ddn28 < 0.0f) return;
    Vec n84 = n83*ddn;
    Vec p29 = p1 - n84;
    Vec n85 = p29 - p;
    float n85n85 = n85.x*n85.x + n85.y*n85.y + n85.z*n85.z;
    float f28 = 1.0f - n85n85/v;
    Vec n86 = n85*f28;
    float n86n86 = n86.x*n86.x + n86.y*n86.y + n86.z*n86.z;
    float ddn29 = ddn*n86n86;
    if (ddn29 < 0.0f) return;
    Vec n87 = n86*ddn;
    Vec p30 = p1 - n87;
    Vec n88 = p30 - p;
    float n88n88 = n88.x*n88.x + n88.y*n88.y + n88.z*n88.z;
    float f29 = 1.0f - n88n88/v;
    Vec n89 = n88*f29;
    float n89n89 = n89.x*n89.x + n89.y*n89.y + n89.z*n89.z;
    float ddn30 = ddn*n89n89;
    if (ddn30 < 0.0f) return;
    Vec n90 = n89*ddn;
    Vec p31 = p1 - n90;
    Vec n91 = p31 - p;
    float n91n91 = n91.x*n91.x + n91.y*n91.y + n91.z*n91.z;
    float f30 = 1.0f - n91n91/v;
    Vec n92 = n91*f30;
    float n92n92 = n92.x*n92.x + n92.y*n92.y + n92.z*n92.z;
    float ddn31 = ddn*n92n92;
    if (ddn31 < 0.0f) return;
    Vec n93 = n92*ddn;
    Vec p32 = p1 - n93;
    Vec n94 = p32 - p;
    float n94n94 = n94.x*n94.x + n94.y*n94.y + n94.z*n94.z;
    float f31 = 1.0f - n94n94/v;
    Vec n95 = n94*f31;
    float n95n95 = n95.x*n95.x + n95.y*n95.y + n95.z*n95.z;
    float ddn32 = ddn*n95n95;
    if (ddn32 < 0.0f) return;
    Vec n96 = n95*ddn;
    Vec p33 = p1 - n96;
    Vec n97 = p33 - p;
    float n97n97 = n97.x*n97.x + n97.y*n97.y + n97.z*n97.z;
    float f32 = 1.0f - n97n97/v;
    Vec n98 = n97*f32;
    float n98n98 = n98.x*n98.x + n98.y*n98.y + n98.z*n98.z;
    float ddn33 = ddn*n98n98;
    if (ddn33 < 0.0f) return;
    Vec n99 = n98*ddn;
    Vec p34 = p1 - n99;
    Vec n100 = p34 - p;
    float n100n100 = n100.x*n100.x + n100.y*n100.y + n100.z*n100.z;
    float f33 = 1.0f - n100n100/v;
    Vec n101 = n100*f33;
    float n101n101 = n101.x*n101.x + n101.y*n101.y + n101.z*n101.z;
    float ddn34 = ddn*n101n101;
    if (ddn34 < 0.0f) return;
    Vec n102 = n101*ddn;
    Vec p35 = p1 - n102;
    Vec n103 = p35 - p;
    float n103n103 = n103.x*n103.x + n103.y*n103.y + n103.z*n103.z;
    float f34 = 1.0f - n103n103/v;
    Vec n104 = n103*f34;
    float n104n104 = n104.x*n104.x + n104.y*n104.y + n104.z*n104.z;
    float ddn35 = ddn*n104n104;
    if (ddn35 < 0.0f) return;
    Vec n105 = n104*ddn;
    Vec p36 = p1 - n105;
    Vec n106 = p36 - p;
    float n106n106 = n106.x*n106.x + n106.y*n106.y + n106.z*n106.z;
    float f35 = 1.0f - n106n106/v;
    Vec n107 = n106*f35;
    float n107n107 = n107.x*n107.x + n107.y*n107.y + n107.z*n107.z;
    float ddn36 = ddn*n107n107;
    if (ddn36 < 0.0f) return;
    Vec n108 = n107*ddn;
    Vec p37 = p1 - n108;
    Vec n109 = p37 - p;
    float n109n109 = n109.x*n109.x + n109.y*n109.y + n109.z*n109.z;
    float f36 = 1.0f - n109n109/v;
    Vec n110 = n109*f36;
    float n110n110 = n110.x*n110.x + n110.y*n110.y + n110.z*n110.z;
    float ddn37 = ddn*n110n110;
    if (ddn37 < 0.0f) return;
    Vec n111 = n110*ddn;
    Vec p38 = p1 - n111;
    Vec n112 = p38 - p;
    float n112n112 = n112.x*n112.x + n112.y*n112.y + n112.z*n112.z;
    float f37 = 1.0f - n112n112/v;
    Vec n113 = n112*f37;
    float n113n113 = n113.x*n113.x + n113.y*n113.y + n113.z*n113.z;
    float ddn38 = ddn*n113n113;
    if (ddn38 < 0.0f) return;
    Vec n114 = n113*ddn;
    Vec p39 = p1 - n114;
    Vec n115 = p39 - p;
    float n115n115 = n115.x*n115.x + n115.y*n115.y + n115.z*n115.z;
    float f38 = 1.0f - n115n115/v;
    Vec n116 = n115*f38;
    float n116n116 = n116.x*n116.x + n116.y*n116.y + n116.z*n116.z;
    float ddn39 = ddn*n116n116;
    if (ddn39 < 0.0f) return;
    Vec n117 = n116*ddn;
    Vec p40 = p1 - n117;
    Vec n118 = p40 - p;
    float n118n118 = n118.x*n118.x + n118.y*n118.y + n118.z*n118.z;
    float f39 = 1.0f - n118n118/v;
    Vec n119 = n118*f39;
    float n119n119 = n119.x*n119.x + n119.y*n119.y + n119.z*n119.z;
    float ddn40 = ddn*n119n119;
    if (ddn40 < 0.0f) return;
    Vec n120 = n119*ddn;
    Vec p41 = p1 - n120;
    Vec n121 = p41 - p;
    float n121n121 = n121.x*n121.x + n121.y*n121.y + n121.z*n121.z;
    float f40 = 1.0f - n121n121/v;
    Vec n122 = n121*f40;
    float n122n122 = n122.x*n122.x + n122.y*n122.y + n122.z*n122.z;
    float ddn41 = ddn*n122n122;
    if (ddn41 < 0.0f) return;
    Vec n123 = n122*ddn;
    Vec p42 = p1 - n123;
    Vec n124 = p42 - p;
    float n124n124 = n124.x*n124.x + n124.y*n124.y + n124.z*n124.z;
    float f41 = 1.0f - n124n124/v;
    Vec n125 = n124*f41;
    float n125n125 = n125.x*n125.x + n125.y*n125.y + n125.z*n125.z;
    float ddn42 = ddn*n125n125;
    if (ddn42 < 0.0f) return;
    Vec n126 = n125*ddn;
    Vec p43 = p1 - n126;
    Vec n127 = p43 - p;
    float n127n127 = n127.x*n127.x + n127.y*n127.y + n127.z*n127.z;
    float f42 = 1.0f - n127n127/v;
    Vec n128 = n127*f42;
    float n128n128 = n128.x*n128.x + n128.y*n128.y + n128.z*n128.z;
    float ddn43 = ddn*n128n128;
    if (ddn43 < 0.0f) return;
    Vec n129 = n128*ddn;
    Vec p44 = p1 - n129;
    Vec n130 = p44 - p;
    float n130n130 = n130.x*n130.x + n130.y*n130.y + n130.z*n130.z;
    float f43 = 1.0f - n130n130/v;
    Vec n131 = n130*f43;
    float n131n131 = n131.x*n131.x + n131.y*n131.y + n131.z*n131.z;
    float ddn44 = ddn*n131n131;
    if (ddn44 < 0.0f) return;
    Vec n132 = n131*ddn;
    Vec p45 = p1 - n132;
    Vec n133 = p45 - p;
    float n133n133 = n133.x*n133.x + n133.y*n133.y + n133.z*n133.z;
    float f44 = 1.0f - n133n133/v;
    Vec n134 = n133*f44;
    float n134n134 = n134.x*n134.x + n134.y*n134.y + n134.z*n134.z;
    float ddn45 = ddn*n134n134;
    if (ddn45 < 0.0f) return;
    Vec n135 = n134*ddn;
    Vec p46 = p1 - n135;
    Vec n136 = p46 - p;
    float n136n136 = n136.x*n136.x + n136.y*n136.y + n136.z*n136.z;
    float f45 = 1.0f - n136n136/v;
    Vec n137 = n136*f45;
    float n137n137 = n137.x*n137.x + n137.y*n137.y + n137.z*n137.z;
    float ddn46 = ddn*n137n137;
    if (ddn46 < 0.0f) return;
    Vec n138 = n137*ddn;
    Vec p47 = p1 - n138;
    Vec n139 = p47 - p;
    float n139n139 = n139.x*n139.x + n139.y*n139.y + n139.z*n139.z;
    float f46 = 1.0f - n139n139/v;
    Vec n140 = n139*f46;
    float n140n140 = n140.x*n140.x + n140.y*n140.y + n140.z*n140.z;
    float ddn47 = ddn*n140n140;
    if (ddn47 < 0.0f) return;
    Vec n141 = n140*ddn;
    Vec p48 = p1 - n141;
    Vec n142 = p48 - p;
    float n142n142 = n142.x*n142.x + n142.y*n142.y + n142.z*n142.z;
    float f47 = 1.0f - n142n142/v;
    Vec n143 = n142*f47;
    float n143n143 = n143.x*n143.x + n143.y*n143.y + n143.z*n143.z;
    float ddn48 = ddn*n143n143;
    if (ddn48 < 0.0f) return;
    Vec n144 = n143*ddn;
    Vec p49 = p1 - n144;
    Vec n145 = p49 - p;
    float n145n145 = n145.x*n145.x + n145.y*n145.y + n145.z*n145.z;
    float f48 = 1.0f - n145n145/v;
    Vec n146 = n145*f48;
    float n146n146 = n146.x*n146.x + n146.y*n146.y + n146.z*n146.z;
    float ddn49 = ddn*n146n146;
    if (ddn49 < 0.0f) return;
    Vec n147 = n146*ddn;
    Vec p50 = p1 - n147;
    Vec n148 = p50 - p;
    float n148n148 = n148.x*n148.x + n148.y*n148.y + n148.z*n148.z;
    float f49 = 1.0f - n148n148/v;
    Vec n149 = n148*f49;
    float n149n149 = n149.x*n149.x + n149.y*n149.y + n149.z*n149.z;
    float ddn50 = ddn*n149n149;
    if (ddn50 < 0.0f) return;
    Vec n150 = n149*ddn;
    Vec p51 = p1 - n150;
    Vec n151 = p51 - p;
    float n151n151 = n151.x*n151.x + n151.y*n151.y + n151.z*n151.z;
    float f50 = 1.0f - n151n151/v;
    Vec n152 = n151*f50;
    float n152n152 = n152.x*n152.x + n152.y*n152.y + n152.z*n152.z;
    float ddn51 = ddn*n152n152;
    if (ddn51 < 0.0f) return;
    Vec n153 = n152*ddn;
    Vec p52 = p1 - n153;
    Vec n154 = p52 - p;
    float n154n154 = n154.x*n154.x + n154.y*n154.y + n154.z*n154.z;
    float f51 = 1.0f - n154n154/v;
    Vec n155 = n154*f51;
    float n155n155 = n155.x*n155.x + n155.y*n155.y + n155.z*n155.z;
    float ddn52 = ddn*n155n155;
    if (ddn52 < 0.0f) return;
    Vec n156 = n155*ddn;
    Vec p53 = p1 - n156;
    Vec n157 = p53 - p;
    float n157n157 = n157.x*n157.x + n157.y*n157.y + n157.z*n157.z;
    float f52 = 1.0f - n157n157/v;
    Vec n158 = n157*f52;
    float n158n158 = n158.x*n158.x + n158.y*n158.y + n158.z*n158.z;
    float ddn53 = ddn*n158n158;
    if (ddn53 < 0.0f) return;
    Vec n159 = n158*ddn;
    Vec p54 = p1 - n159;
    Vec n160 = p54 - p;
    float n160n160 = n160.x*n160.x + n160.y*n160.y + n160.z*n160.z;
    float f53 = 1.0f - n160n160/v;
    Vec n161 = n160*f53;
    float n161n161 = n161.x*n161.x + n161.y*n161.y + n161.z*n161.z;
    float ddn54 = ddn*n161n161;
    if (ddn54 < 0.0f) return;
    Vec n162 = n161*ddn;
    Vec p55 = p1 - n162;
    Vec n163 = p55 - p;
    float n163n163 = n163.x*n163.x + n163.y*n163.y + n163.z*n163.z;
    float f54 = 1.0f - n163n163/v;
    Vec n164 = n163*f54;
    float n164n164 = n164.x*n164.x + n164.y*n164.y + n164.z*n164.z;
    float ddn55 = ddn*n164n164;
    if (ddn55 < 0.0f) return;
    Vec n165 = n164*ddn;
    Vec p56 = p1 - n165;
    Vec n166 = p56 - p;
    float n166n166 = n166.x*n166.x + n166.y*n166.y + n166.z*n166.z;
    float f55 = 1.0f - n166n166/v;
    Vec n167 = n166*f55;
    float n167n167 = n167.x*n167.x + n167.y*n167.y + n167.z*n167.z;
    float ddn56 = ddn*n167n167;
    if (ddn56 < 0.0f) return;
    Vec n168 = n167*ddn;
    Vec p57 = p1 - n168;
    Vec n169 = p57 - p;
    float n169n169 = n169.x*n169.x + n169.y*n169.y + n169.z*n169.z;
    float f56 = 1.0f - n169n169/v;
    Vec n170 = n169*f56;
    float n170n170 = n170.x*n170.x + n170.y*n170.y + n170.z*n170.z;
    float ddn57 = ddn*n170n170;
    if (ddn57 < 0.0f) return;
    Vec n171 = n170*ddn;
    Vec p58 = p1 - n171;
    Vec n172 = p58 - p;
    float n172n172 = n172.x*n172.x + n172.y*n172.y + n172.z*n172.z;
    float f57 = 1.0f - n172n172/v;
    Vec n173 = n172*f57;
    float n173n173 = n173.x*n173.x + n173.y*n173.y + n173.z*n173.z;
    float ddn58 = ddn*n173n173;
    if (ddn58 < 0.0f) return;
    Vec n174 = n173*ddn;
    Vec p59 = p1 - n174;
    Vec n175 = p59 - p;
    float n175n175 = n175.x*n175.x + n175.y*n175.y + n175.z*n175.z;
    float f58 = 1.0f - n175n175/v;
    Vec n176 = n175*f58;
    float n176n176 = n176.x*n176.x + n176.y*n176.y + n176.z*n176.z;
    float ddn59 = ddn*n176n176;
    if (ddn59 < 0.0f) return;
    Vec n177 = n176*ddn;
    Vec p60 = p1 - n177;
    Vec n178 = p60 - p;
    float n178n178 = n178.x*n178.x + n178.y*n178.y + n178.z*n178.z;
    float f59 = 1.0f - n178n178/v;
    Vec n179 = n178*f59;
    float n179n179 = n179.x*n179.x + n179.y*n179.y + n179.z*n179.z;
    float ddn60 = ddn*n179n179;
    if (ddn60 < 0.0f) return;
    Vec n180 = n179*ddn;
    Vec p61 = p1 - n180;
    Vec n181 = p61 - p;
    float n181n181 = n181.x*n181.x + n181.y*n181.y + n181.z*n181.z;
    float f60 = 1.0f - n181n181/v;
    Vec n182 = n181*f60;
    float n182n182 = n182.x*n182.x + n182.y*n182.y + n182.z*n182.z;
    float ddn61 = ddn*n182n182;
    if (ddn61 < 0.0f) return;
    Vec n183 = n182*ddn;
    Vec p62 = p1 - n183;
    Vec n184 = p62 - p;
    float n184n184 = n184.x*n184.x + n184.y*n184.y + n184.z*n184.z;
    float f61 = 1.0f - n184n184/v;
    Vec n185 = n184*f61;
    float n185n185 = n185.x*n185.x + n185.y*n185.y + n185.z*n185.z;
    float ddn62 = ddn*n185n185;
    if (ddn62 < 0.0f) return;
    Vec n186 = n185*ddn;
    Vec p63
```

BVH

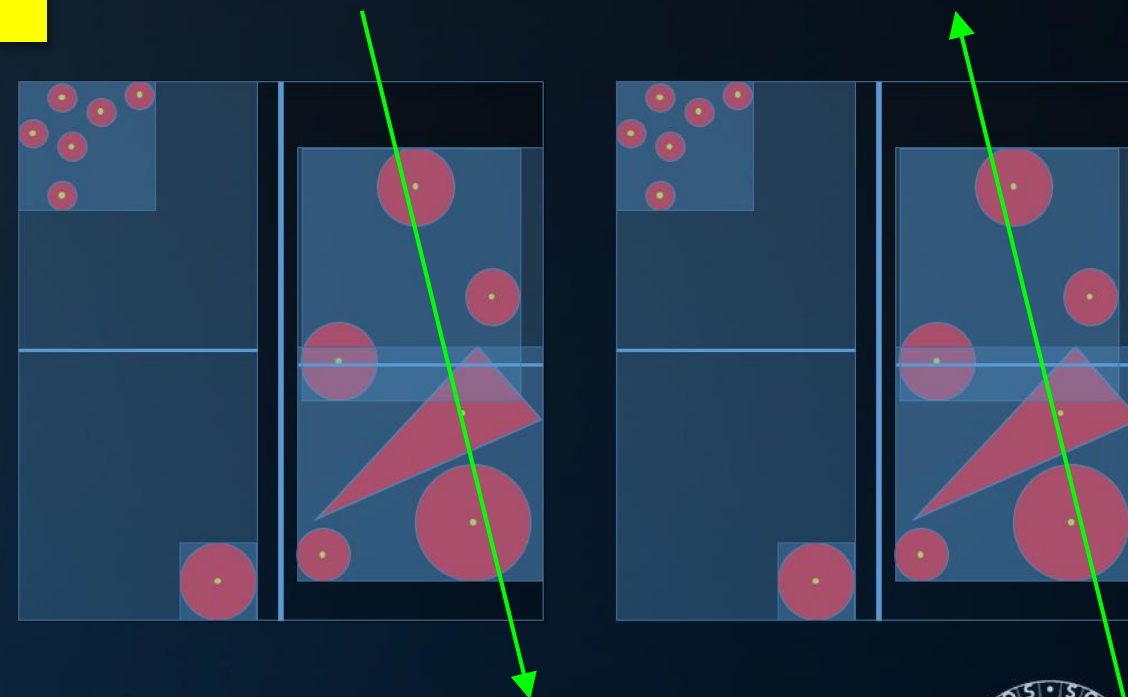
BVH Traversal

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:

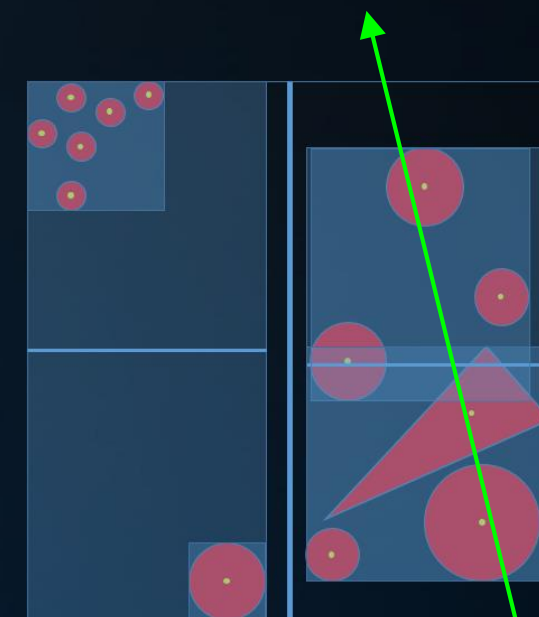
- Calculate distance to both child nodes
- Traverse the nearest child node first

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.

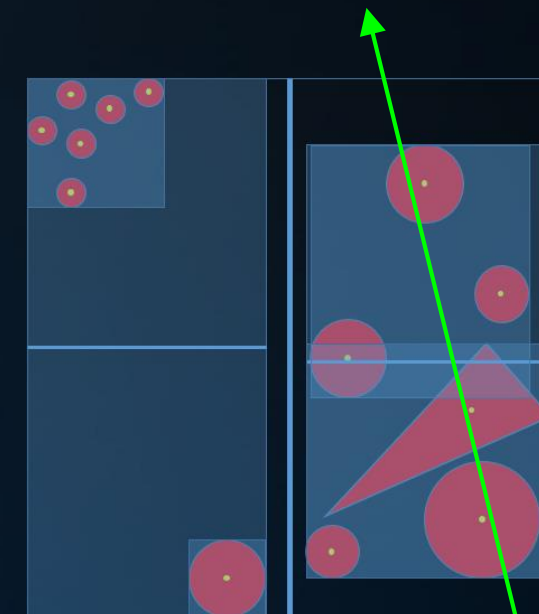
- Nodes may overlap.

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 2019 - February 2020

END of “Acceleration Structures”

next lecture: “Light Transport”

