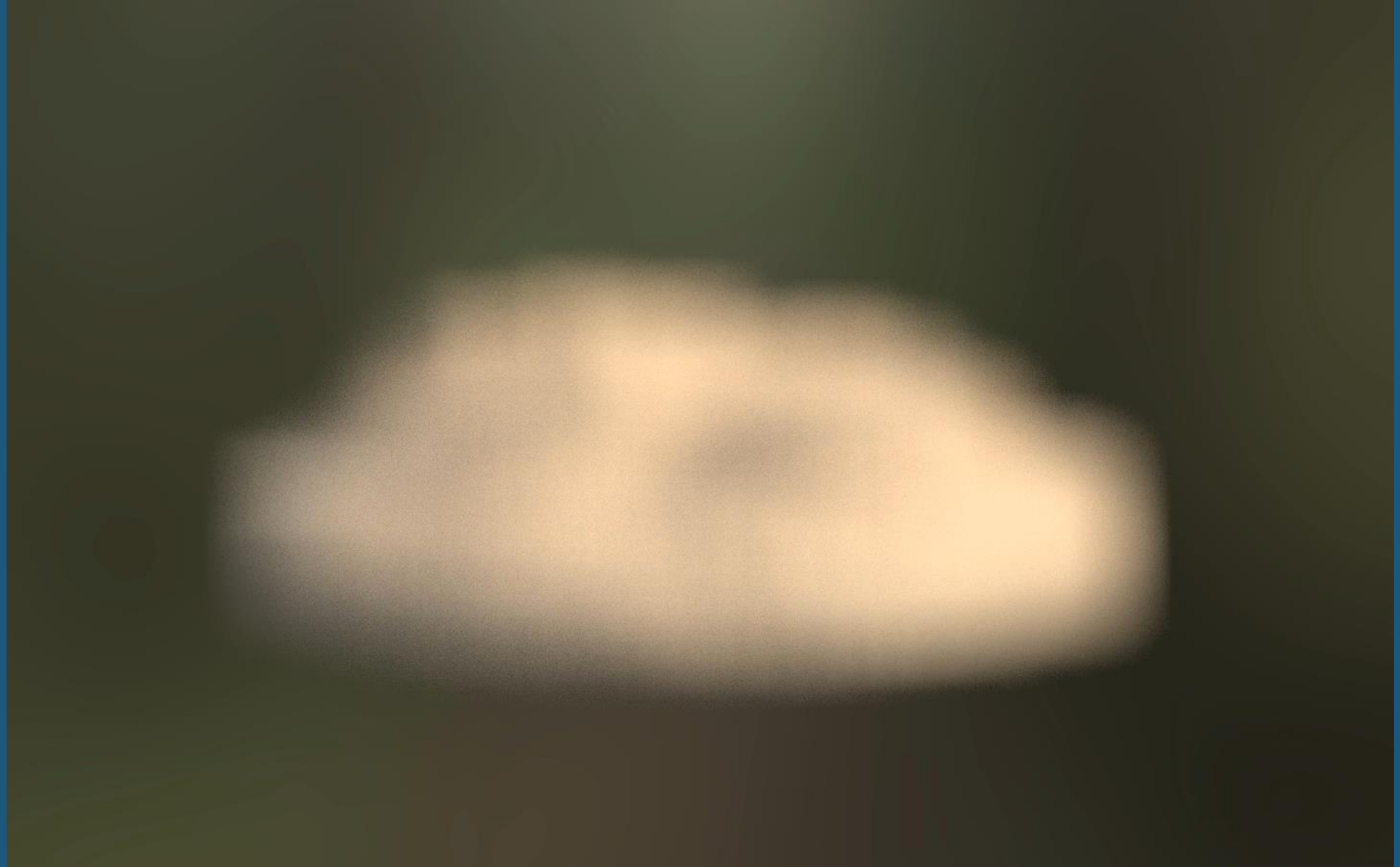
THE HITCHHIKER'S GUIDE TO THE VOXELS

DON’T PANIC

USHIOSTARFISH

MOTIVATION

- Ray Tracing with Triangles
  - A lot of solutions
    - Embree, OptiX, DXR, VK_KHR_acceleration_structure, HIPRT
  - A lot of issues you may face in rendering
    - Degenerate polygons, self-intersections, no-manifold meshes…
- Ray Tracing with Voxels?
  - How about using a dedicated acceleration structure & traversal?
    - Let's try ☺
  - Custom Geometry in Ray Tracing SDKs?
    - They are not optimized for voxels
    - Example) Embree 55M voxels: 5 GB only for BVH ☹



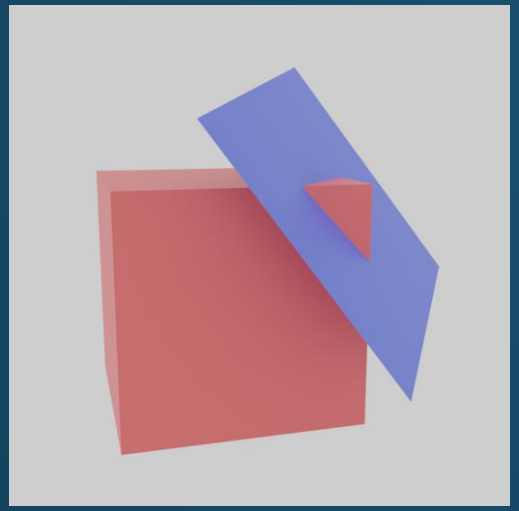
Voxelization

VOXELIZATION OF TRIANGLES

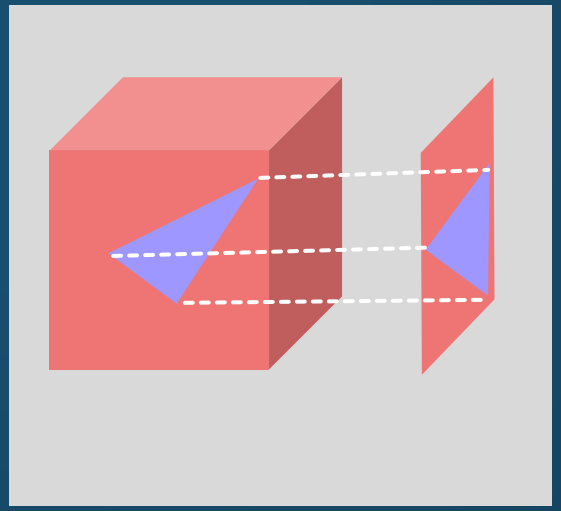
- Rasterization based
  - Cyril Crassin and Simon Green, "OpenGL Insights / Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer"
  - I DO NOT want to touch the rasterizer pipeline ☹
- Compute based
  - Michael Schwarz, Hans-Peter Seidel, "Fast Parallel Surface and Solid Voxelization on GPUs"
  - Jacopo Pantaleoni, " VoxelPipe: A Programmable Pipeline for 3D Voxelization"

VOXEL – TRIANGLE COLLISION DETECTION

- Voxelization: check if the voxel and triangle are overlapped
- Two conditions
  - (condition 1) Triangle plane and the voxel are overlapped
  - (condition 2) 2d projected triangle and 2d projected voxel are overlapped for each axis (X, Y, Z)



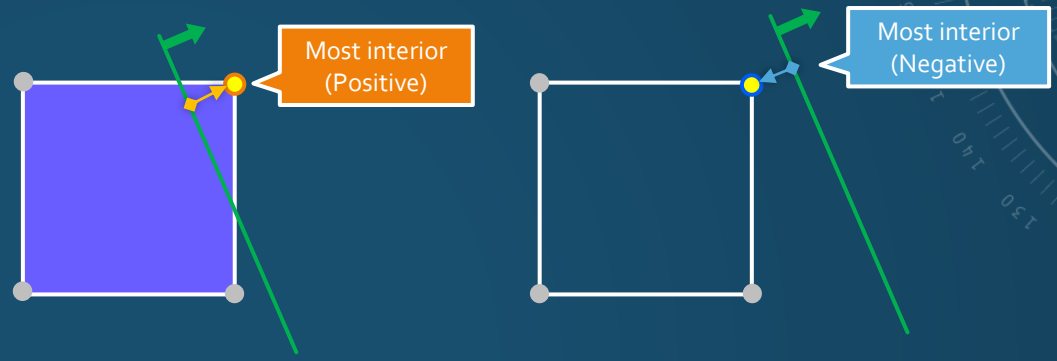
condition 1



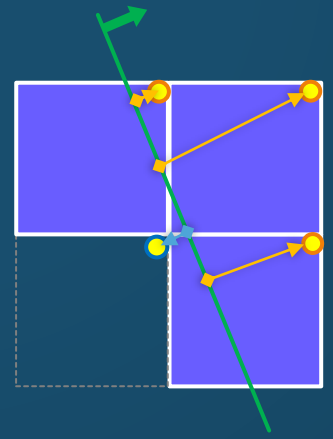
condition 2

2D TRIANGLE VS BOX OVERLAP TEST - CONSERVATIVE

- Use points called “critical points”
- Check if the most interior point is inside



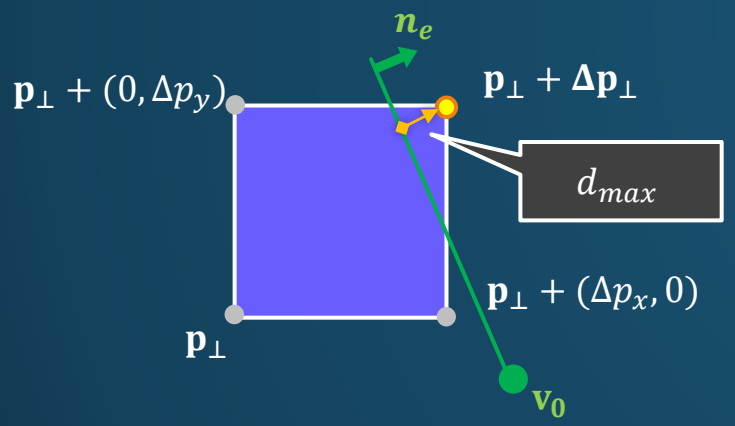
Critical Points for
Conservative Voxelization



Note: the points are shifted to clarify its belongings

2D TRIANGLE VS BOX OVERLAP TEST - CONSERVATIVE

- Critical points can be expressed by 2d projected position $\mathbf{p}_\perp$ and voxel size $\mathbf{\Delta p}_\perp = (\Delta p_x, \Delta p_y)$
- Calculate the signed distance d_{max} to the most interior point

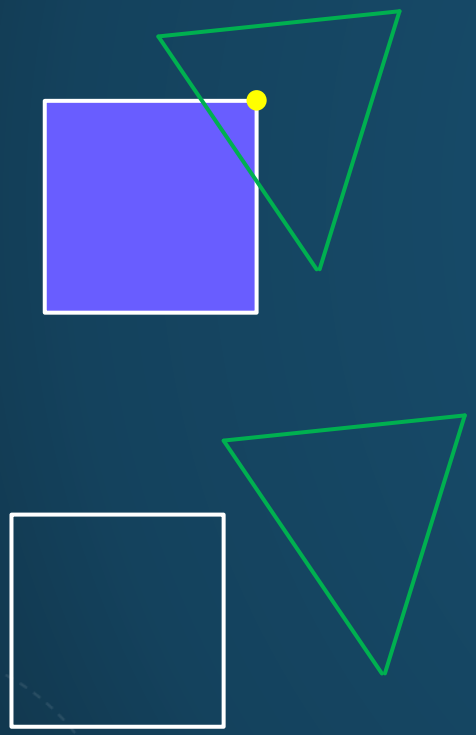


$$\mathbf{n_e} \cdot \mathbf{p}_\perp - \mathbf{n_e} \cdot \mathbf{v_0}$$
$$\mathbf{n_e} \cdot \left(\mathbf{p}_\perp + (0, \Delta p_y)\right) - \mathbf{n_e} \cdot \mathbf{v_0}$$
$$\mathbf{n_e} \cdot \left(\mathbf{p}_\perp + (\Delta p_x, 0)\right) - \mathbf{n_e} \cdot \mathbf{v_0}$$
$$\mathbf{n_e} \cdot (\mathbf{p}_\perp + \mathbf{\Delta p}_\perp) - \mathbf{n_e} \cdot \mathbf{v_0}$$

Max value of these = d_{max}

➡ $$d_{max} = \mathbf{n_e} \cdot \mathbf{p}_\perp + \underbrace{\max\left(\mathbf{n}_{e,x}\Delta p_x, 0\right) + \max\left(\mathbf{n}_{e,y}\Delta p_y, 0\right) - \mathbf{n_e} \cdot \mathbf{v_0}}_{\text{Constant. Calculate just once per edge.}}$$

2D TRIANGLE VS BOX OVERLAP TEST - CONSERVATIVE



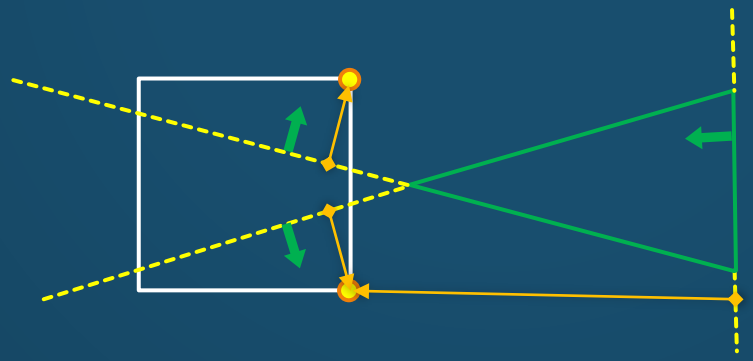
2d overlap (conservative) =

$$\forall_d \in edges, 0 \leq d_{max}$$

$$d_{max} = \mathbf{n_e} \cdot \mathbf{p}_\perp + \max\left(\mathbf{n}_{e,x}\Delta \mathrm{p_x}, 0\right) + \max\left(\mathbf{n}_{e,y}\Delta \mathrm{p_y}, 0\right) - \mathbf{n_e} \cdot \mathbf{v_0}$$

2D TRIANGLE VS BOX OVERLAP TEST – CORNER CASES

- There are corner cases in which the test shows "overlap" but it is not true
  - It can be completely filtered by additional 2d bounding box test
  - The subsequent algorithm in this presentation does need the test
    - 2d bounding box test is implicitly taken into account
    - You may need an extra 2d bounding box test if you use it for other purposes

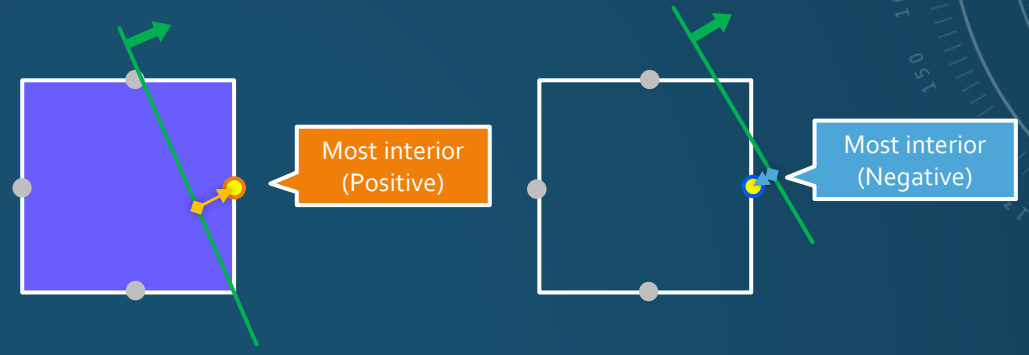


A corner case – false positive

2D TRIANGLE VS BOX OVERLAP TEST – 6 SEPARATING

- Another variant of voxelization
  - Often closer to the original geometry
- Same rule but just different "Critical Points"

Critical Points for
6-separating Voxelization



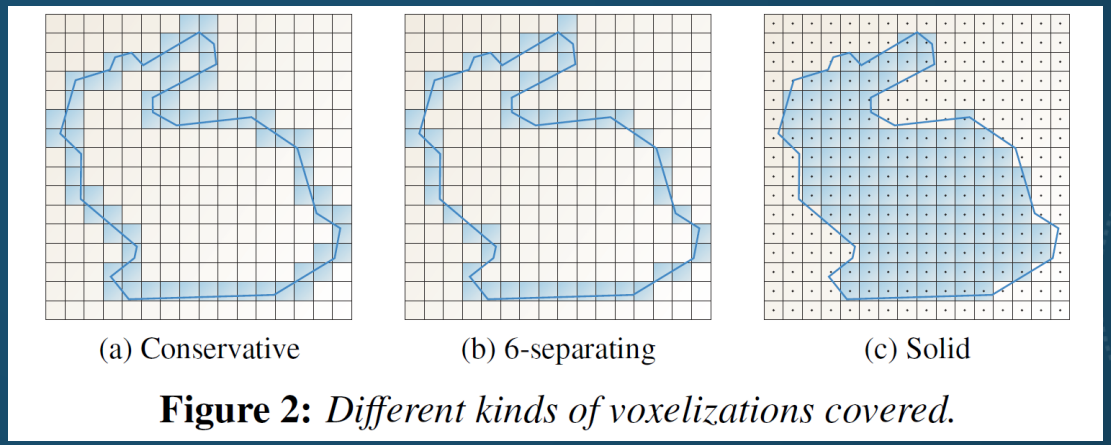
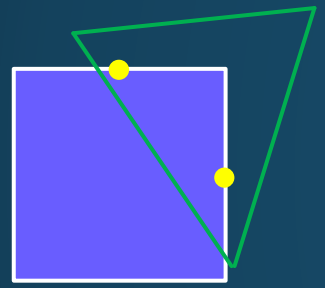
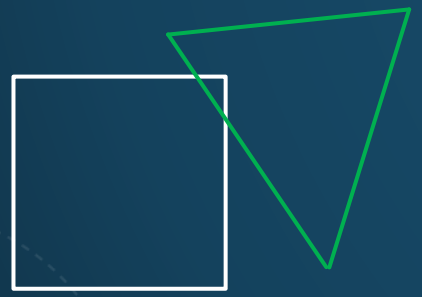


Figure 2: *Different kinds of voxelizations covered.*

"Fast Parallel Surface and Solid Voxelization on GPUs"

2D TRIANGLE VS BOX OVERLAP TEST – 6 SEPARATING





2d overlap (6-separating) =

$$\forall_d \in edges, 0 \leq d_{max}$$

$$d_{max} = \mathbf{n_e} \cdot \mathbf{p}_\perp + \mathbf{n_e} \cdot \left(\frac{1}{2}\Delta\mathbf{p}_\perp - \mathbf{v_0}\right) + \frac{1}{2}\Delta p \max\left(|\mathbf{n}_{e,x}|, |\mathbf{n}_{e,y}|\right)$$

Same with Conservative

Constant. Calculate just once per edge

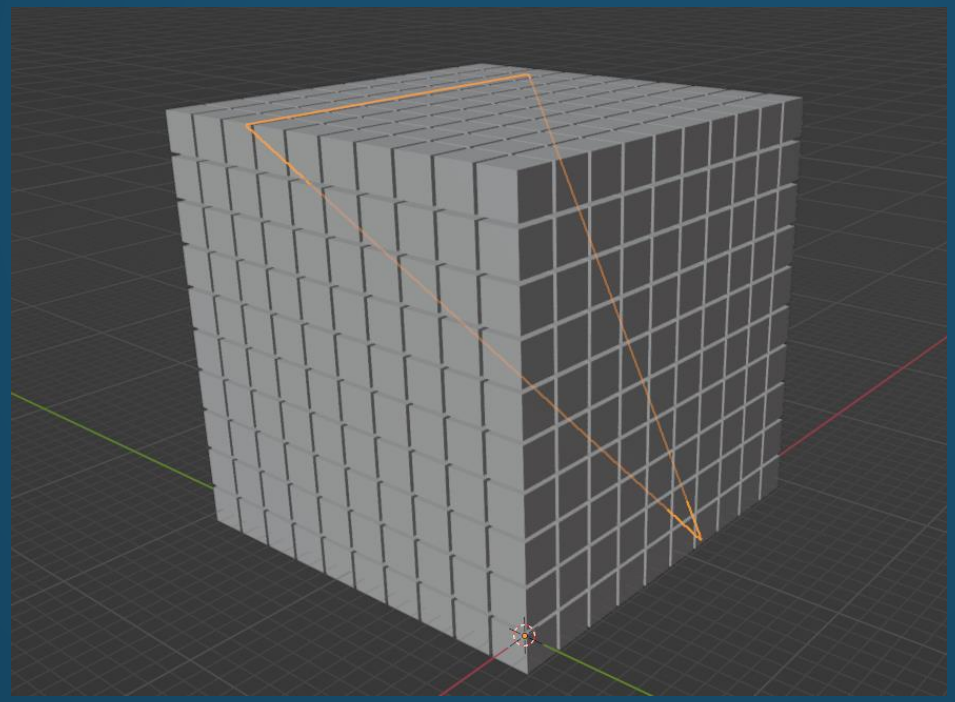
Where $\Delta p_x = \Delta p_y = \Delta p$

- As an implementation-wise advantage, the test algorithm can be the same
  - Just switching the constant terms ☺

A PROBLEM OF VOXEL VISITING

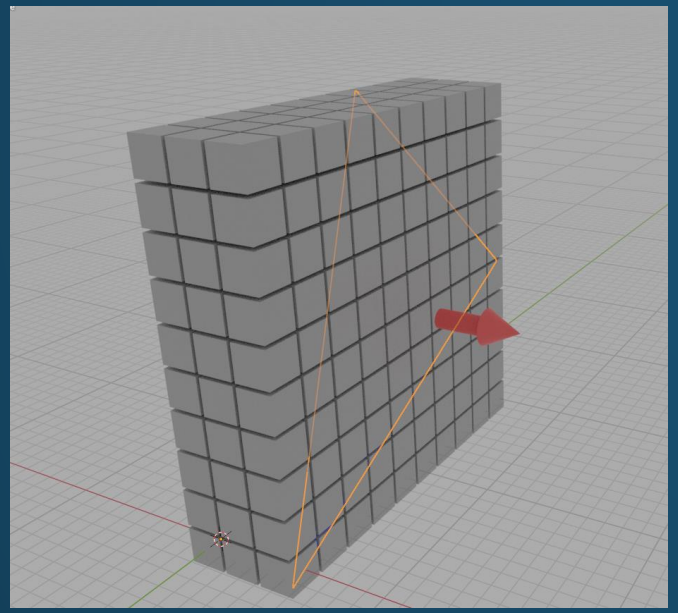
To visit all voxels in the bounding box of a triangle

- Too many voxels to check!!!! ☹

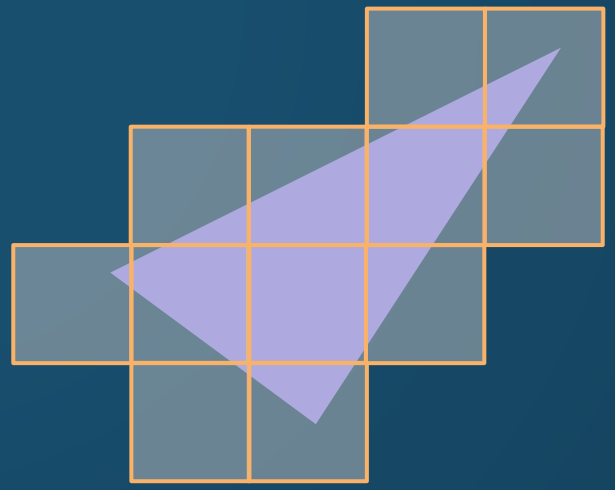


MINIMIZING NUMBER OF VOXELS TO VISIT

- Choose "major" axis
- Rasterize them in 2D space



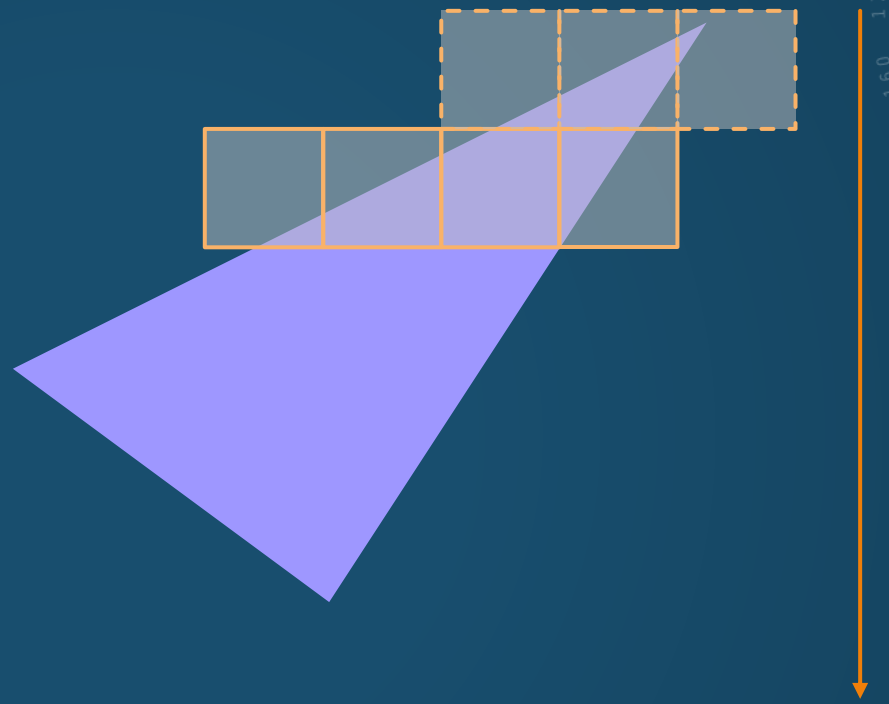
Major Axis (X)



Rasterize in 2D

RASTERIZATION

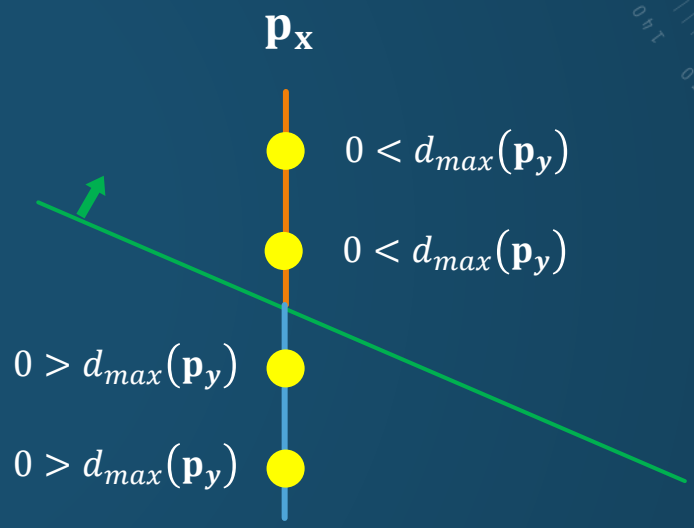
- Process voxels for each line
- The first axis-x range is trivial
  - But for the second axis-y?



THE EDGE FUNCTION

- d_{max} can be considered a linear function

$$d_{max} = \mathbf{n_e} \cdot \mathbf{p} + d_{const}$$
$$= \mathbf{n_{e,x}} \cdot \mathbf{p_x} + \mathbf{n_{e,y}} \cdot \mathbf{p_y} + d_{const}$$
$$d_{max}(\mathbf{p_y}) = \mathbf{n_{e,x}} \cdot \mathbf{p_x} + \mathbf{n_{e,y}} \cdot \mathbf{p_y} + d_{const}$$



THE EDGE FUNCTION

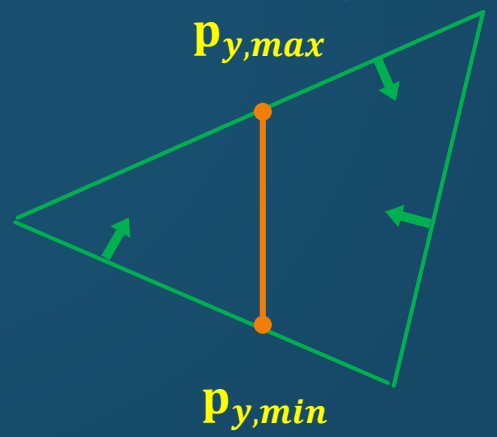
- We can obtain a half interval of $\mathbf{p}_y$ from $0 \leq d_{max}(\mathbf{p}_y)$

$$0 \leq \mathbf{n}_{\mathbf{e},x} \cdot \mathbf{p}_\mathbf{x} + \mathbf{n}_{\mathbf{e},y} \cdot \mathbf{p}_y + d_{const}$$

$$-\mathbf{n}_{\mathbf{e},x} \cdot \mathbf{p}_\mathbf{x} - d_{const} \leq \mathbf{n}_{\mathbf{e},y} \cdot \mathbf{p}_y$$

$$\begin{cases} -\dfrac{\mathbf{n}_{\mathbf{e},x} \cdot \mathbf{p}_\mathbf{x} + d_{const}}{\mathbf{n}_{\mathbf{e},y}} < \mathbf{p}_\mathbf{y} & \left(\mathbf{0} \leq \mathbf{n}_{\mathbf{e},\mathbf{y}}\right) \\ \mathbf{p}_\mathbf{y} \leq -\dfrac{\mathbf{n}_{\mathbf{e},x} \cdot \mathbf{p}_\mathbf{x} + d_{const}}{\mathbf{n}_{\mathbf{e},y}} & \left(\mathbf{n}_{\mathbf{e},\mathbf{y}} < \mathbf{0}\right) \end{cases}$$

We can obtain the min and max by combining 3 half intervals ☺



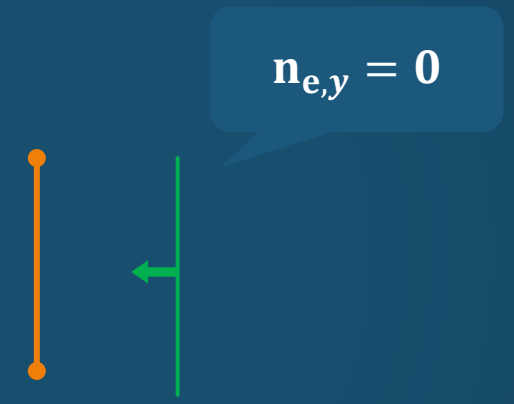
THE EDGE FUNCTION – A CORNER CASE

- $\mathbf{n}_{\mathbf{e},y} = \mathbf{0}$ case?

$$0 \leq \mathbf{n}_{\mathbf{e},x} \cdot \mathbf{p}_{\mathbf{x}} + \mathbf{n}_{\mathbf{e},y} \cdot \mathbf{p}_{y} + d_{const}$$

$$0 \leq \mathbf{n}_{\mathbf{e},x} \cdot \mathbf{p}_{\mathbf{x}} + d_{const}$$

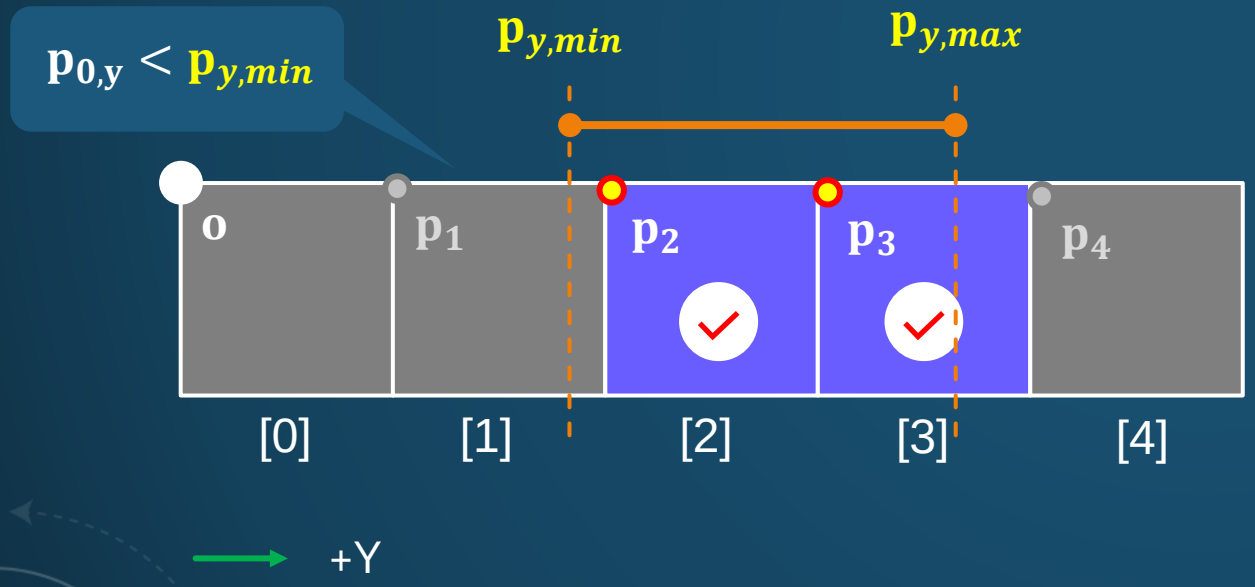
$$-d_{const} \leq \mathbf{n}_{\mathbf{e},x} \cdot \mathbf{p}_{\mathbf{x}}$$



All reject voxels it is false.
Otherwise, just skip the edge.

THE EDGE FUNCTION - VOXELS

- A condition for valid voxels
  - $\mathbf{p}_{y,min} \leq \mathrm{p_y} \leq \mathbf{p}_{y,max}$



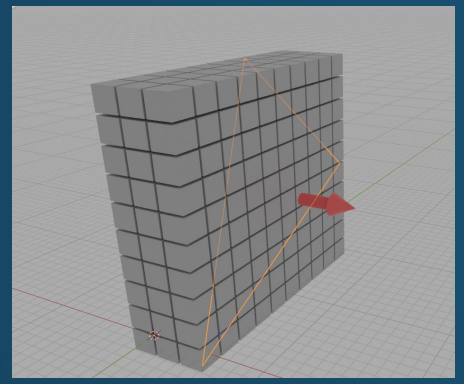
Voxel Index Range:

$$\left\lceil \frac{\mathbf{p}_{y,min} - \mathbf{o}_y}{\Delta p} \right\rceil \leq v_{index,y} \leq \left\lfloor \frac{\mathbf{p}_{y,max} - \mathbf{o}_y}{\Delta p} \right\rfloor$$

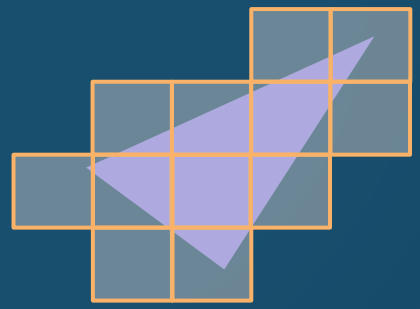
o: The voxel origin

DEPTH DIRECTION

- We have already minimized voxels in 2d
- What about the last axis?



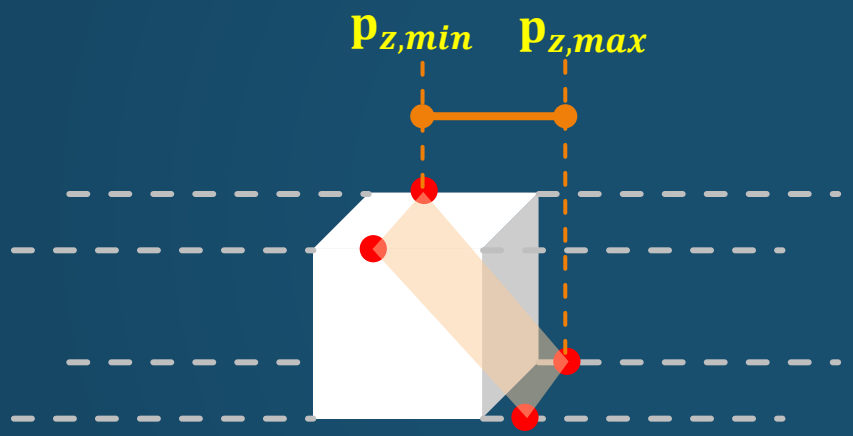
Major Axis (X)



Rasterize in 2D

DEPTH DIRECTION - CONSERVATIVE

- There are minimum and maximum z to overlap between the plane and edges-z



$$k_x = -\frac{\mathbf{n_x}}{\mathbf{n_z}}$$
$$k_y = -\frac{\mathbf{n_y}}{\mathbf{n_z}}$$

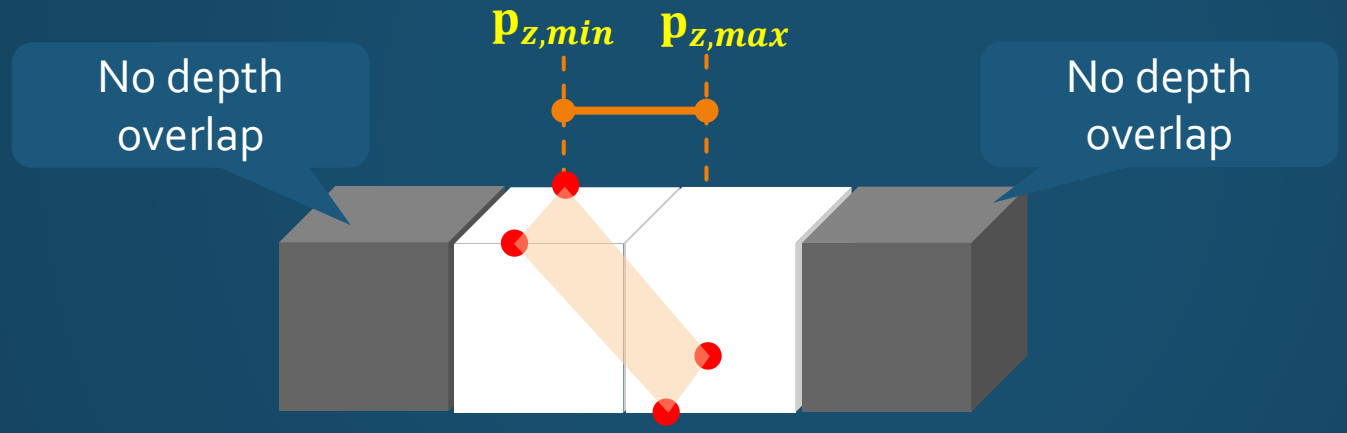
$$\mathbf{p}_{z,min} = k_x \mathbf{o_x} + k_y \mathbf{o}_y - k_x \mathbf{v}_{0,x} - k_y \mathbf{v}_{0,y} + \mathbf{v}_{0,z} + \Delta p\{\max(k_x, 0) + \max(k_y, 0)\}$$
$$\mathbf{p}_{z,max} = k_x \mathbf{o_x} + k_y \mathbf{o}_y - k_x \mathbf{v}_{0,x} - k_y \mathbf{v}_{0,y} + \mathbf{v}_{0,z} + \Delta p\{\min(k_x, 0) + \min(k_y, 0)\}$$

Constant. Calculate just once per triangle

$\mathbf{o}$: The voxel origin

DEPTH DIRECTION - CONSERVATIVE

- There are minimum and maximum z to overlap between the plane and edges-z
  - Just check the depth overlap ☺
  - Always less than or equal to 3 voxels as we chose a major axis for z

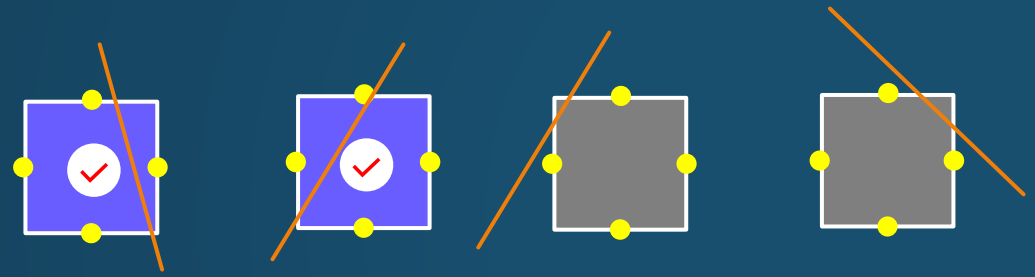


$$\left\lfloor \frac{\mathbf{p}_{z,min} - \mathbf{o}_z}{\Delta p} \right\rfloor \leq v_{index,z} \leq \left\lfloor \frac{\mathbf{p}_{z,max} - \mathbf{o}_z}{\Delta p} \right\rfloor$$

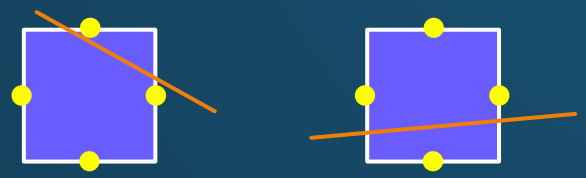
$\mathbf{o}$: The voxel origin

DEPTH DIRECTION – 6-SEPARATING

- Check if the plane is located between critical points

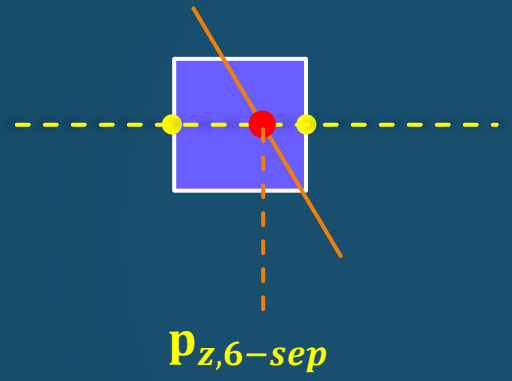


- Don’t have to handle a plane that is tilted to the z-axis by more than 45 degrees
  - Remember we chose a major axis for z



DEPTH DIRECTION – 6-SEPARATING

- Only the center axis is matter



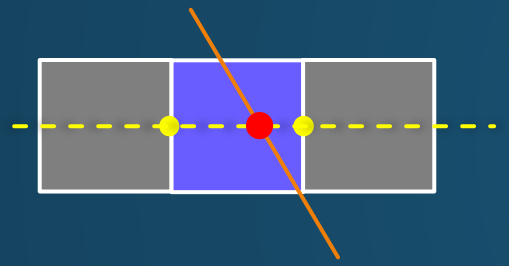
$$k_x = -\frac{\mathbf{n_x}}{\mathbf{n_z}}$$
$$k_y = -\frac{\mathbf{n_y}}{\mathbf{n_z}}$$

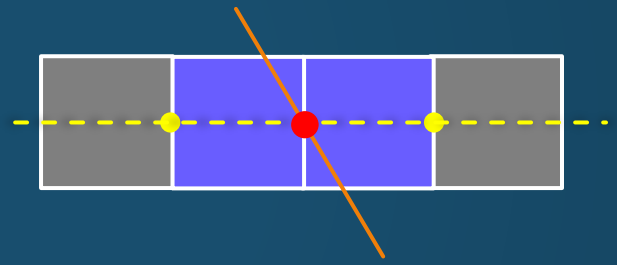
$$\mathbf{p}_{z,6-sep} = k_x \mathbf{o_x} + k_y \mathbf{o}_y - k_x \mathbf{v}_{0,x} - k_y \mathbf{v}_{0,y} + \mathbf{v}_{0,z} + \frac{1}{2}\Delta p\left(k_x + k_y\right)$$

Constant. Calculate just once per triangle

DEPTH DIRECTION – 6-SEPARATING

- Just 1 or 2 voxels
  - For conservative voxelization, it might be better to handle the case 2





$$v_{index,z} = \left\lfloor \frac{\mathbf{p}_{z,6-sep} - \mathbf{o}_z}{\Delta p} \right\rfloor$$

$$v_{index,z} = \left\{ \left\lfloor \frac{\mathbf{p}_{z,6-sep} - \mathbf{o}_z}{\Delta p} \right\rfloor - 1, \left\lfloor \frac{\mathbf{p}_{z,6-sep} - \mathbf{o}_z}{\Delta p} \right\rfloor \right\}$$

Case 1

Case 2

VOXELIZATION – HIGH LEVEL ALGORITHM

```
calculate constant terms for the triangle
int2 xrange = x range from bounding box of the triangle
for( int x = xrange.x; x <= xrange.y; x++ )
{
    int2 yrange = y range from 2d projected edges and the bounding box of the triangle
    for( int y = yrange.x; y <= yrange.y; y++ )
    {
        int2 zrange = z range from plane overlap
        for( int z = zrange.x; z <= zrange.y; z++ )
        {
            if( 2d overlap test for side axis ( x, y ) )
            {
                report a voxel
            }
        }
    }
}
```

VOXELIZATION – HIGH LEVEL ALGORITHM

(condition 2)
2d projected triangle and 2d projected voxel are overlapped for each axis (X, Y, Z)

```
calculate constant terms for the triangle
int2 xrange = x range from bounding box of the triangle
for( int x = xrange.x; x <= xrange.y; x++ )
{
    int2 yrange = y range from 2d projected edges and the bounding box of the triangle
    for( int y = yrange.x; y <= yrange.y; y++ )
    {
        int2 zrange = z range from plane overlap
        for( int z = zrange.x; z <= zrange.y; z++ )
        {
            if( 2d overlap test for side axis ( x, y ) )
            {
                report a voxel
            }
        }
    }
}
```

VOXELIZATION – HIGH LEVEL ALGORITHM

```
calculate constant terms for the triangle
int2 xrange = x range from bounding box of the triangle
for( int x = xrange.x; x <= xrange.y; x++ )
{
    int2 yrange = y range from 2d projected edges and the bounding box of the triangle
    for( int y = yrange.x; y <= yrange.y; y++ )
    {
        int2 zrange = z range from plane overlap
        for( int z = zrange.x; z <= zrange.y; z++ )
        {
            if( 2d overlap test for side axis ( x, y ) )
            {
                report a voxel
            }
        }
    }
}
```

(condition 1) Triangle plane and the voxel are overlapped

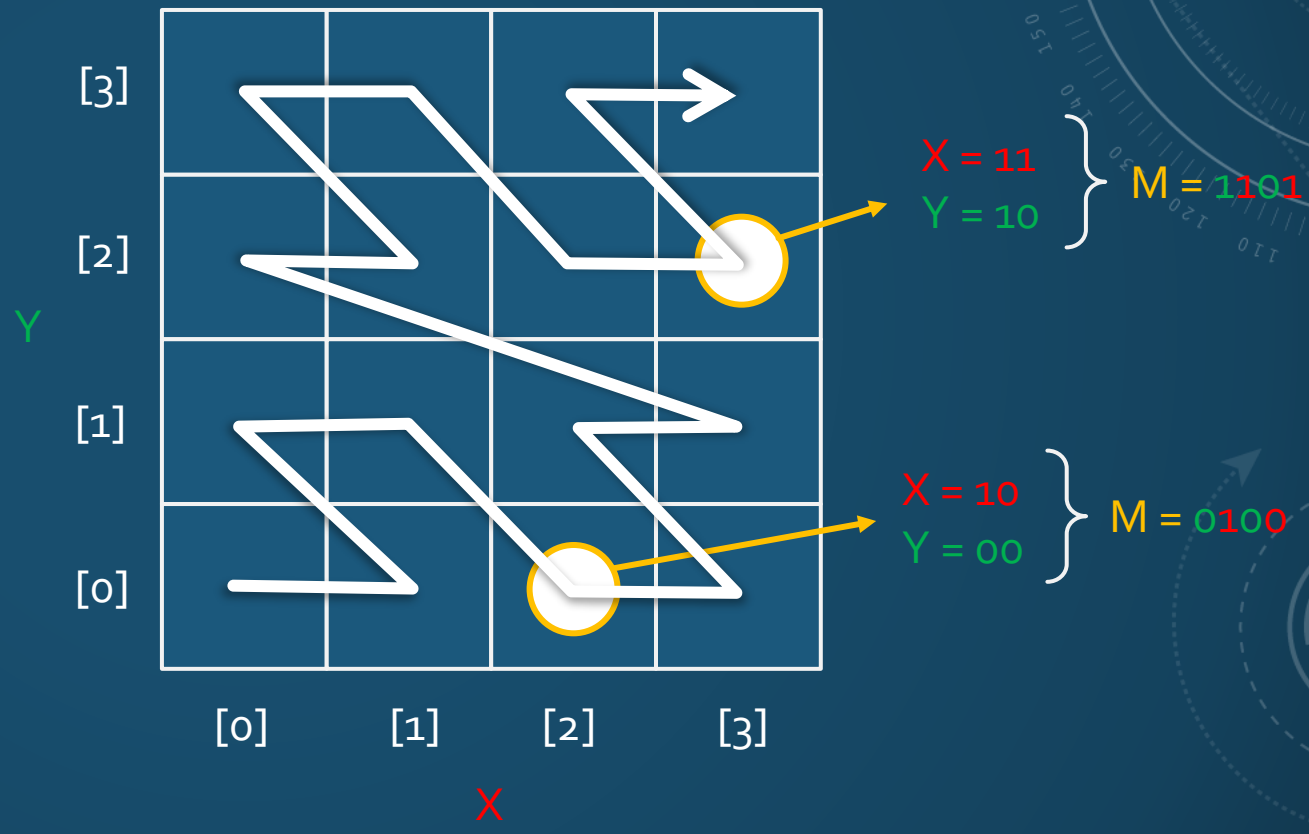
Acceleration structure

OCTREE

- Papers
  - J. Baert, A. Lagae and Ph. Dutré, “Out-of-Core Construction of Sparse Voxel Octrees”
  - Martin P¨atzold and Andreas Kolb, “Grid-Free Out-Of-Core Voxelization to Sparse Voxel Octrees on GPU”
- The target geometries are voxels
  - An octree leaf == a voxel
- I focus on “In core” algorithm for simplicity

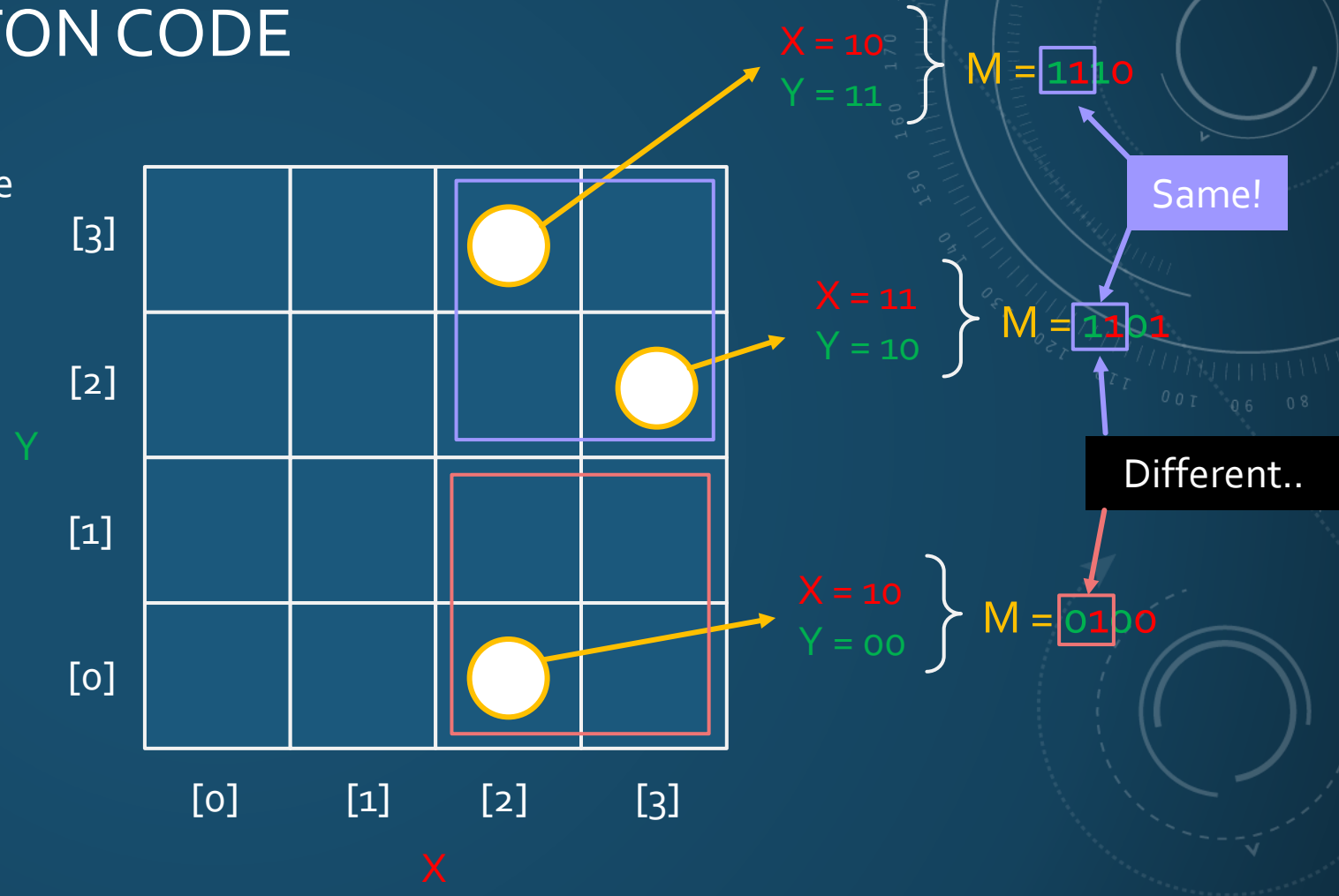
KEY COMPONENT: MORTON CODE

- Linearization of a grid
- Construction
  1. Get the binary index for each axis
  2. Interleave the bits



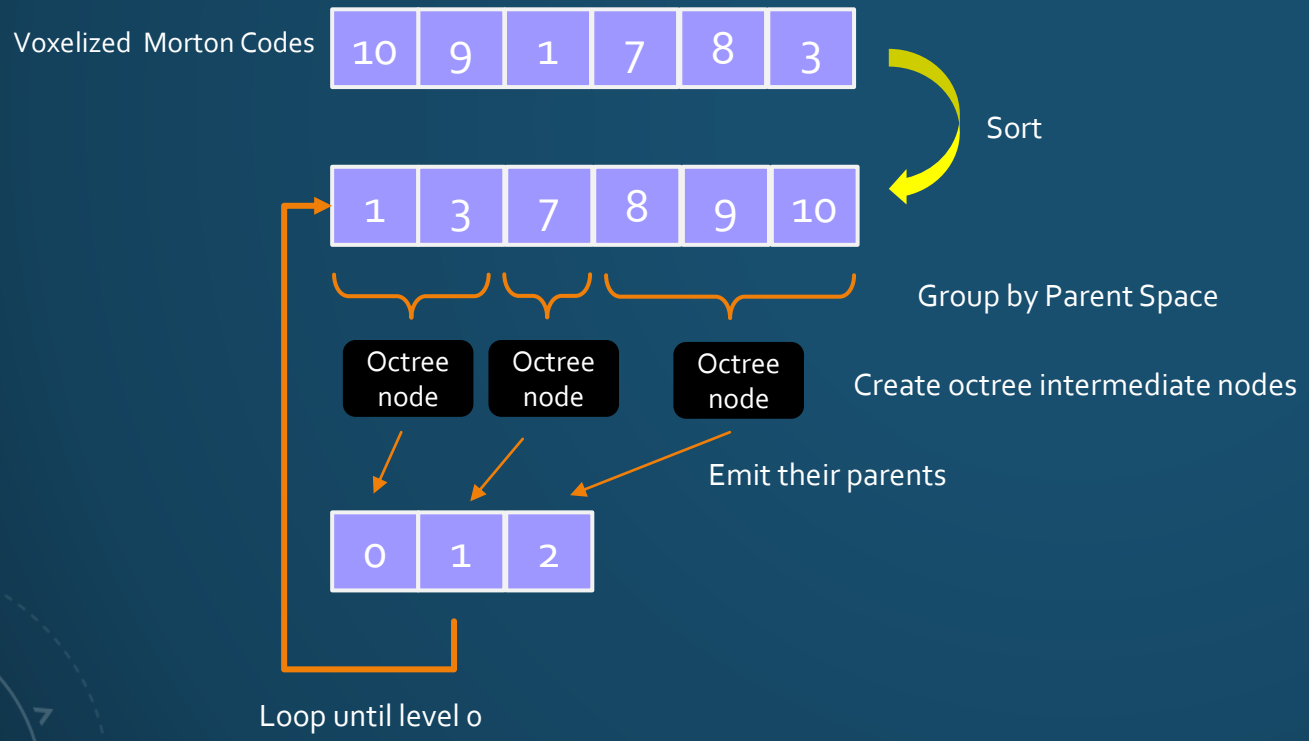
KEY COMPONENT: MORTON CODE

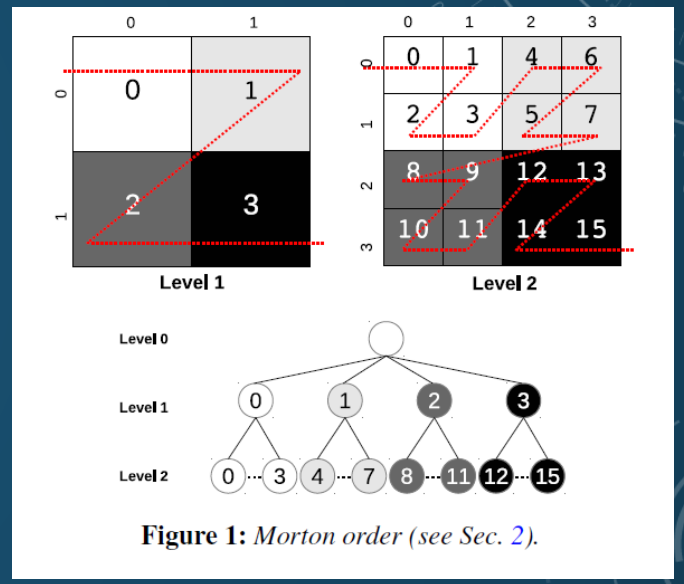
- Check if two Morton codes have the same parent
  - **Easy to get parent space by x >> 2**
  - **Just compare higher bits ☺**



BOTTOM-UP OCTREE CONSTRUCTION BY MORTON CODE

- A voxel can be expressed by a Morton code
- Group by their parent
  - Voxels with the same parent are always next to each other





2d example. From "Out-of-Core Construction of Sparse Voxel Octrees"

Octree Compression

NODE REUSE

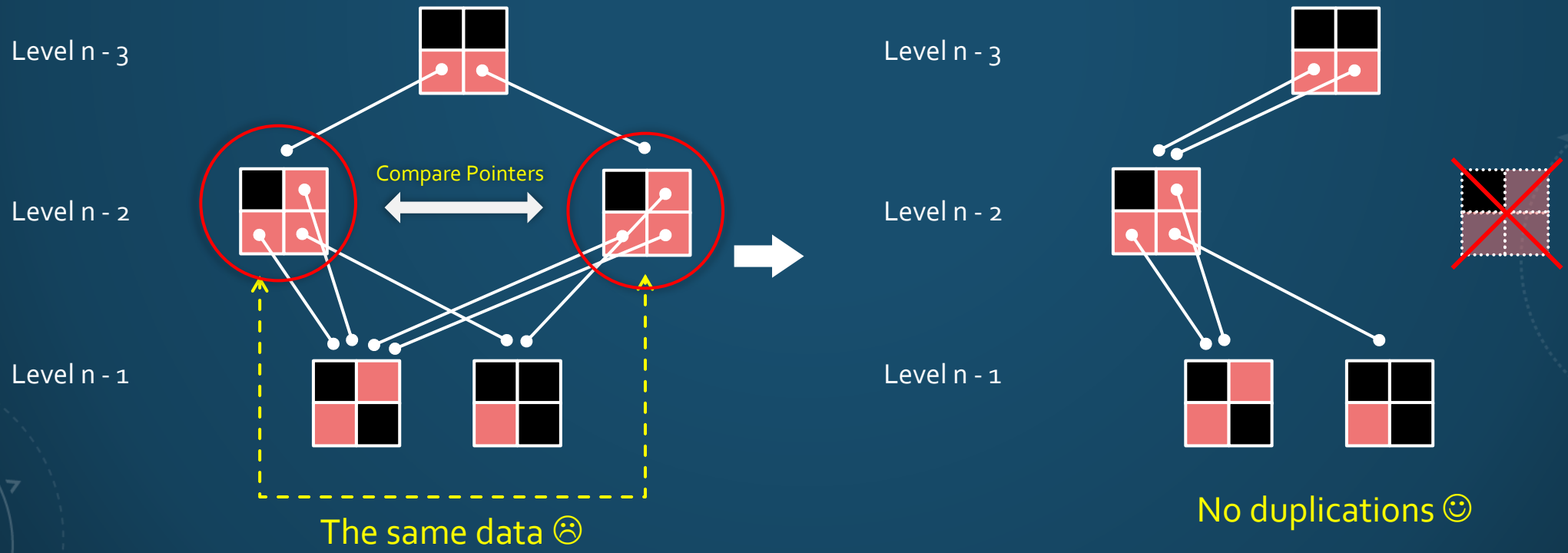
- Paper
  - Viktor Kampe, Erik Sintorn, Ulf Assarsson, “High Resolution Sparse Voxel DAGs”
- Remove Octree Node Redundancy

Example) Leaf Nodes

BOTTOM-UP COMPRESSION

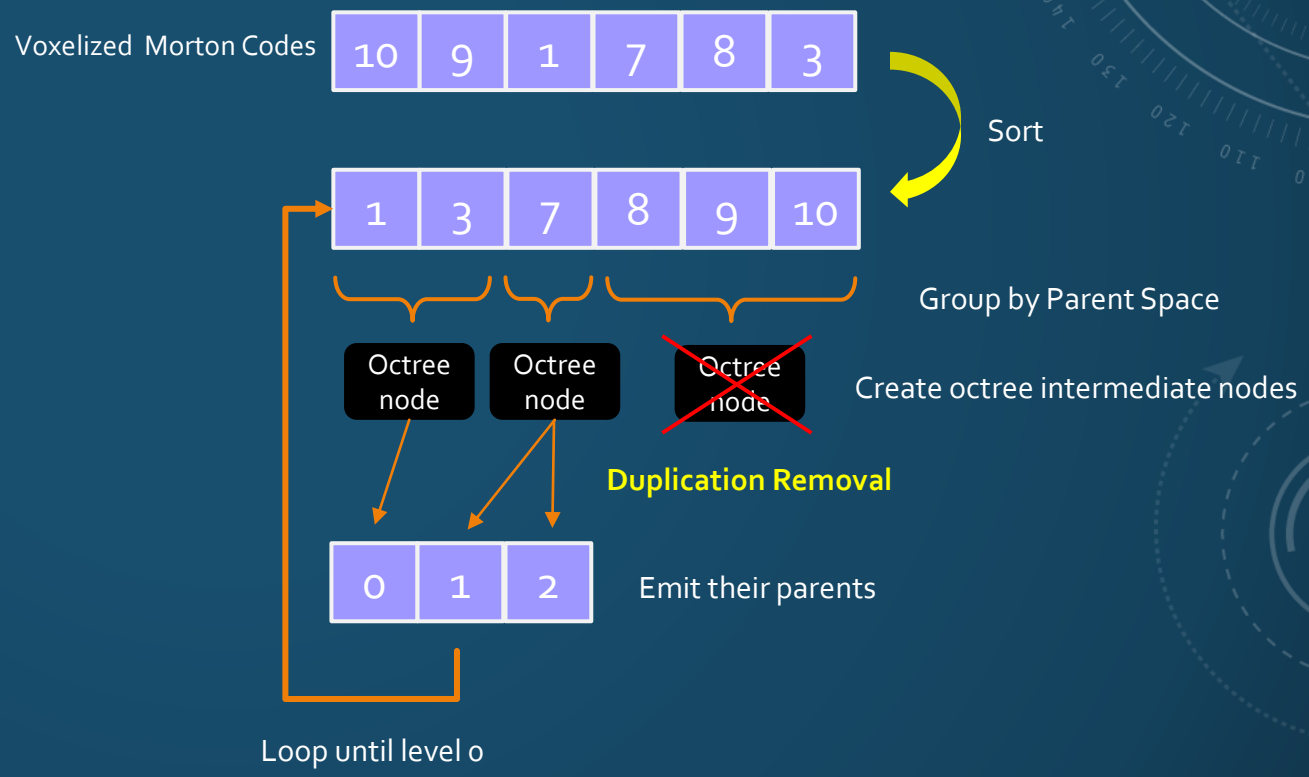
- Duplication removals in a bottom-up fashion
  - Possible to detect duplications by just checking child pointers ☺
  - For 2 nodes, their child pointers are equal ⇔ The tree structures are the same
  - A unique tree has a unique pointer

Example) 1 more upper node



BOTTOM-UP COMPRESSION DURING CONSTRUCTION

- Duplication removal during construction
  - Straight forward integration



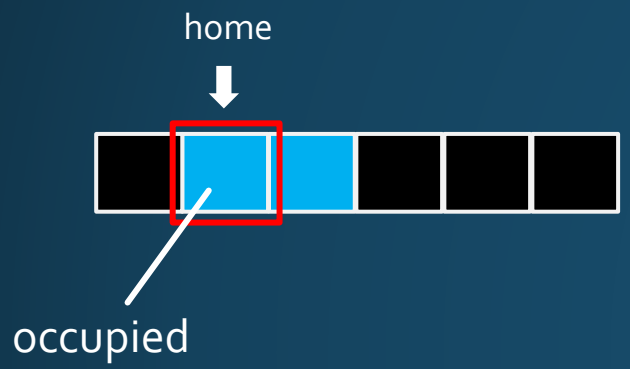
DUPLICATION REMOVAL ON THE GPU

- Linear Probing looks promising approach
  - But the size of the key is large - child pointers x 8 + mask

```
struct OctreeNode
{
    uint8_t mask;
    uint32_t children[8];
};
```

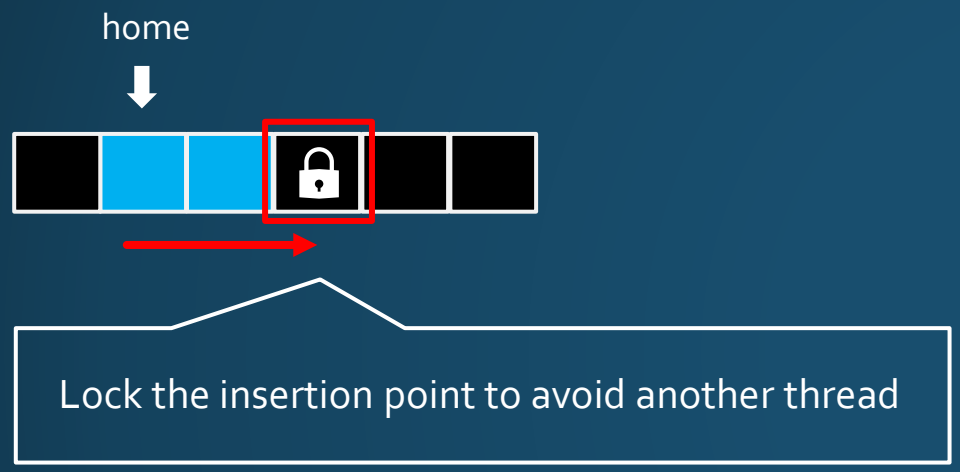
  - Partially-locking linear probing

PARTIALLY-LOCKING LINEAR PROBING



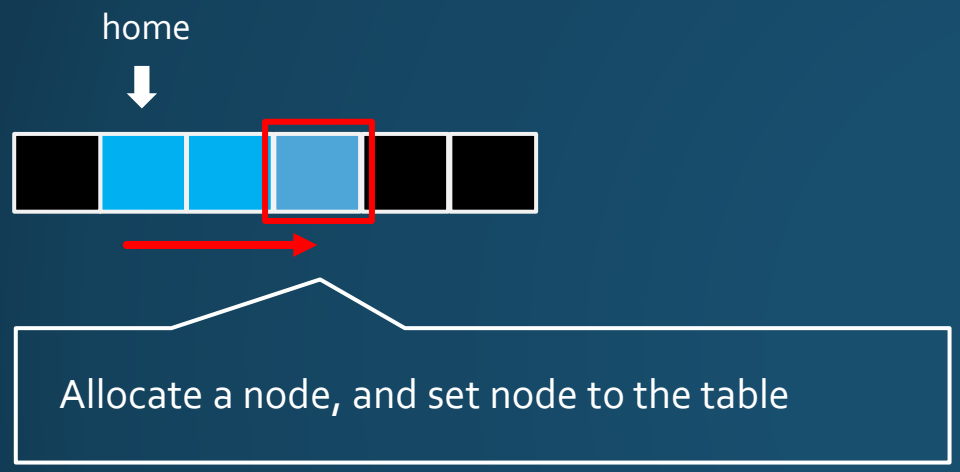

```
home = hash( newNode ) % tableSize
int node;
for( int i = 0 ; i < k ; i++ )
{
    int location = ( home + i ) % tableSize;
    uint32_t nodeInTable = atomicCAS( &table[location], 0, LOCK );
    __threadfence();
    if( nodeInTable == 0 ) // succeeded to lock
    {
        node = createNode();
        __threadfence();
        atomicExch( &table[location], node );
        break;
    }
    else if( nodeInTable == LOCK )
    {
        i--; // try again
    }
    else
    {
        equal = check equality nodeInTable and newNode
        if( equal )
        {
            node = nodeInTable
            break;
        }
    }
}
```

PARTIALLY-LOCKING LINEAR PROBING




```
home = hash( newNode ) % tableSize
int node;
for( int i = 0 ; i < k ; i++ )
{
    int location = ( home + i ) % tableSize;
    uint32_t nodeInTable = atomicCAS( &table[location], 0, LOCK );
    __threadfence();
    if( nodeInTable == 0 ) // succeeded to lock
    {
        node = createNode();
        __threadfence();
        atomicExch( &table[location], node );
        break;
    }
    else if( nodeInTable == LOCK )
    {
        i--; // try again
    }
    else
    {
        equal = check equality nodeInTable and newNode
        if( equal )
        {
            node = nodeInTable
            break;
        }
    }
}
```

PARTIALLY-LOCKING LINEAR PROBING




```
home = hash( newNode ) % tableSize
int node;
for( int i = 0 ; i < k ; i++ )
{
    int location = ( home + i ) % tableSize;
    uint32_t nodeInTable = atomicCAS( &table[location], 0, LOCK );
    __threadfence();
    if( nodeInTable == 0 ) // succeeded to lock
    {
        node = createNode();
        __threadfence();
        atomicExch( &table[location], node );
        break;
    }
    else if( nodeInTable == LOCK )
    {
        i--; // try again
    }
    else
    {
        equal = check equality nodeInTable and newNode
        if( equal )
        {
            node = nodeInTable
            break;
        }
    }
}
```

Note: This is a code for ITS environment

HOW SMALL IS IT?

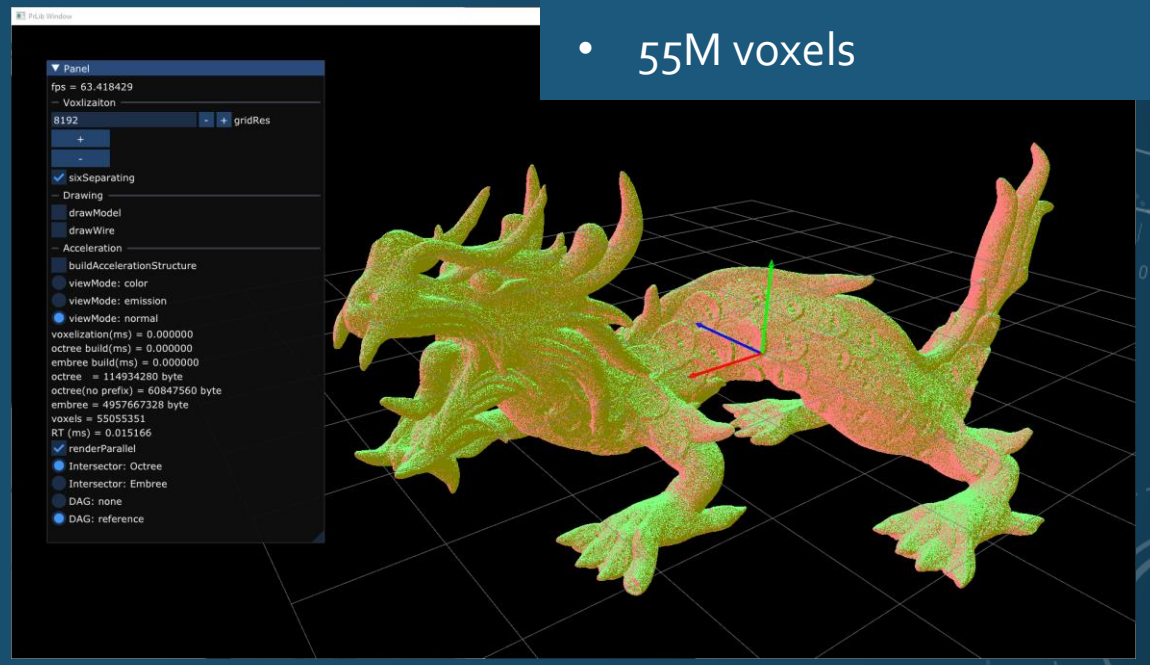
- **955M bytes for naive octree**
- **61M bytes for a compressed octree ☺**
  - 5G bytes with Embree(4.1.0) UserGeometry

```
struct OctreeNode
{
    uint8_t mask;
    uint32_t children[8];
};
```

Stupidly simple representation for both octree

Asian Dragon

- 8192^3 Resolution
- 55M voxels



Ray Traversal

PAPERS

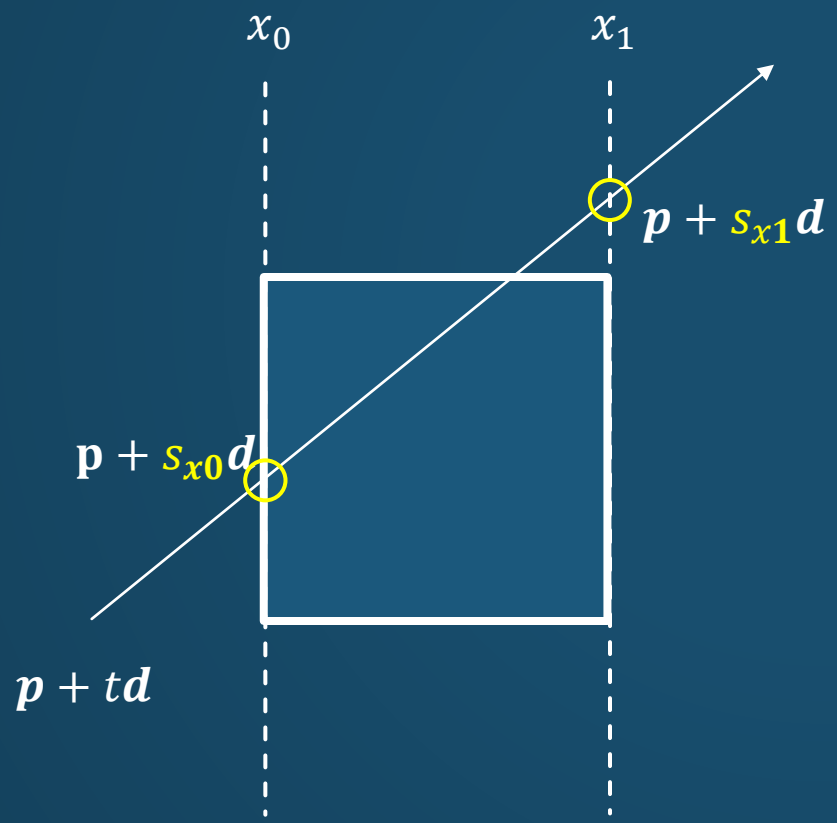
- John Amanatides and Andrew Woo, "A Fast Voxel Traversal Algorithm"
  - For uniform grid
- Mark Agate, Richard L. Grimsdale and Paul F. Lister, "The HERO Algorithm for Ray-Tracing Octrees"
- J. Revelles, C. Urena, M. Lastra, "An Efficient Parametric Algorithm for Octree Traversal"
  - The HERO Algorithm is simplified
  - Easier explanation: https://chiranjivi.tripod.com/octrav.html

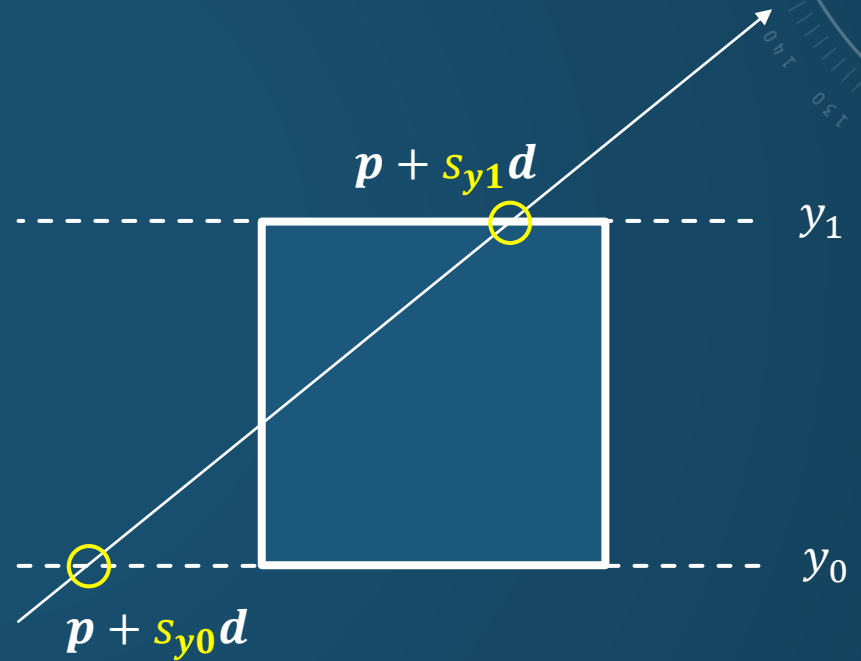
OCTREE TRAVERSAL

- Slab test

$$s_0 = \frac{x_0 - p_x}{d_x}$$

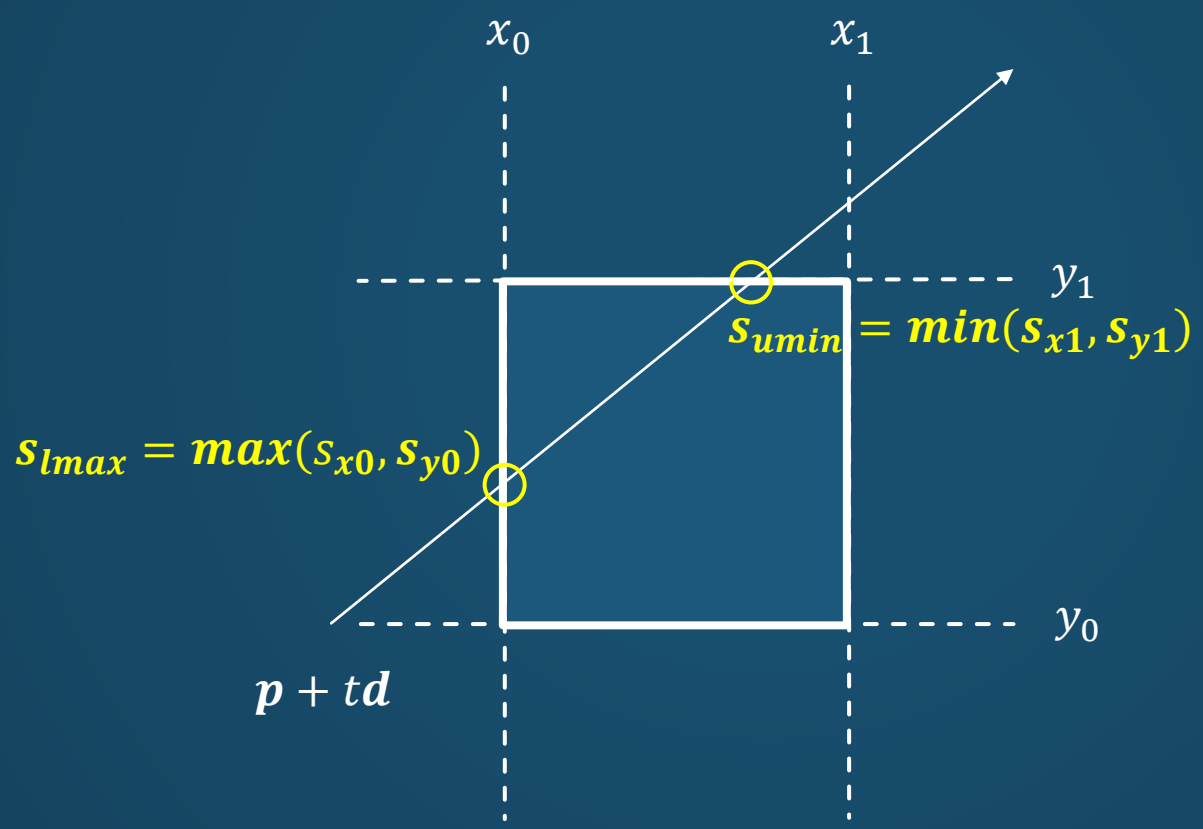
$$s_1 = \frac{x_1 - p_x}{d_x}$$





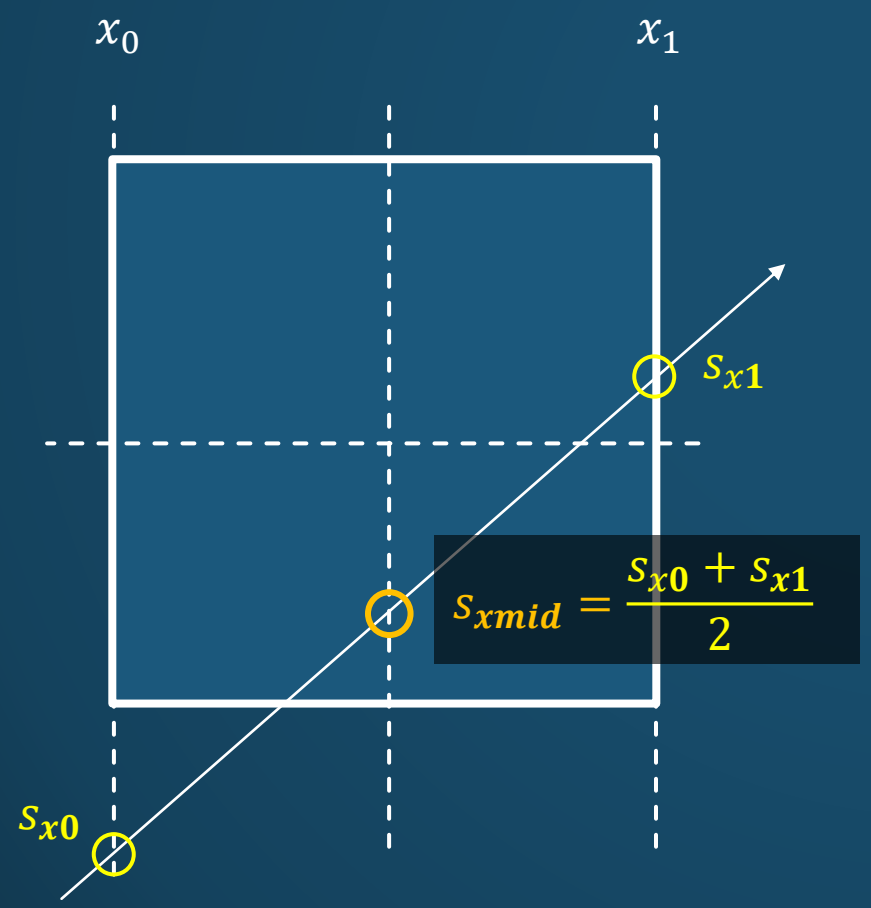
OCTREE TRAVERSAL

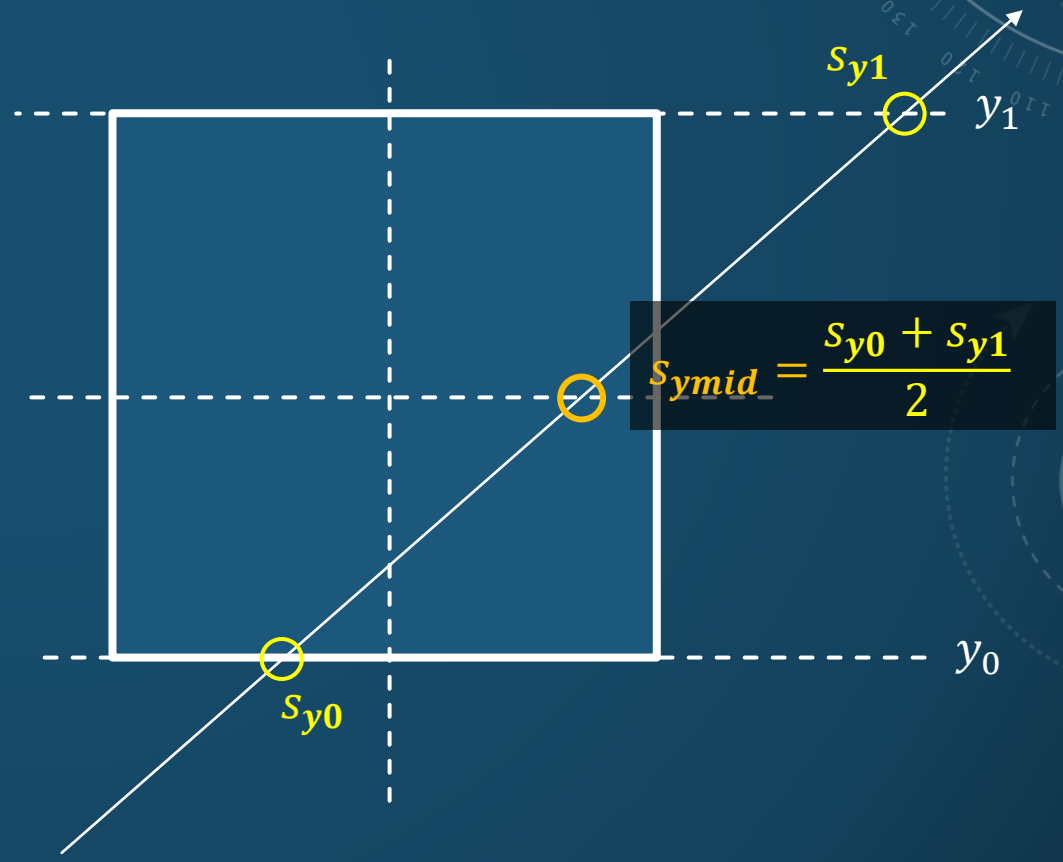
- Slab test



CHILD VOXEL TRAVERSAL

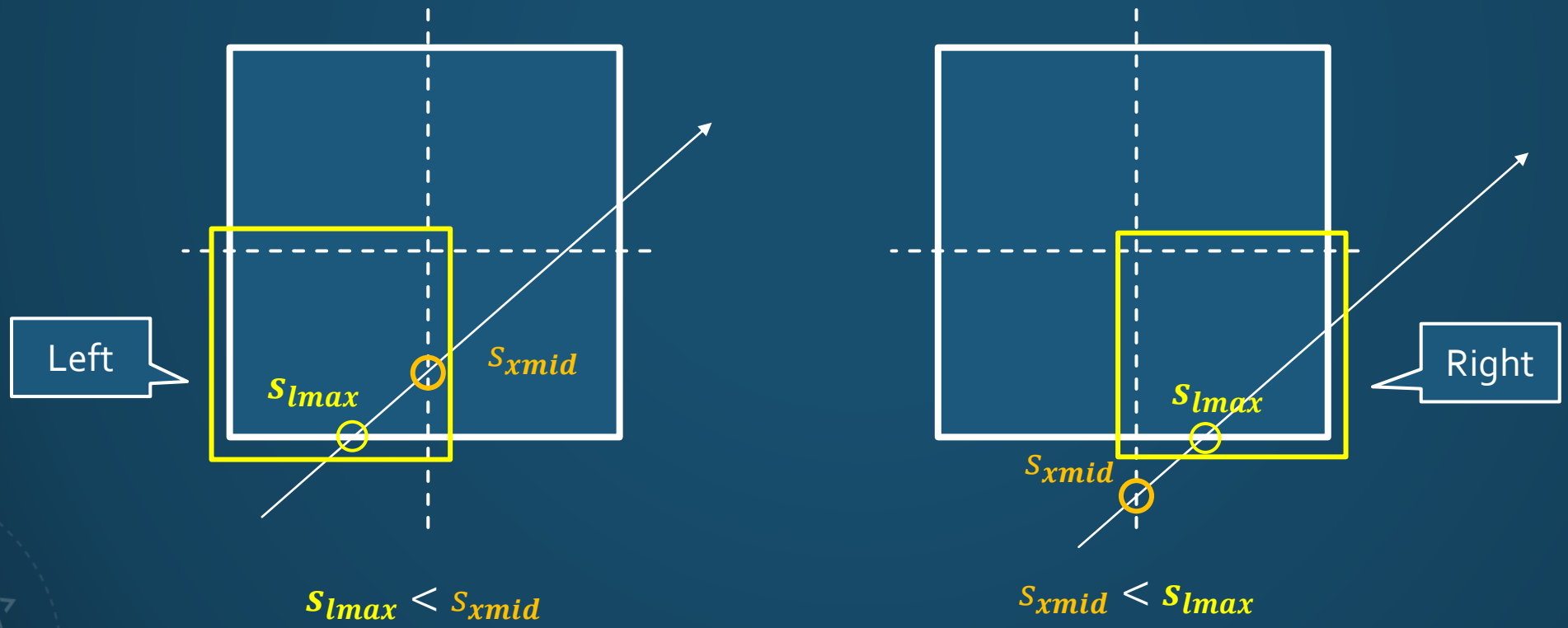
- The first crossed node





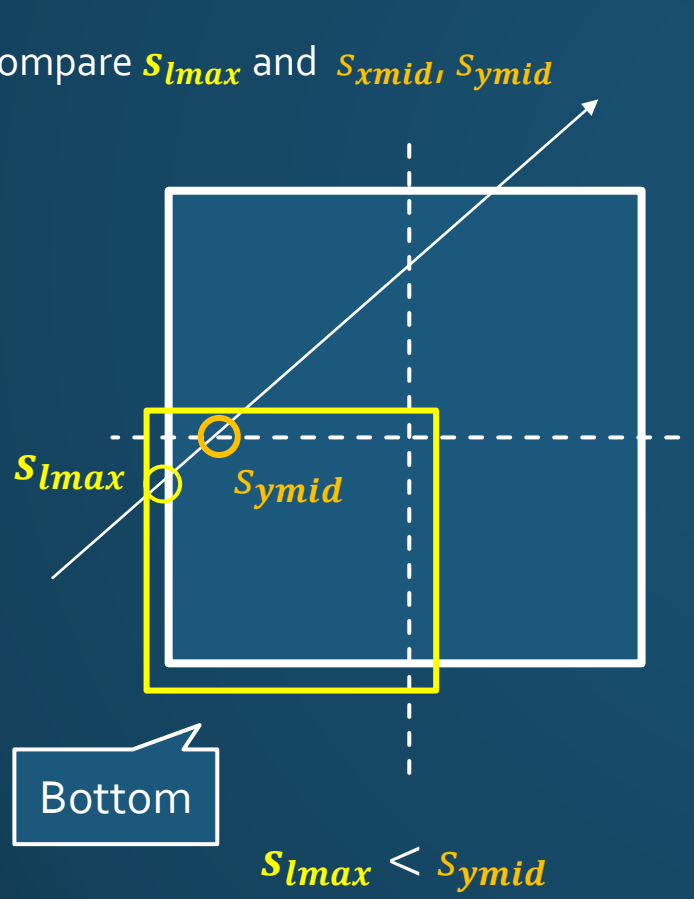
CHILD VOXEL TRAVERSAL

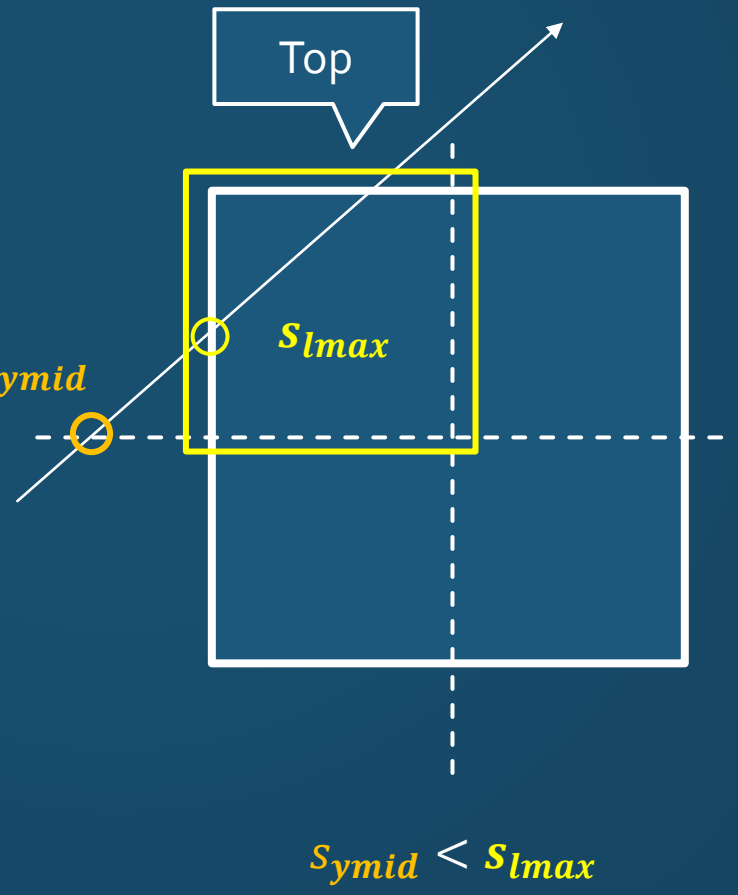
- The first crossed node
  - Compare s_{lmax} and s_{xmid}, s_{ymid}



CHILD VOXEL TRAVERSAL

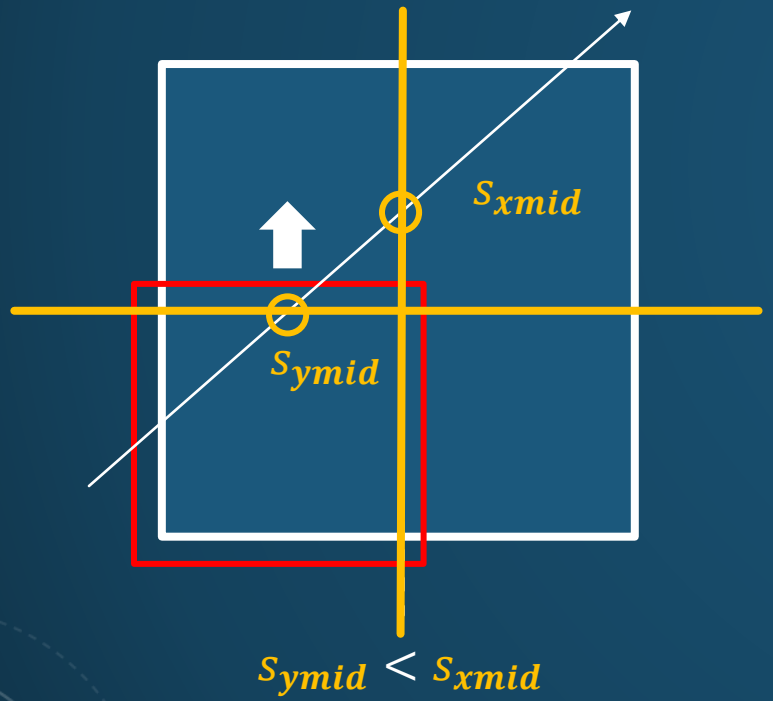
- The first crossed node
  - Compare s_{lmax} and s_{xmid}, s_{ymid}





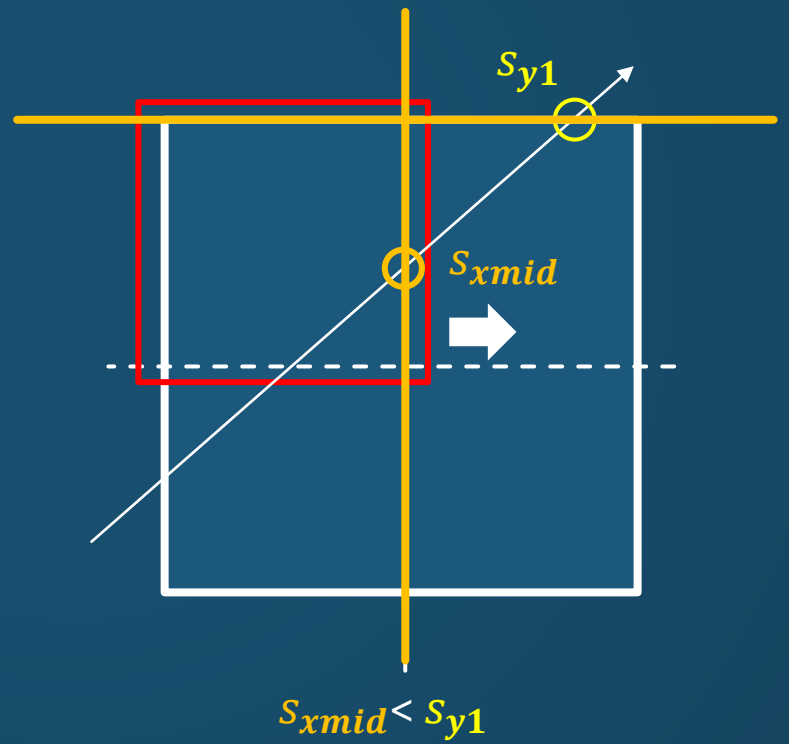
CHILD VOXEL TRAVERSAL

- What's the next node?
  - Choose the closest distance to the next planes



$S_{ymid} < S_{xmid}$

The next is **Y**

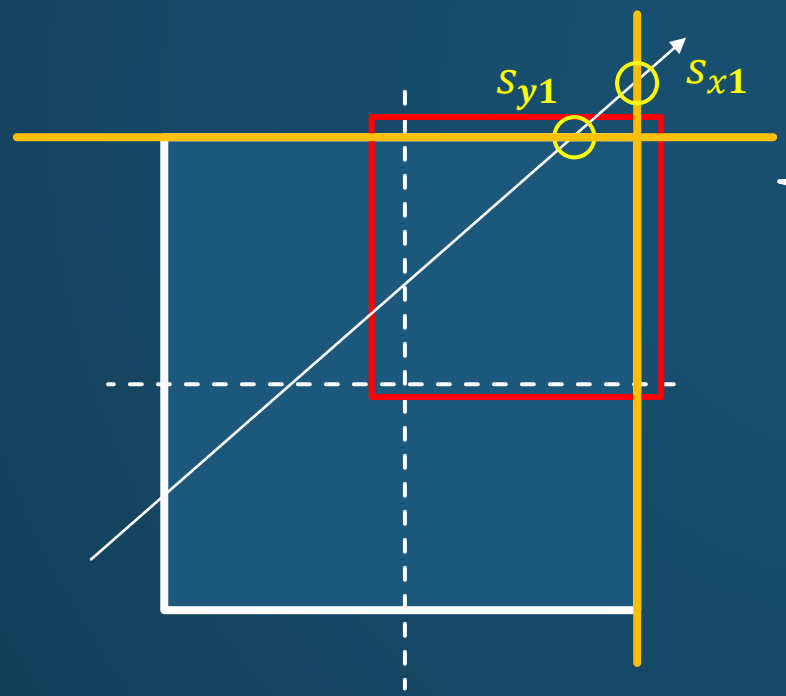


$S_{xmid} < S_{y1}$

The next is **X**

CHILD VOXEL TRAVERSAL

- What's the next node?
  - Choose the closest distance to the next planes



Either way, no children anymore. Finish traversal.

STATE TRANSITIONS?

- Table-based management is proposed in "An Efficient Parametric Algorithm for Octree Traversal
  - But it is not necessary in my experience because you can just keep the directions you have moved as bits

Current sub-node (state)	Exit plane YZ	Exit plane XZ	Exit plane XY
0	4	2	1
1	5	3	End
2	6	End	3
3	7	End	End
4	End	6	5
5	End	7	End
6	End	End	7
7	End	End	End

Table 3: State transitions.

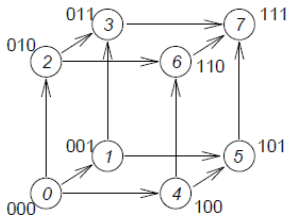


Figure 5: Sub-nodes that can be reached from an initial node.

```
for( ;; )
{
    // find minimum( x1, y1, z1 ) for next hit
    uint32_t mv =
        x1 < y1 ? ( x1 < z1 ? 1u : 4u ) : ( y1 < z1 ? 2u : 4u );

    // The move bit is yet empty, you can move to the direction
    bool hasNext = ( cur.childMask & mv ) == 0;

    ...

    // mark your movement
    cur.childMask |= mv;
```

“PUSH ALL HIT CHILDREN” VS “SAVE STATE AND GO DOWN”

- “Save state and go down” has multiple advantages
  - Less stack usage (maximum == depth of the octree)
  - The code is shorter as no need to reverse children to push
  - Only “moved bits” is the variable for state management

```
node = ...
while()
{
    hitChildren = node.findCrossedChildren()

    foreach( child in reverse( hitChildren ) )
    {
        push( child )
    }

    node = pop()
}
```

Push all hit children

```
node = ...
while()
{
    next = node.findNextCrossedChild()

    if( node.mayHaveNext() )
    {
        push( node )
    }

    if( next is valid )
    {
        node = next
    }
    else
    {
        node = pop()
    }
}
```

Save state and go down

"PUSH ALL HIT CHILDREN" VS "SAVE STATE AND GO DOWN"

- "Save state and go down" has multiple advantages
  - Less stack usage (maximum == depth of the octree)
  - The code is shorter as no need to reverse children to push
  - Only "moved bits" is the variable for state management

One approach to using a stack for the search would be to push information regarding all intersected child nodes (between 1 and 4) for any one parent, entering these in descending order of distance. The top node could then be popped and used for the next level of search, and so on. However, a more efficient approach is to store status information for each parent node on the stack, so whenever a parent is popped, either the next child hit is derived and the parent's status is updated, or if all its children are exhausted, the parent is removed from the stack. In order to implement this operation, the index to MASKLIST should also be stored on the stack, together with a pointer to the previous child's s_{umin} (to be used as the next child's s_{lmax}). Each stack node therefore contains the following:

```
    node = po
}
```

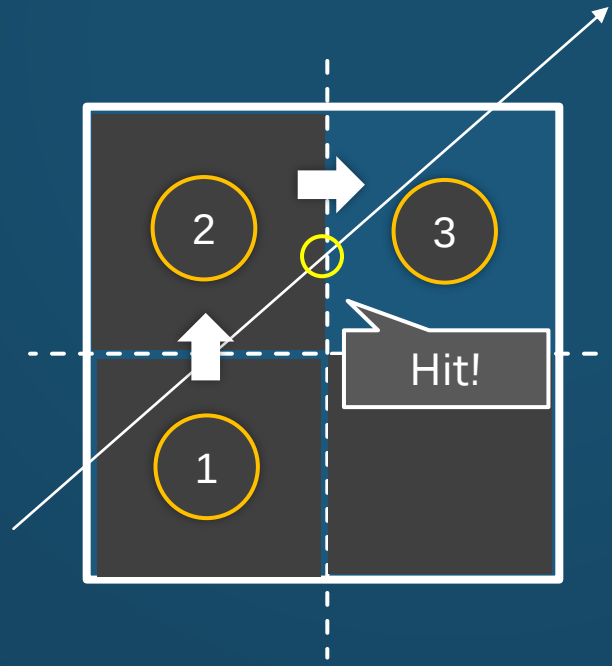
Sec. 4 in "The HERO Algorithm for Ray-Tracing Octrees"

```
node = ...
while()
{
    next = node.findNextCrossedChild()

    if( node.mayHaveNext() )
    {
        push( node )
    }

    if( next is valid )
    {
        node = next
    }
    else
    {
        node = pop()
    }
}
```

Save state and go down

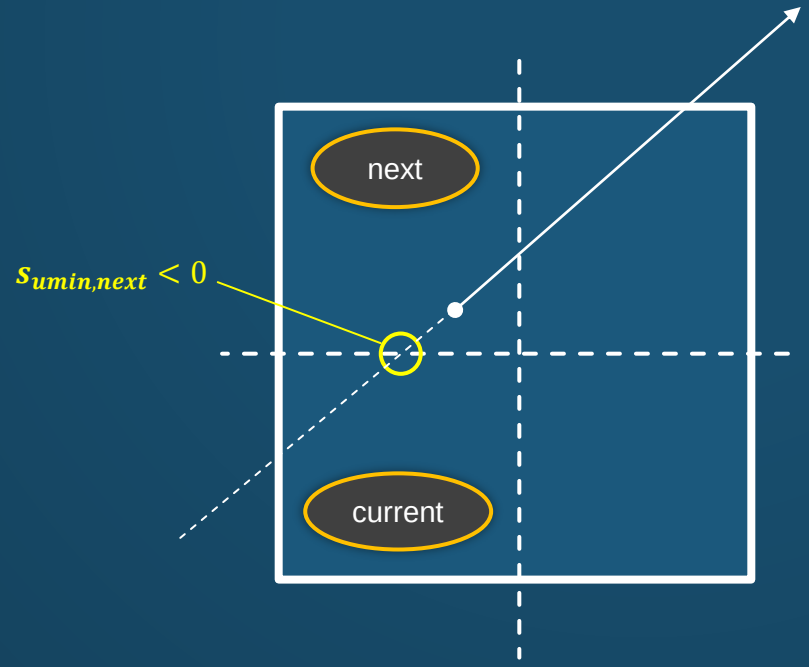
A TIP: WHEN YOU CAN STOP TRAVERSAL

- The traversal order is always ideal
  - You can stop traversal immediately after you find a voxel in the deepest level☺
  - Pushed node in the stack can be totally skipped



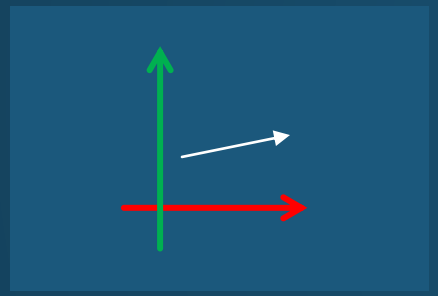
A TIP: BACK VOXEL SKIP

- Voxels behind the ray can be skipped
  - When the next $s_{umin,next}$ is negative, the current voxel is totally behind
  - You don't have to traverse into the voxel ☺

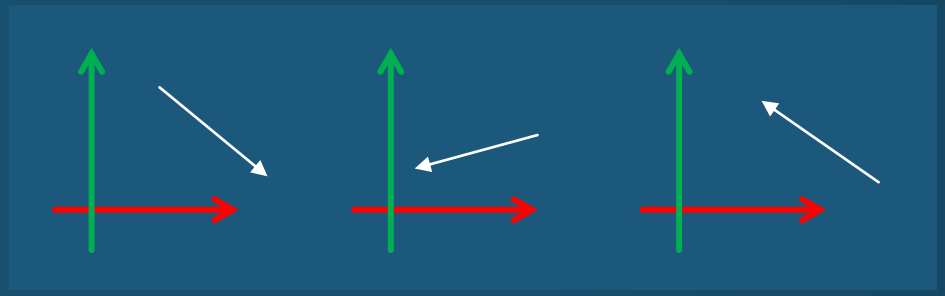


GENERALIZING FOR RAYS WITH NEGATIVE DIRECTIONS

- The traversal algorithm so far assumes all components of ray direction d are positive
- How to handle negative rays?



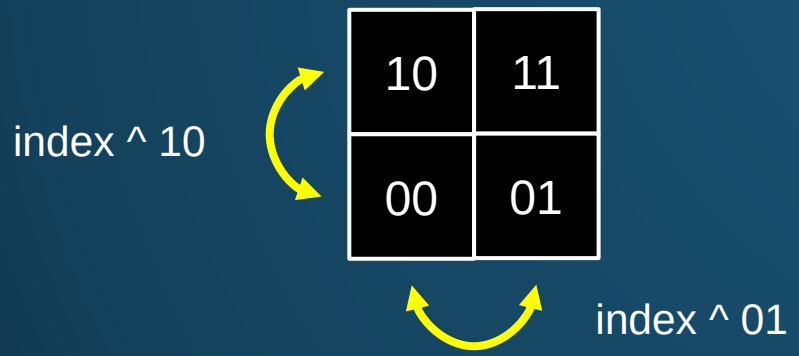
positive



negative

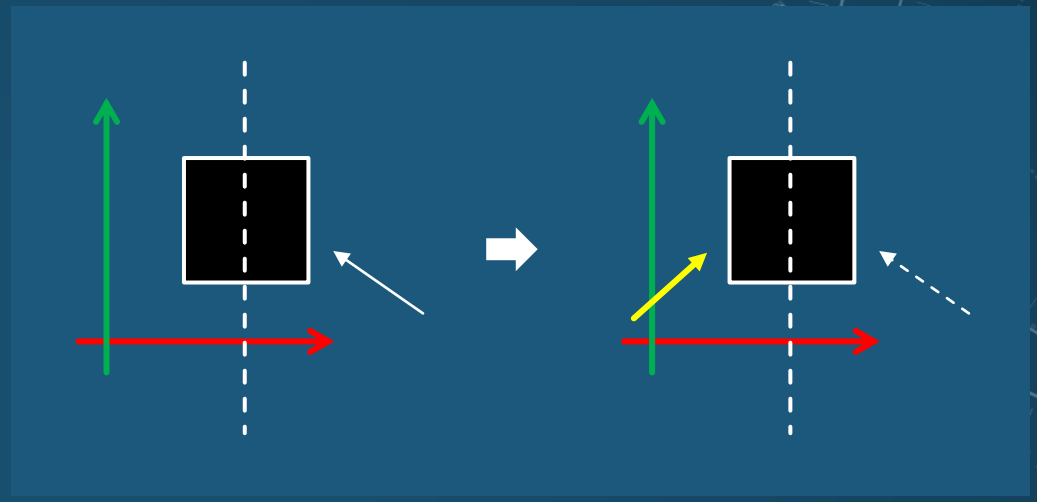
MIRRORING

- Mirror image around the center of the octree
  - Do it for each axis
- What about children's order?
  - They can be reordered by XOR bit operation
  - Just apply "flip bits" to the child index

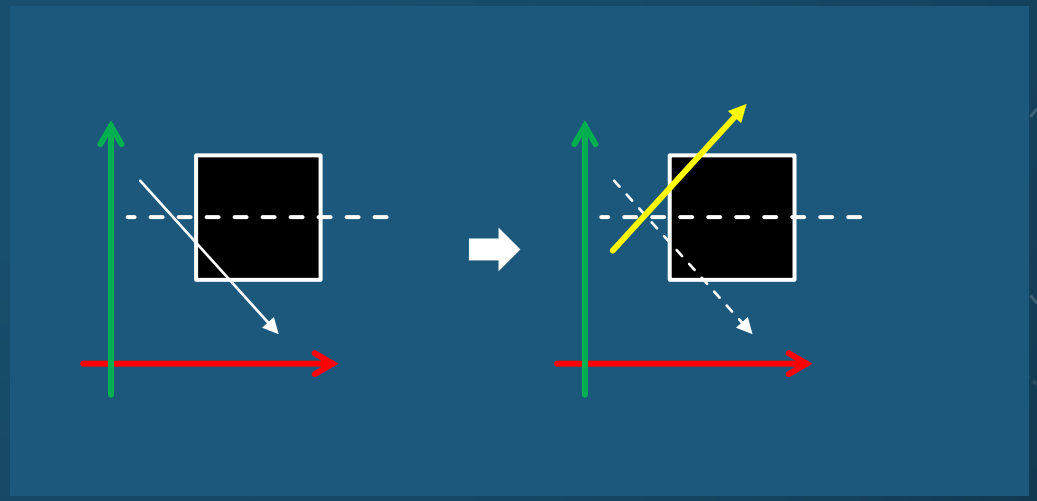



```
... = node.children[ index ^ flipBits ];
```

X flip



Y flip



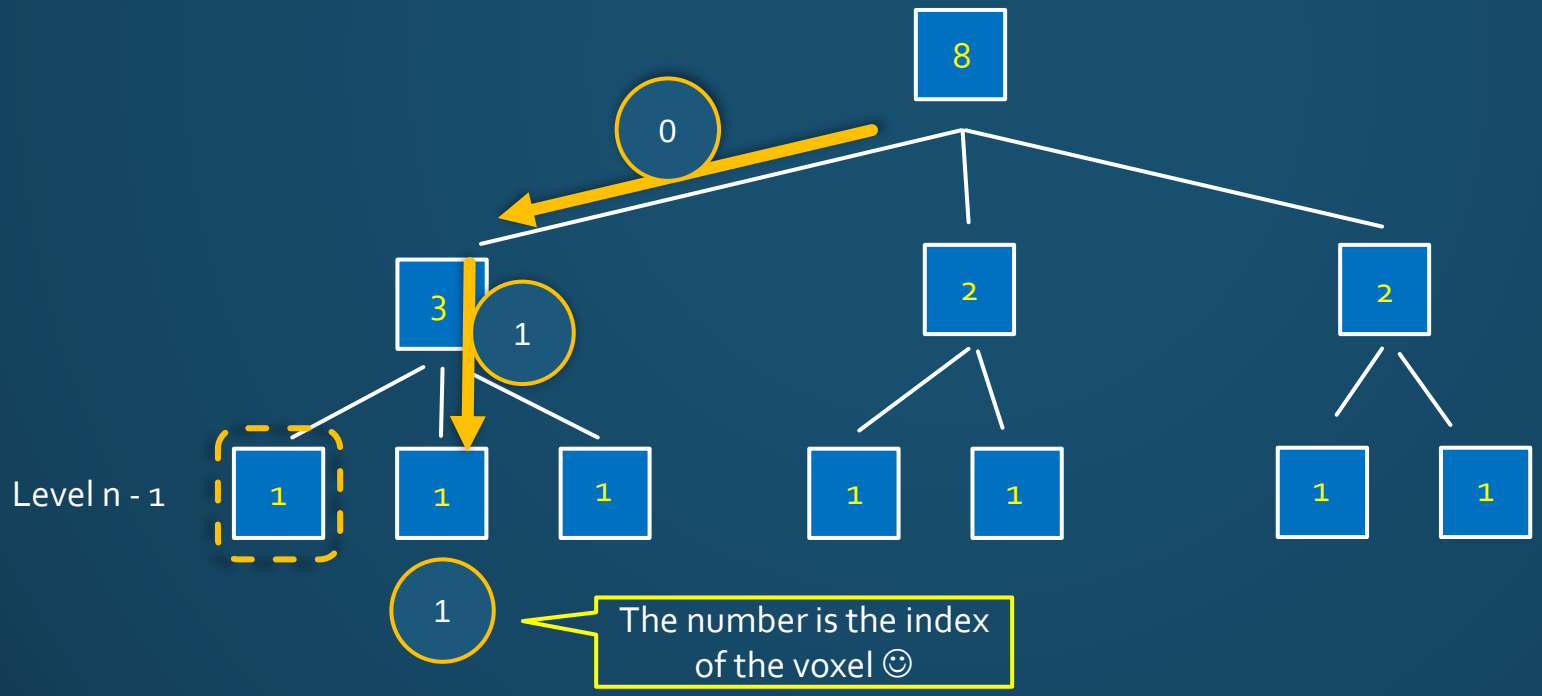
Voxel Attribute Support

VOXEL ATTRIBUTE SUPPORT

- DAG compression is great
  - Simple bottom-up compression
  - Significantly reduce memory
- … but what about colors?
  - Attributes per voxels can be useful for shading
  - DAG Octree shares redundant nodes but it is hard to share colored nodes because of spatially varying values
- A paper elegantly resolved
  - Dan Dolonius, Erik Sintorny, Viktor Kampezand Ulf Assarsson, "Compressing Color Data for Voxelized Surface Geometry"

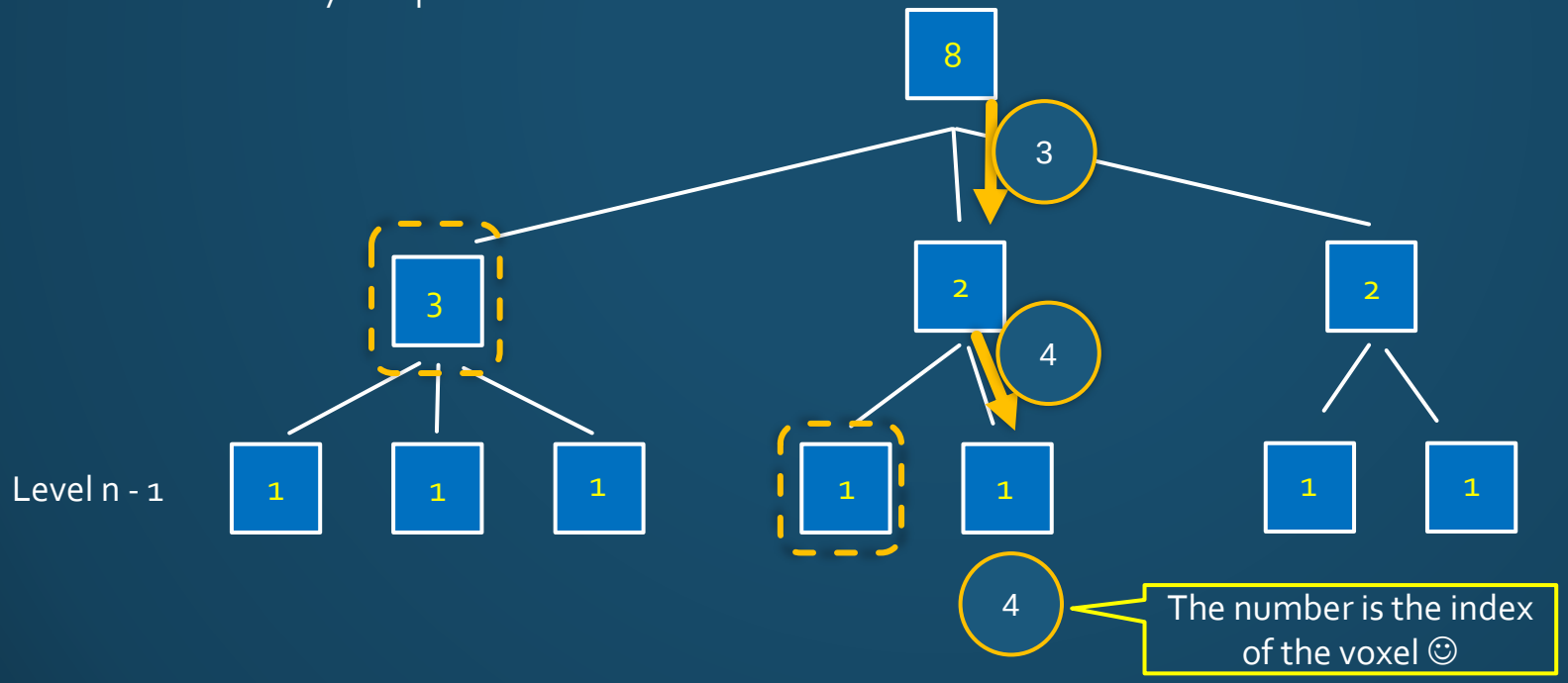
DYNAMIC VOXEL INDEXING

- Embed "the number of voxels in children" to each node
  - Count all left children's voxels when you traverse



DYNAMIC VOXEL INDEXING

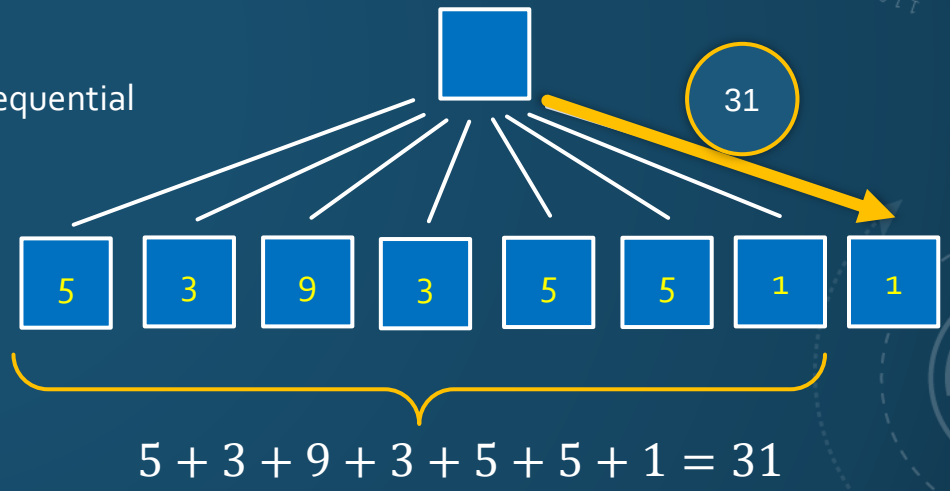
- Embed "the number of voxels in children" to each node
  - Count all left children's voxels when you traverse
  - Number of Voxels can be shared unlike colors in DAG
  - Attribute data is totally independent from Octree



DYNAMIC VOXEL INDEXING

- The paper mentioned the padding bits on an octree node can be used
- However, there are a few disadvantages to this approach
  - The total number of voxels has to be less than 2^{24}
  - You have to count all of the left children during the traversal
    - It can not be cached in the stack because traversal order is not sequential
    - It requires fetching data from children
      - It is expensive specifically on the GPU

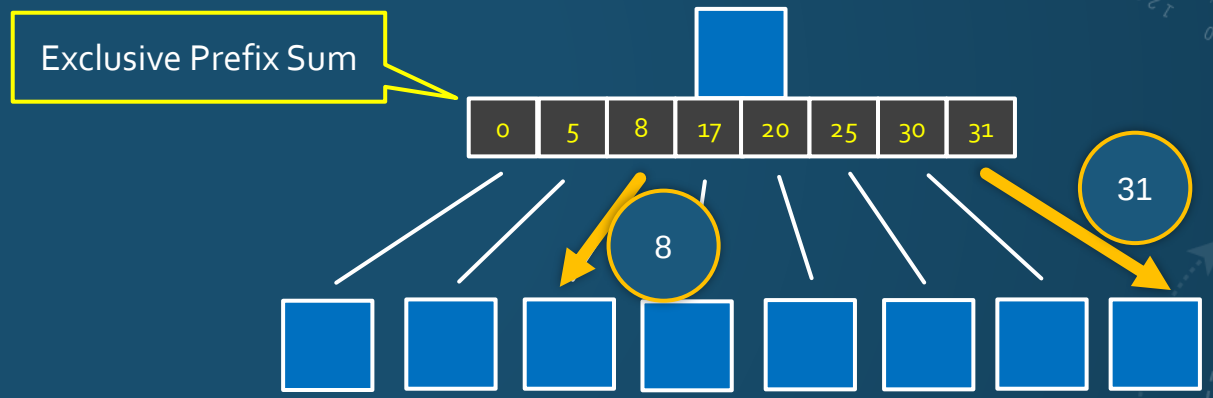
```
struct OctreeNode
{
    uint8_t mask;
    uint8_t pads[3];
    uint32_t children[8];
};
```



ALTERNATIVE APPROACH

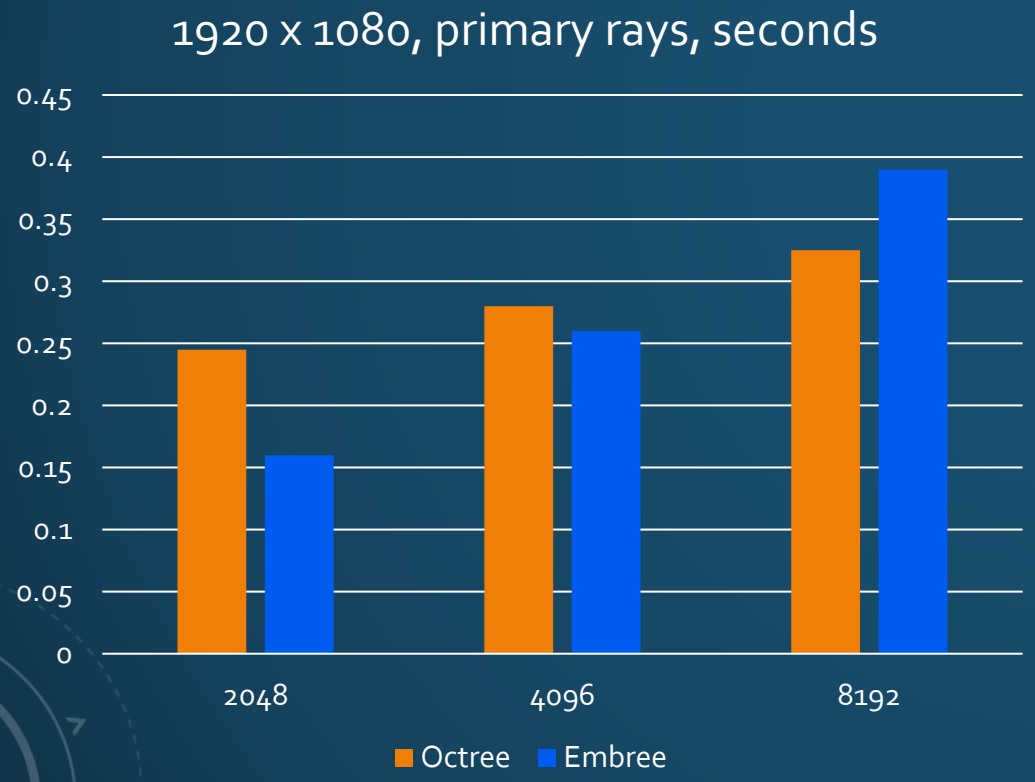
- Prefix Sum can be stored in the parent node
  - Counting is replaced to look up the table ☺
  - No memory access to the children☺
  - It almost doubles octree memory ☹
    - But it is a trade-off

```
struct OctreeNode
{
    uint8_t mask;
    uint32_t children[8];
    uint32_t nVoxelsPSum[8];
};
```



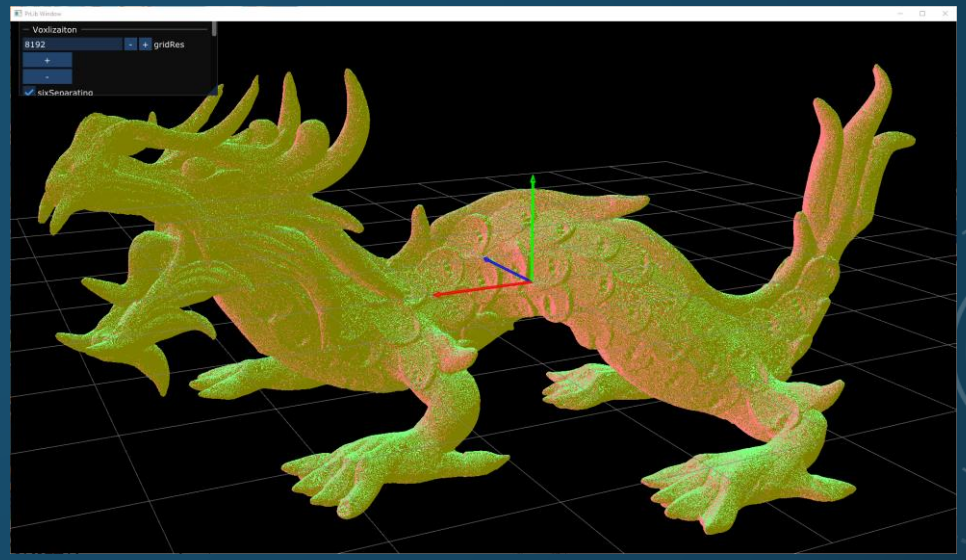
PERFORMANCE - CPU

- $2048^3, 4096^3, 8192^3$ Resolutions
- vs Embree user geometry, single threaded



Asian Dragon(8192^3)

- 8192^3 Resolution
- 55M voxels

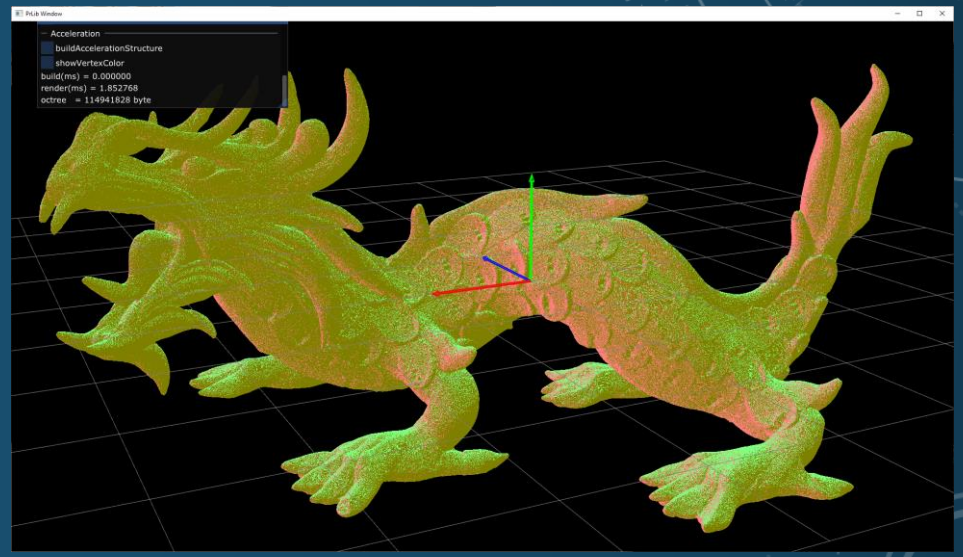


PERFORMANCE - GPU

- Primary Ray Casting (1920 x 1080)
- RX 7900 XTX, RTX 3090 Ti
  - 2 ms for primary for both

Asian Dragon(8192^3)

- 8192^3 Resolution
- 55M voxels

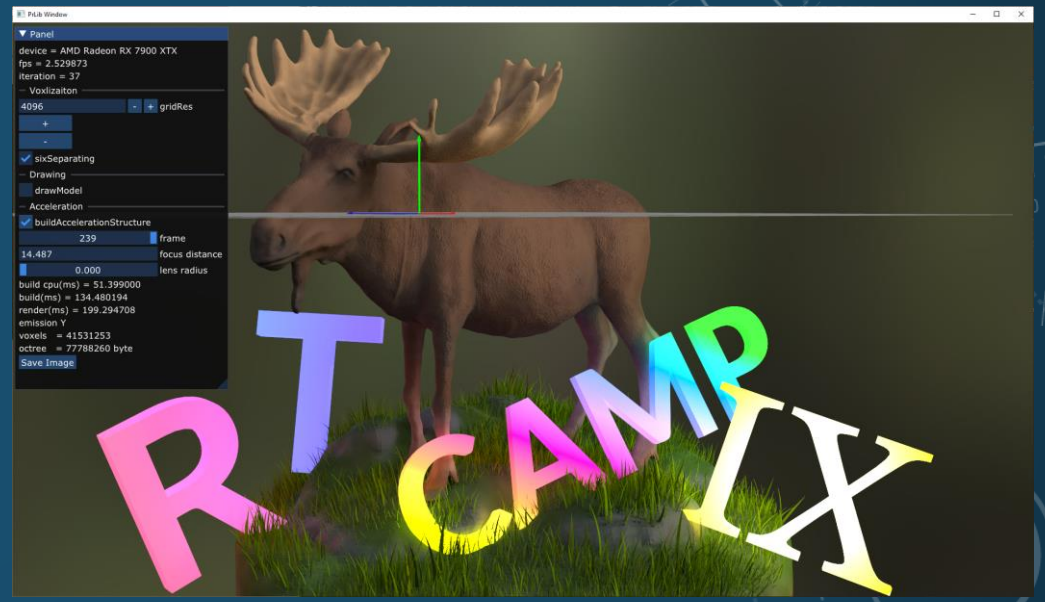


PERFORMANCE - GPU

- 8 bounce diffuse GI (1920 x 1080)
  - IBL shadow rays
  - No shadow rays for emissive voxels
- RX 7900 XTX
  - ~ 200 ms / 16 spp (12.5 ms / sample)
  - ~ 135 ms for voxelization and build octree

RT Camp Scene(8192^3)

- 4096^3 Resolution
- 41M voxels



CONCLUSION

- Tons of amazing papers about voxels!
  - Cool ideas
    - Voxelization, octree construction, ray traversal
  - Easy to implement, efficient execution
  - Gives us a lot of new insights
- Is triangle really for you?