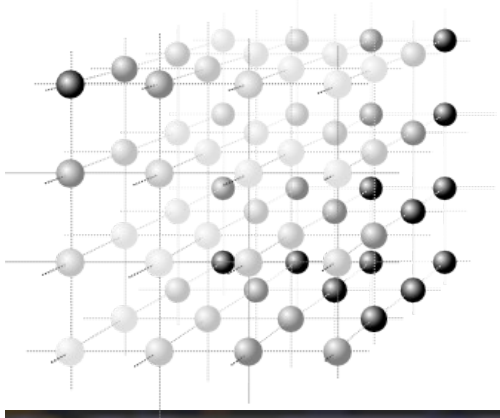
GPU Voxelizer

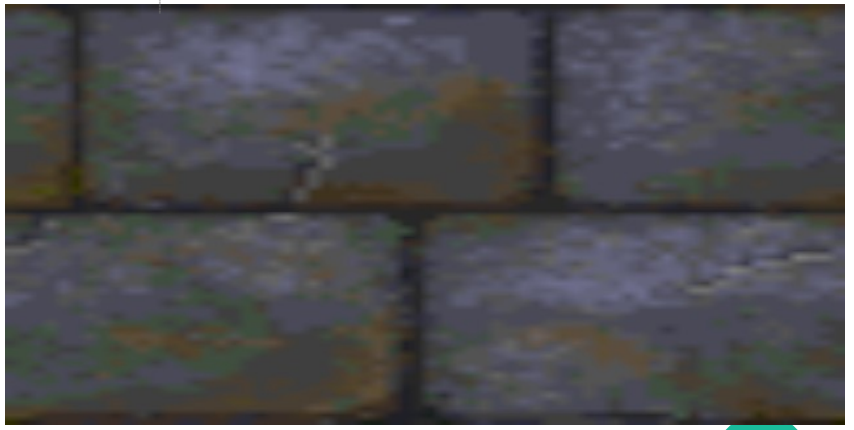
?

Pt.1: Background & data structures

Background

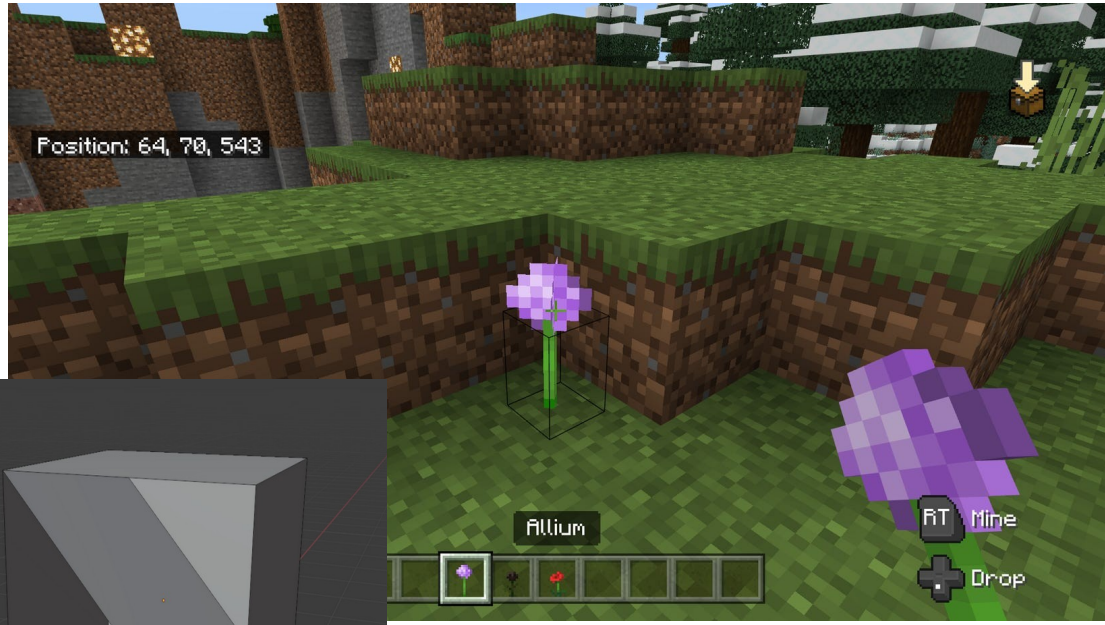
- Voxels are just the 3D equivalent of pixels
- But! Pixels are not little squares, voxels are not little squares
- Instead are evenly spaced points of data
- Can be interpreted in a variety of ways, interpolated, stretched etc
- Locations of pixels/voxels are almost always implictly encoded

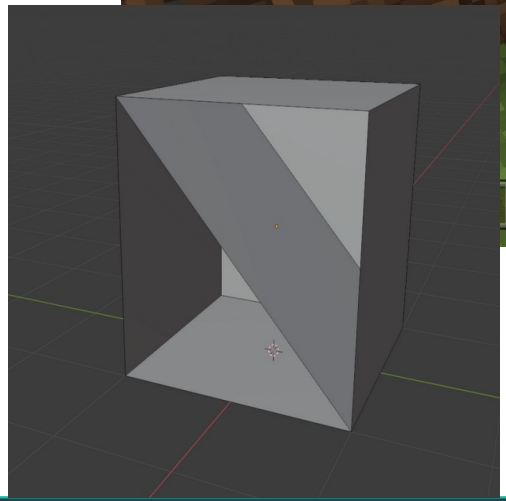




Voxel Uses: Games

- Minecraft obviously the most famous example
- Good example of non-cube voxels (flowers etc)
- Voxels are very large (1m^3)
- Incredibly easy to build stuff compared to 3D modeling software
- Right-click to place, left-click to delete
- Impossible to have non-manifold geometry



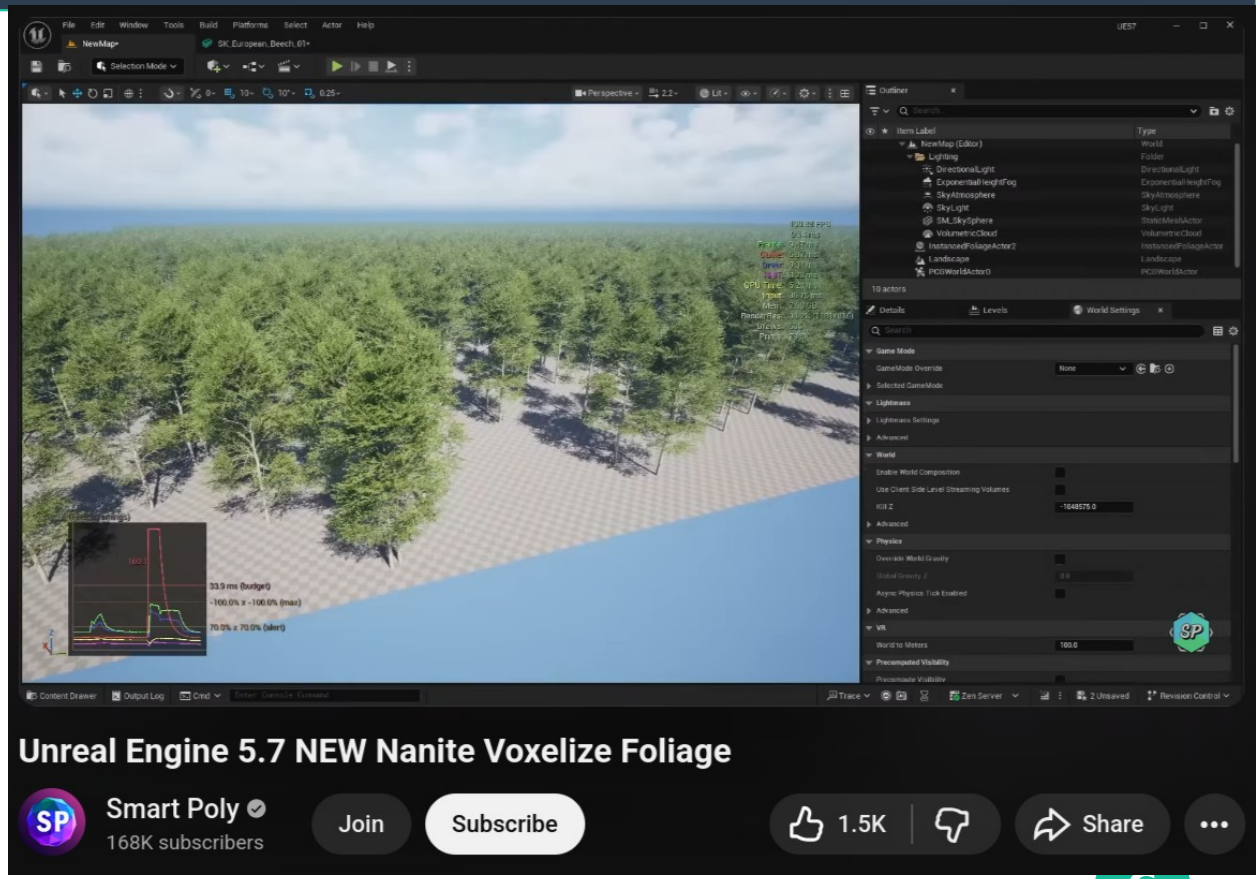


Voxel Uses: Games Cont.

- Teardown by Tuxedo Labs
- Voxels are a lot smaller!
- Voxel geometry allows for large-scale destruction and physics without having to split triangle meshes somehow

Voxel Uses: Games Cont.

- Unreal Engine 5.7+ has a mode to voxelize distant folliage
- Avoids a tradeoff between distant trees looking ‘thin’ due to having many transparent areas vs bad performance

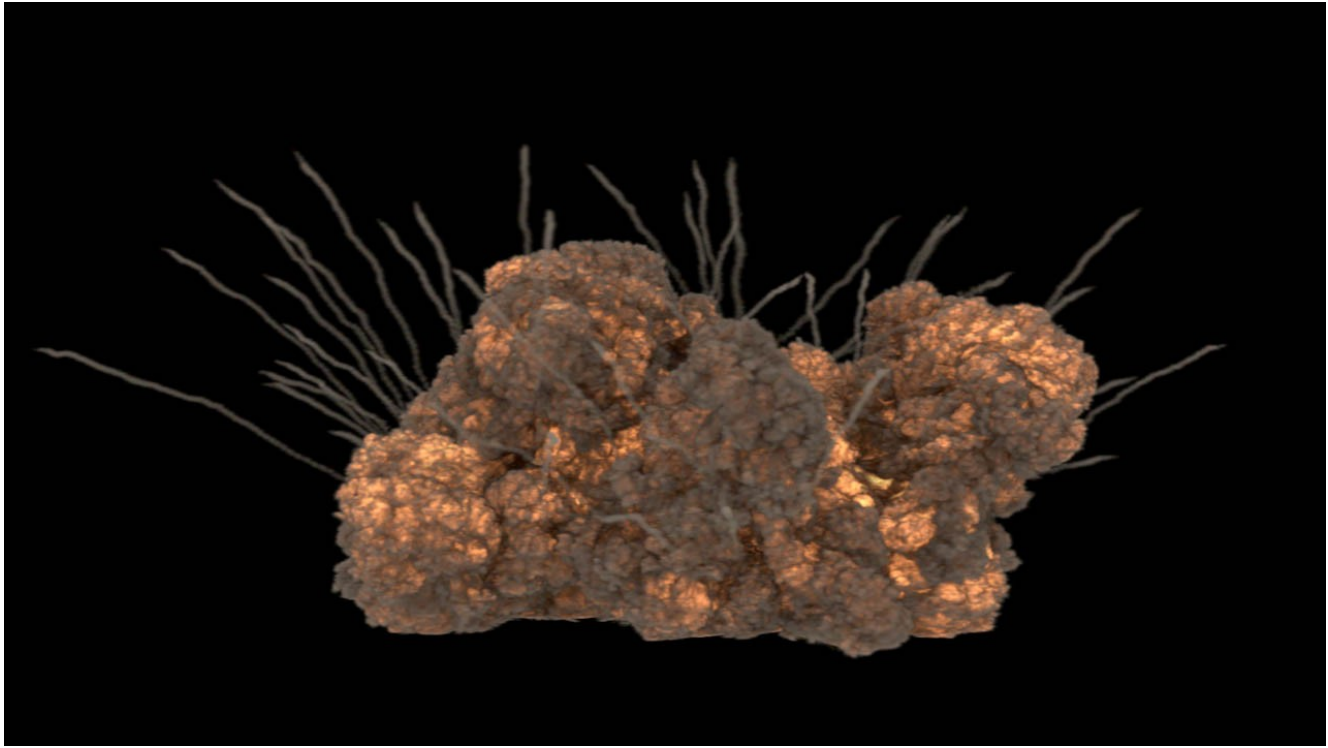


Voxel Uses: Games Cont.

- This guys video came onto my feed
- Using voxels to cache signed distance field data
- Check the video out :)

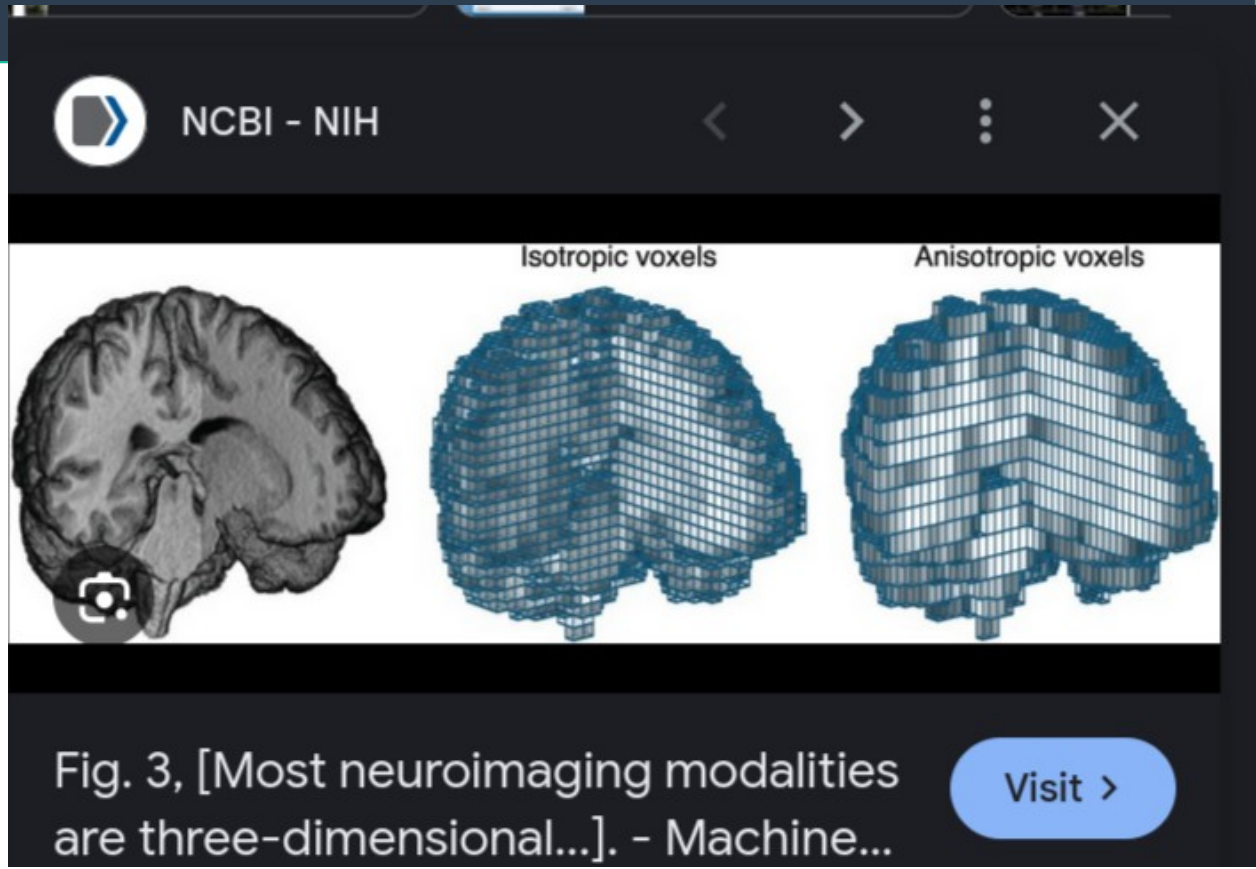
Voxel Uses: VFX

- Used for fire/smoke/cloud sims in vfx
- Voxels store density and temperature



Voxel Uses: MRIs

- **Brain scans and stuff**



Voxel Storage

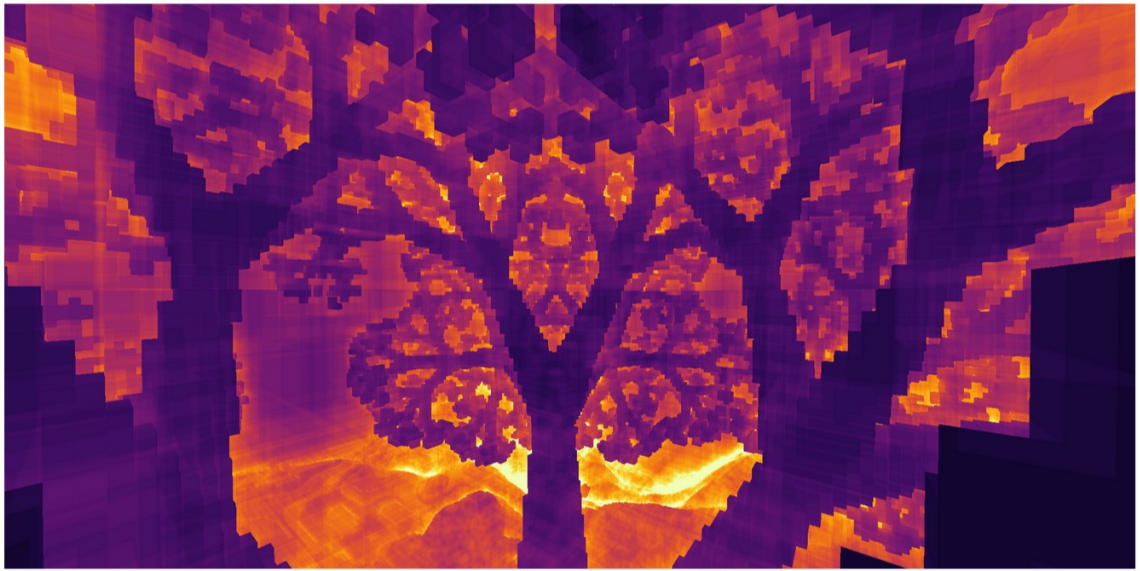
- **Images made of pixels are fairly easy to manage**
  - Just need a width and list of pixel data
  - Height can be derived as len(data)/width
- **Same can be applied to voxels**
  - Depth = len(data)/width/height
  - But storage requirements are cubic – go up very fast
  - Unlike pixel images, most voxel scenes in games will consist of a large amount of empty space – a uniform voxel value
  - Can exploit this

Voxel Storage – Cont.

- **Sparse Voxel Octrees (SVOs) are a common voxel data structure**
- **Tree structure where each node has up to 8 children**
- **Voxel positions are implict – defined by location in tree**
- **Large empty spaces are indicated by lack of nodes**
- **Could go into the implementation, except I found this blog post on 64-trees**

A guide to fast voxel ray tracing using sparse 64-trees

Oct 3, 2024



64-Trees

```rust
#[repr(C, packed)]
pub struct Node {
    pub is_leaf_and_ptr: u32,
    pub pop_mask: u64,
}

impl Node {
    pub fn new(is_leaf: bool, ptr: u32, pop_mask: u64) -> Self {
        Self {
            is_leaf_and_ptr: (ptr << 1) | (is_leaf as u32),
            pop_mask,
        }
    }
}
```

```rust
pub struct Tree64<T> {
    pub nodes: Vec<Node>,
    pub data: Vec<T>
}
```

- Less overhead and better performance characterics than a standard SVO
- Each node contains a 4^3 block of child nodes
- Consists of a pointer and 64-bit mask (+boolean)
- pop_mask.count_ones() gives the number of children total
- Child nodes are tightly packed at the ptr location
- Leaf nodes point to `data` instead of other nodes

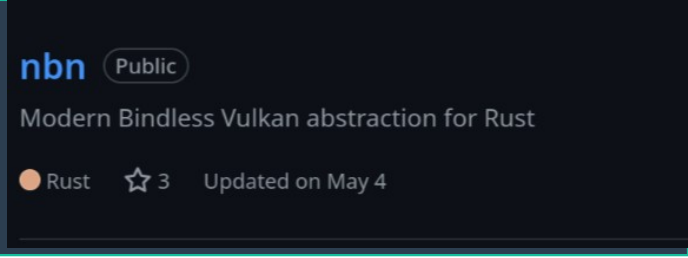
Rendering

- Lots of voxel engines generate triangle meshes for rendering
- Can be done efficiently-ish, but is tricky to pull off
- Meanwhile, voxel structure is heirarchical in a way that makes it perfect for ray tracing
- Ray traced ‘in software’ without using the Hardware RT APIs.
- I just copied the code from the 64-tree blog post guy and adapted it for webgpu

Pt. 2: Voxelizing 3D Triangle Meshes

?

GPU Voxelization – Host Code




```rust
let gltf = device.create_buffer_with_data(nbn::BufferInitDescriptor {
    name: "gltf",
    data: &[Gltf {
        buffer_views: *buffer_views,
        accessors: *accessors,
        buffer: *buffer,
        primitives: *primitives,
        materials: *materials,
        num_primitives: primitives_iter().count() as u32,
    }],
});

let voxelization_shader = device.load_shader("shaders/compiled/voxelize.spv");

let voxelization_pipeline = device.create_graphics_pipeline(nbn::GraphicsPipelineDesc {
    vertex: nbn::ShaderDesc {
        module: &voxelization_shader,
        entry_point: c"vertex",
    },
    fragment: nbn::ShaderDesc {
        module: &voxelization_shader,
        entry_point: c"fragment",
    },
    color_attachment_formats: &[],
    conservative_rasterization: true,
    depth: Default::default(),
});
```

- Basic idea: make use of the standard GPU triangle rasterization pipeline
- Uses my Vulkan abstraction library
- Initial loading of 3D models, textures etc as per usual
- Models are rasterized in chunks to avoid running out of laptop GPU memory

GPU Voxelization – Vertex Shader Code

```
uint8_t swizzle;
if (normal.z == max) {
    // do nothing
    swizzle = 0;
} else if (normal.y == max) {
    swizzle = 1;
    position = position.xzy;
} else {
    swizzle = 2;
    position = position.yzx;
}
```

- Fairly ordinary
- Triangle vertex positions are ‘swizzled’ so they’re always rasterized from a view with the largest area
- E.g. flat ground triangles are rasterized top-down and not from the side

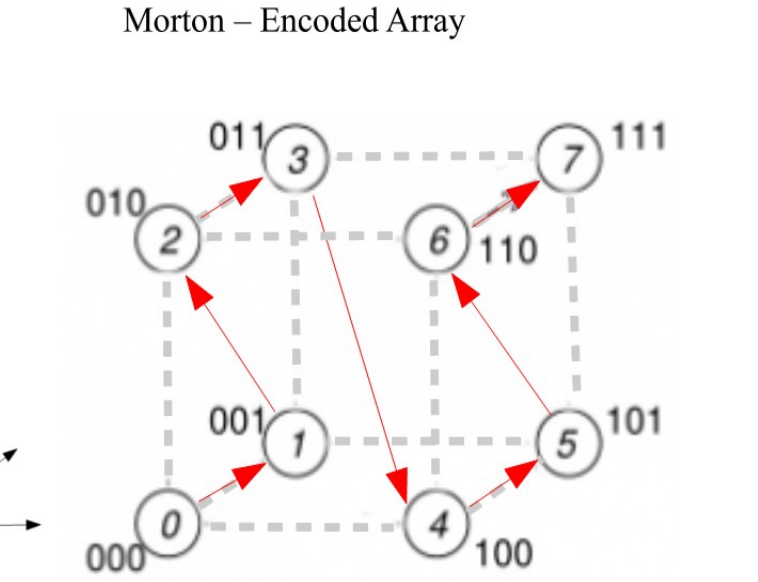
GPU Voxelization – Pixel Shader Code

```
reinterpret<uint, uint8_t4>(uint8_t4(
    linear_to_srgb(base_colour),
    uint8_t(is_emissive) | (uint8_t(is_metallic) << 1)
        | (posterize(aux, 6) << 2)
))
```

```
let output_index = push_constants.num_outputs.add(1);
if (output_index < push_constants.output_size) {
    push_constants.output[output_index] = ...
}
```

- Each rasterized pixel = one voxel
- Swizzle is undone
- Position in 3D calculated via depth
- Textures are sampled to get material data
- Material information is packed into 32 bits:
  - 24 bits for RGB base colour
  - 2 bits for type (metallic,non-metallic,emissive)
  - 6 auxilary bits (surface roughness or emissive power)
- Instead of writing to an attached texture, output voxel values + position are pushed onto a a list using an atomic add

GPU Voxelization - Sorting



- Voxel positions are morton encoded – the bits of each field are interlaced:

XXXXYYYYZZZZ -> XYZXYZXYZXYZ

- The voxel list is read back to the CPU
- Radix sort is used to sort the voxels by position – much faster than standard sort for ints
- Because the voxels are sorted by morton code, the voxels of each 4^3 chunk are grouped together
- Generating a list of leaf nodes for the voxels just becomes a linear scan
- Leaf nodes are also sorted via morton encoding, so collecting the nodes that point to them is also a linear scan!
- Keep going until theres a single root node

Fin :)