SIMD is very much an underrated technique to massively increase an application's performance. It stands for same-instruction-multiple-data, and applying this to voxels is intuitive. Usually doing so requires using extended instruction sets (eg. SSE or AVX) but there is another approach when working on a bit level.

This article describes how to apply the same principles of SIMD to cell mask building. Aside from sampling, this is likely to be the slowest part in any algorithm due to the cold fetches from neighbor lookups.

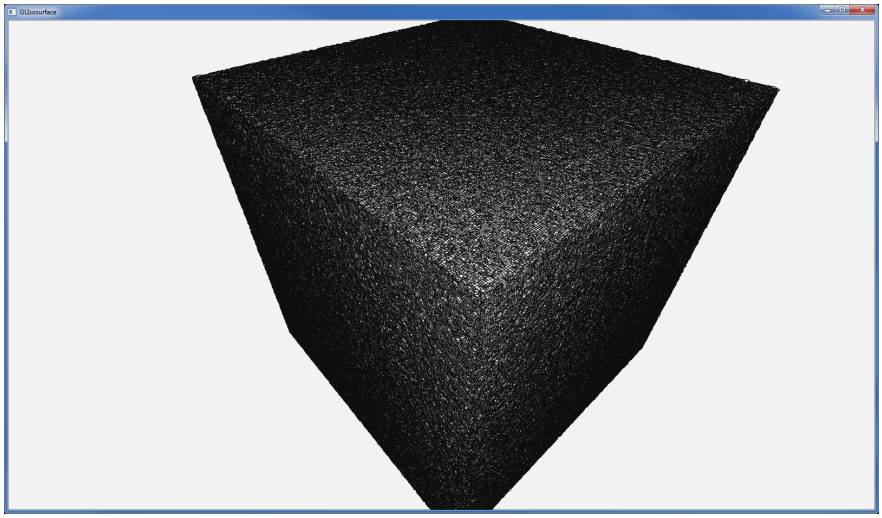

Figure 1: 256^3 white noise that produced 16 million vertices and 49 million triangles. Building cell masks took 143ms in a single AMD FX-8350 CPU thread.

In order to take advantage of this technique, we first have to store the signs of our grid at each corner in a linear order along the Z axis in a way that our stride will always contain a linear group of data. This often means the resolution is a multiple of a word (often 32 or 64 bits) and the number of cells is one less than the resolution. In this case, it means the resolution has to be a multiple of the number of bits in a word, with the cells following the minus-one rule.

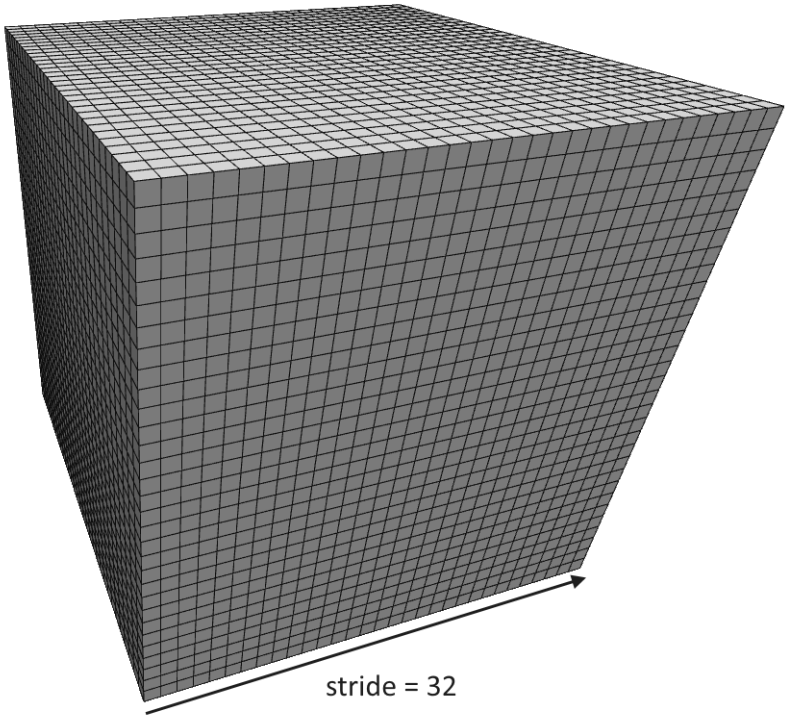

Figure 2: Visualization of a chunk with a stride of 32. Notice how it contains 31 cells.

To fetch a row of corner samples, we feed in an X and Y coordinate, optionally feeding in a Z coordinate to get the specific bit we're looking for. This might look like:

```
uint32_t encode_vertex(uint32_t x, uint32_t y, uint32_t z, const uint32_t z_per_y, const uint32_t y_per_x, uint32_t& out_mask, const uint32_t stride)
{
        uint32_t index = (z / stride) + (y * z_per_y) + (x * y_per_x);
        out_mask = 1 << (z % stride);
        return index;
}
```

Where
• x, y, z - the coordinates to be looked up
• z_per_y - how many strides fit in a row of Zs
• y_per_x - how many strides fit in a row of Zs and Ys
• stride - the amount of bits in a stride
• out_mask - where the specific bit mask will be written to
• index - the index in the array of data to lookup[/list]

Building the array of data is trivial, and is up to the user to decide to convert the data they are given. To operate on the data and build the cell masks, we start by grouping the cell masks into words linearly along Z as well. The order for individual cell masks also needs to be defined in a way where Z depends on the first bit, with Y and X following after. Building the masks involves 4 passes along the Z axis - once for each Z edge of a cell. Each pass iterates over the entirety of X and Y cells, and however many strides fit in a row of Z's (previously defined as *z_per_y*). The current stride is

referred to as *z_block*. The relative index within the stride is referred to simply as *z*, whereas the absolute Z is referred to as *total_z*.

The idea is to do as many operations on a chunk of data as possible without doing any fetches. The 4 iterations guarantees that for each pass, we can operate on <stride> blocks at a time with only 2 fetches since everything is in order. The word size is likely to be 64 bits and a cache line usually contains 64 bytes, so one 64-bit fetch followed by 8 64-bit fetches guarantees we get everything within those fetches.

For each pass, we also define a mask offset that controls which bits are set in the cell masks. This correlates with the edge and typically formats as follows:

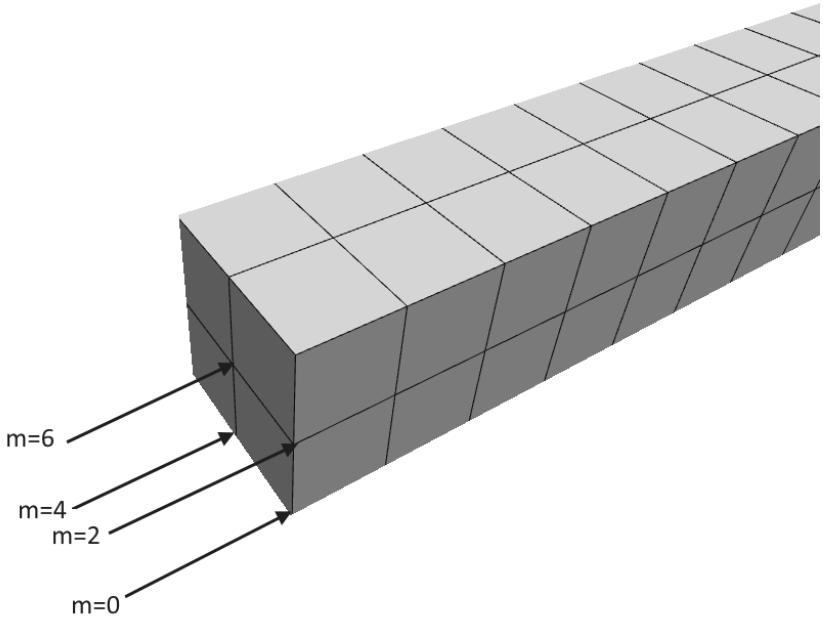

Figure 3: Visualizing the mask offset as m.

As demonstrated by the previous figure, we also have to offset the Y and X axis of our samples by 1 depending on which pass we're in. Following the mask offset order, this means (0, 0), (0, 1), (1, 0), and (1, 1).

The next step is to actually build the cell masks using the data. Each grid sign read involves writing to two cells - the current cell being iterated and the one before it. This requires special handling when at boundaries. We only write to the cell prior if total_z is greater than 0, and we only write to the current cell if total_z is less than the number of cells in an entire Z axis.

Inside each pass, in order to minimize writes to main memory, we want to work with local variables as long as possible. Within each z_block, we create an array of words representing temporary masks being written to that accounts for the amount of cells within a z_block. The only thing left to do at this point is build the local masks, paying careful attention to boundaries. This can be done inside a loop, or for optimal performance an unwrapped one.

Inside the first pass, we always directly assign the global masks to the local ones so as to avoid having to initialize the global masks to 0. For the other passes, we perform bitwise OR if there are any changes.

A full pass on the final Z axis where X and Y offsets are both one might look like this:

```
for (uint32_t x = 0; x < dim; x++)
{
        for (uint32_t y = 0; y < dim; y++)
        {
                const int m = 6;
                for (uint32_t z_block = 0; z_block < z_count; z_block++)
                {
                        uint32_t line = samples[(x + 1) * y_per_x + (y + 1) * z_per_y + z_block];
                        if (line == 0)
                                continue;
                        uint64_t* __restrict line_masks = masks + x * y_per_x8 + y * z_per_y8 + z_block * 4;
                        uint64_t local_masks[4] = { 0, 0, 0, 0 };
                        EDGE_LINE(0, 0, 0x1);
                        EDGE_LINE(0, 1, 0x2);
                        EDGE_LINE(0, 2, 0x4);
                        EDGE_LINE(0, 3, 0x8);
                        EDGE_LINE(0, 4, 0x10);
                        EDGE_LINE(0, 5, 0x20);
                        EDGE_LINE(0, 6, 0x40);
                        EDGE_LINE(0, 7, 0x80);
                        EDGE_LINE(1, 0, 0x100);
                        EDGE_LINE(1, 1, 0x200);
                        EDGE_LINE(1, 2, 0x400);
                        EDGE_LINE(1, 3, 0x800);
                        EDGE_LINE(1, 4, 0x1000);
                        EDGE_LINE(1, 5, 0x2000);
                        EDGE_LINE(1, 6, 0x4000);
                        EDGE_LINE(1, 7, 0x8000);
                        EDGE_LINE(2, 0, 0x10000);
                        EDGE_LINE(2, 1, 0x20000);
                        EDGE_LINE(2, 2, 0x40000);
                        EDGE_LINE(2, 3, 0x80000);
                        EDGE_LINE(2, 4, 0x100000);
                        EDGE_LINE(2, 5, 0x200000);
                        EDGE_LINE(2, 6, 0x400000);
                        EDGE_LINE(2, 7, 0x800000);
                        EDGE_LINE(3, 0, 0x1000000);
```

```
                        EDGE_LINE(3, 1, 0x2000000);
                        EDGE_LINE(3, 2, 0x4000000);
                        EDGE_LINE(3, 3, 0x8000000);
                        EDGE_LINE(3, 4, 0x10000000);
                        EDGE_LINE(3, 5, 0x20000000);
                        EDGE_LINE(3, 6, 0x40000000);
                        EDGE_LINE(3, 7, 0x80000000);
                        if (local_masks[0] != 0)
                                line_masks[0] |= local_masks[0];
                        if (local_masks[1] != 0)
                                line_masks[1] |= local_masks[1];
                        if (local_masks[2] != 0)
                                line_masks[2] |= local_masks[2];
                        if (local_masks[3] != 0)
                                line_masks[3] |= local_masks[3];
                }
        }
}
```

```
#define EDGE_LINE(l_z, z, b)
if(line & b) {
if (z_block < z_count - 1 || l_z * 8 + z < 31)
        local_masks[l_z] |= 1ull << (z * 8 + m);
if (z > 0)
        local_masks[l_z] |= 1ull << ((z - 1) * 8 + m + 1);
else if (l_z > 0)
        local_masks[l_z - 1] |= 1ull << (7 * 8 + m + 1);
else if (z_block > 0)
        line_masks[-1] |= 1ull << (7 * 8 + m + 1);
}
```

In the code sample, rows of signs have a stride of 32 whereas the masks are stored in 64 bits. This is to allow a minimum chunk resolution of 32 rather than 64. The *z_per_y* and *y_per_x*variables are also post-fixed by 8 to denote them being divided by 8 ahead of time to account for combining masks into a 64-bit word.

Another key point is that when writing to a previous cell when z_block is at least one but z is zero, we have to directly write to the main memory since the local masks do not represent that cell. This only happens a small handful of times so the performance impact is minimal.

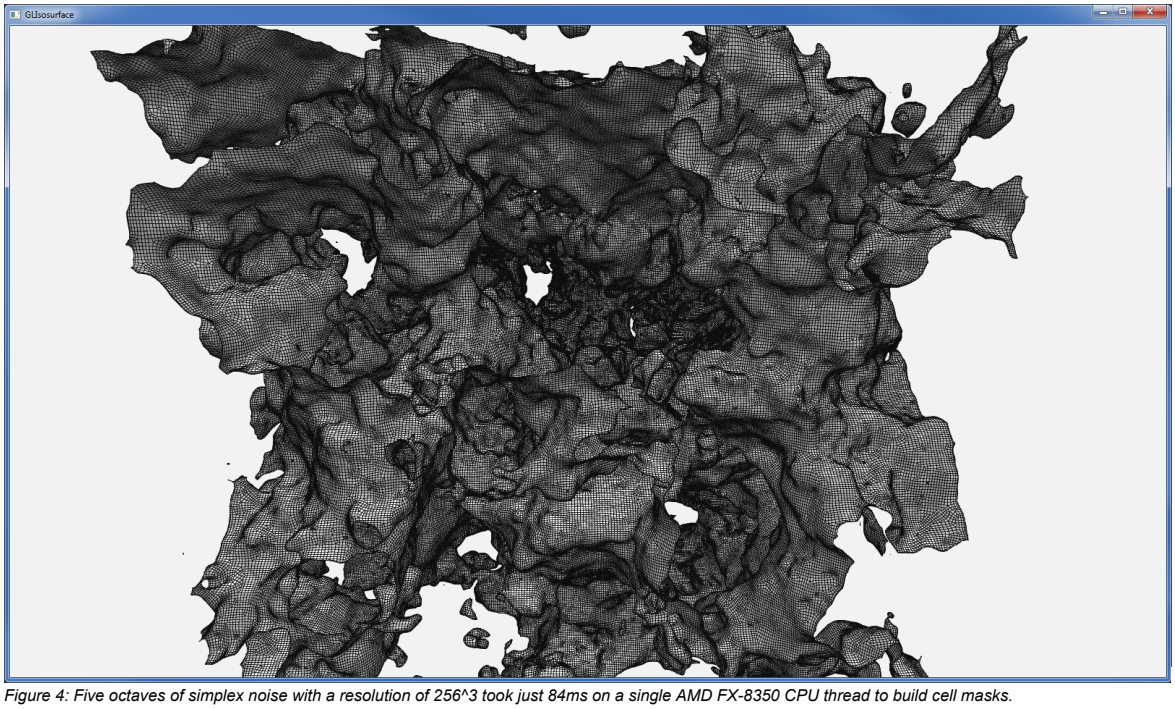

Figure 4: Five octaves of simplex noise with a resolution of 256^3 took just 84ms on a single AMD FX-8350 CPU thread to build cell masks.

Future work involves dealing with multiple materials, since sign classifications may not be binary in some cases. The ideas presented here still apply, but the bits-per-corner increases so some modification is necessary.

Special thanks to realz for discussing this optimization and helping to make it as fast as possible.