Examination of the advantages and disadvantages of voxel greedy meshing on the GPU for voxel based game engines

Niklas Mäckle
Hochschule der Medien
Stuttgart, Germany
nm067@hdm-stuttgart.de

***Abstract*—This paper proposes to leverage the power of parallel processing on GPUs to improve the performance of greedy meshing in voxel-based game engines. It presents two new greedy meshing algorithms and compares their performance and meshing efficiency with binary greedy meshing, a greedy meshing method optimized for the CPU through two example voxel models.**

***Index Terms*—computer graphics, voxels, greedy meshing**

I. Introduction

Voxel game engines rely on efficiently rendering large amount of voxels at once. This is achieved either by sending ray casts into a volume of voxels or by generating a mesh from the raw voxel data.

As ray casting can take up a lot of the performance budget, especially on lower end GPUs, meshing is often the preferred method as it is generally the cheaper rendering method.

Meshing all faces of each voxel individually produces a lot of vertices that don't contribute anything to the final rendered image, especially for bigger surfaces. This takes up performance in rendering through additional vertex and fragment shader calls. Because of this, there are multiple methods to reduce the rendered vertex count.

One of these methods is called **greedy meshing**.

The main objective of greedy meshing is to combine the faces of multiple voxels that are next to each other into bigger ones to reduce the vertex count whenever possible.

The state of the art way of doing this is with binary operations, so-called **binary greedy meshing**.

Binary greedy meshing simplifies the decision of whether multiple voxel faces can be combined into a larger one by turning it into a binary problem. A voxel is either solid or empty. This requires a pre-processing step before any meshing is done that generates an occupancy mask for the voxel volume that contains values of 1 (solid) and 0 (empty). By doing so binary operations can be used to determine which faces can be combined into larger ones and which cannot.

A major disadvantage of binary greedy meshing is that it requires multiple loop steps to fully mesh a volume of voxels. Based on implementation, this also includes multiple passes for each voxel color in the model. This can eat up a lot of meshing performance.

While greedy meshing is a much discussed topic when talking about improvements to the algorithms used, potentially moving it from the CPU onto the GPU often comes too short, even though doing more of the processing in parallel sounds like a perfect fit for greedy meshing in general.

To examine the advantages and disadvantages of meshing voxel volumes on the GPU compared to doing so on the CPU, this paper proposes two new algorithms that take advantage of more parallelism a GPU can provide with the goal of achieving a performance gain over traditional greedy meshing methods.

II. Background

As we had not done any programming on the GPU before writing this paper, this was the perfect project to get started with it.

Two algorithms were developed from scratch without looking at any solutions used in binary greedy meshing with two different approaches to discover potential trade-offs you have to make when working with voxel volumes directly on the GPU. Finally, their performance was compared to a preexisting binary greedy meshing implementation [1]. The full source code of this project is available on GitHub [2].

III. Related Work

GPU-optimized greedy meshing is a little discussed topic in the scientific space, as most voxel engines are developed by hobbyists.

However, greedy meshing in general is often discussed online [3] [4]. Multiple videos covering the topic can also be found [5] [6] [7] [8], but barely any attempt to do greedy meshing on the GPU [9].

The meshing performance comparisons made later in the paper stem on an implementation of binary greedy meshing algorithm in OpenGL by Erik Johansson [1].

IV. Method

Two different GPU based greedy meshing algorithms were created, both with the focus of taking advantage of the large parallel processing power of the GPU. They were developed in OpenGL using GLSL compute shaders.

OpenGL was chosen instead of Vulkan due to its lower project setup overhead and because it was easier to get used to working with it.

To simplify the process of generating vertex data from voxel volumes and to further decrease the workload of setting up the OpenGL project index buffers were left out. This leads to bigger vertex counts in the final meshed data, but is something that can be improved on in future work.
Before working on the two greedy meshing algorithms, a basic but inefficient voxel meshing algorithm was developed to get used to working with compute shaders as well as getting used to working with the provided voxel data. This algorithm takes the voxel data and creates two triangles per exposed voxel face.
Any used voxel data is loaded from disk using the *.vox* file format provided by the software called MagicaVoxel [10], as it made the creation of test data quick and simple.
Test files are loaded using the ogt_vox.h library [11].
The two developed meshing algorithms are named **slicing** and **greedy_8x8**. Their meshing runtimes are compared to the Binary Greedy Meshing v2 repository [1] referred to as *BGMv2* in this paper, which provides an OpenGL implementation of binary greedy meshing on the CPU.
Due to time constraints, the draw times of the meshes generated by the algorithms are not measured ignoring their different render performances during runtime.

V. Comparison

The two GPU based algorithms are compared to the BGMv2 repository [1] based on their average runtime over 1000 meshing iterations using two different test scenes.
These test scenes are taken from the BGMv2 repository and replicated for the GPU-based algorithms using .vox files. There are a couple of differences between the GPU based algorithms and BGMv2:

- BGMv2 works with chunks of 62 x 62 x 62 voxels, the two new algorithms with models of 256 x 256 x 256 voxels. Due to this mismatch, all test data is restricted to 62 x 62 x 62 voxels.
- BGMv2 is chunk-based and culls faces between chunks. This isn't the case with the GPU based algorithms as models can have any arbitrary position on the voxel grid.
- BGMv2 generates its test data mathematically in code, the GPU based algorithms load it from disk as .vox files.
- BGMv2's meshing times are gathered using a Timer class that is part of the repository.
  An OpenGL timer query was used for the new algorithms to measure the duration of each meshing iteration's compute dispatch, fetching its measured time between dispatches.

All measurements were made on an NVIDIA RTX 3080 graphics card and an AMD Ryzen 7 3800X CPU.

The first test scene is a simple sphere with dimensions of 60 x 60 x 60 voxels (113752 total voxels).
Running the meshing algorithm of all three implementations for 1000 iterations results in the data shown in *table I*

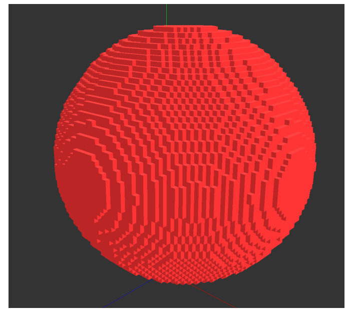

Fig. 1. Sphere test scene

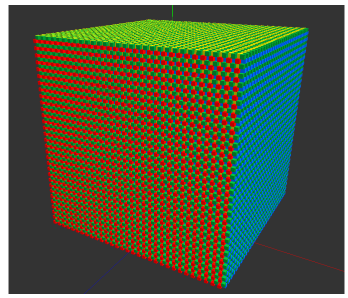

Fig. 2. Worst case test scene

TABLE I
Runtimes after 1000 meshing iterations (Sphere)

	Slicing	**greedy_8x8**	**BGMv2**
Average	123,731 us	120,972 us	134,808 us
Min	119 us	117 us	131 us
Max	131 us	132 us	275 us
Median	124 us	120 us	133 us
Standard Deviation	2.36361 us	2.26875 us	7.44158 us
Total vertices	75984	92052	7201

The second test scene displays a *worst case* scenario made up only of single voxels that do not touch each other but fill up the whole volume of 62 x 62 x 62 voxels. The scene consists of 119164 voxels in total. The runtimes of the three algorithms after 1000 iterations can be found in *table II*

TABLE II
Runtimes after 1000 meshing iterations (Worst Case)

	Slicing	**greedy_8x8**	**BGMv2**
Average	267,479 us	431,613 us	4513,274 us
Min	260 us	420 us	4334 us
Max	280 us	450 us	6969 us
Median	267 us	431 us	4453 us
Standard Deviation	3,28505 us	4,58664 us	221,86500 us
Total vertices	4289904	4289904	714985

VI. Results

The resulting data backs up that there is potential in doing greedy meshing for voxels on the GPU.
Even though the algorithms presented only were an introduction to compute shaders their performance matches and even beats that of binary greedy meshing in multiple cases. Both

GPU algorithms also show much more consistent meshing durations overall.
Especially the slicing algorithm shows potential, being more efficient in reducing vertex counts than greedy_8x8 while also being faster at it than binary greedy meshing.
In the future things like shader subgroups and wave intrinsics can further reduce the meshing durations of the algorithms, simultaneously eliminating the need for some of the memory barriers that are currently required. Incorporating currently missing index buffers to the algorithms can also reduce vertex counts by quite a lot without having to adjust their concepts.

VII. How the algorithms work

A. Required data

Both algorithms require the same data before any meshing is done consisting of three different buffers that are uploaded to the GPU.

1) Voxel Data: This buffer holds a flat array of unsigned 8 bit integers consisting of a voxel model's voxel data in a one dimensional way. Every uint is a color index on the color palette of the model that contains 255 colors in total. If the index has value 0 it means that there's no voxel at that position. Based on a voxel's position in the voxel model's 3D grid an index can be calculated to find its associated color index in the buffer:

```
index = pos.x + (pos.y * model_size.x) +
    (pos.z * model_size.x * model_size.y)
```

Both algorithms expect each voxel model's size to be a multiple of 8 in each dimension. Because of this, each voxel model's size is rounded up in an intermediate step before usage, remapping all indices to the correct model size.

2) Indirect Command: : This buffer holds all the information required to draw the voxel model indirectly after meshing is completed. It is bound to the compute shader to set the correct number of triangles to be drawn for the model during runtime.

3) Instance Data:: Contains a voxel model's size, as well as its offset in world space. Both are required to offset all vertices to their correct positions in world space.

B. How the data is used

1) Algorithm 1: Slicing: The proposed *slicing* algorithm is close to binary greedy meshing in its general approach; however, it only expands faces in one dimension and not two. It is called *slicing* as it splits up a voxel model into slices of 1 x 1 voxels that have the length of the voxel model's size in y direction.
The compute dispatch happens in groups of 8 x 1 x 8 voxels splitting the model's x and y directions into groups of 8 x 8 local threads.
See Appendix A for how the algorithm works.

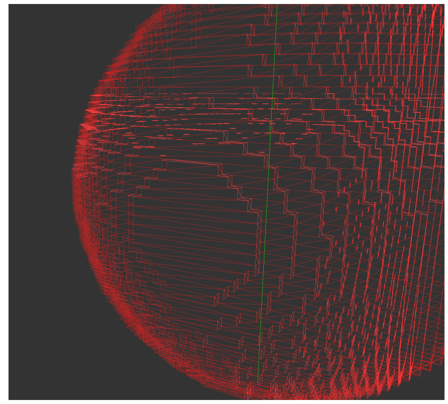

Fig. 3. Example of slicing applied to a sphere mesh

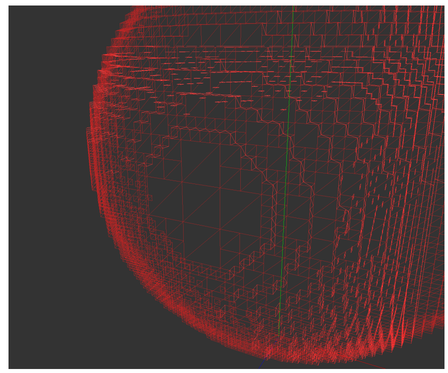

Fig. 4. Example of greedy_8x8 applied to a sphere mesh

2) Algorithm 2: greedy_8x8: This algorithm splits the voxel model into groups of 8 x 8 x 8 voxels. A group of voxels shares six arrays of eight 32 bit unsigned integers.
These arrays are composed as follows:

- A voxel face has six possible facing directions.
  - One array per facing direction
  - Eight layers of voxels per facing direction
- The maximum size of a meshed face in a layer of voxels is 8 x 8.
  - One uint reserved for one facing direction for one layer
  - Each uint has 32 bits, 16 *done* bits and 16 *processing* bits
  - The 8 x 8 voxels are splits into groups of 2 x 2 voxels for processing, single voxel faces are handled separate from this in a final meshing step
    - 1 *done* bit per 2 x 2 group
    - 16 available *processing* bits for 8 x 8 voxels

Processing bits are reserved for calculating whether a face can be meshed or not. The amount of processing bits used increases by a factor of 4 with each step in which a face could not be created. *Done* bit set to 1 marks a group of 2 x 2 voxels as completed. All done bits start off with value 0.

The full functionality of the algorithm with more explanation of the *processing* bits can be found in Appendix B

C. Resulting data

Both algorithms result in a buffer of vertex objects. Each vertex is made up of a position vector along with a 32 bit unsigned integer holding information about its color index on the color palette and its associated normal index.
The normal index represents one of the six possible directions that a voxel face can have on the grid.

VIII. Future Work

The learnings from this paper show that there is a potential performance boost to be gained by moving greedy meshing onto the GPU especially with more optimized algorithms that increase meshing efficiency and performance further.
Future work may include adding support for index buffers to the buffer setup and algorithm logic, which should result in a massive improvement in vertex counts. Reducing the number of memory barriers within the implementations of the algorithms should also be a target. Part of this could be achieved by looking into the use of shader subgroups and wave intrinsics. Potentially moving from OpenGL to Vulkan may also expose additional options for greater meshing efficiency. All combined these improvements could result in an algorithm that is able to generate meshes with smaller vertex counts than binary greedy meshing while also taking less time to do so.

Acknowledgment

Thanks to Stephan Soller for helping me figure out the exact topic for this paper and for supporting me with its creation throughout, as well as Stefan Radicke for being available for questions even though they were in their research semester.

References

[1] E. Johansson, Binary Greedy Meshing v2, 2024. [GitHub]. Available: https://github.com/cgerikj/binary-greedy-meshing. [Accessed: August 11, 2025]

[2] N. Mäckle, Voxel Greedy Meshing Repository, 2025. [GitHub]. Available: https://github.com/knosvoxel/voxel-meshing. [Accessed: August 11, 2025]

[3] M. Lysenko, Meshing in a Minecraft Game, 2012. [Blog Post]. Available: https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/. [Accessed: August 11, 2025]

[4] serg06, Greedy Meshing Using Compute Shaders, 2019. [Reddit]. Available: https://www.reddit.com/r/VoxelGameDev/comments/ebt1th/greedy_meshing_using_compute_shaders/. [Accessed: August 11, 2025]

[5] D. Morley, Greedy Meshing Voxels Fast - Optimism in Design Handmade, Seattle, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=4xs66m1Of4A. [Accessed: August 11, 2025]

[6] Tantan, Blazingly Fast Greedy Mesher - Voxel Engine Optimizations, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=qnGoGq7DWMc. [Accessed: August 11, 2025]

[7] FinalForEach, Optimizing My Minecraft Clone With GREEDY MESHING, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=2xbjP8XbVFY. [Accessed: August 11, 2025]

[8] Vecidium, I Optimized My Game Engine Up To 12000 FPS, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=40JzyaOYJeY. [Accessed: August 11, 2025]

[9] void main, Real-time compute shader greedy meshing, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=ayzq63Z6HFg. [Accessed: August 11, 2025]

[10] ephtracy, MagicaVoxel Release 0.99.7, 2021. [GitHub]. Available: https://github.com/ephtracy/ephtracy.github.io/releases/tag/0.99.7. [Accessed: August 11, 2025]

[11] J. Paver, ogt_vox.h, 2019. [GitHub]. Available: https://github.com/jpaver/opengametools/blob/master/src/ogt_vox.h. [Accessed: August 11, 2025]

Appendix A
Splicing Algorithm Pseudo Code

```
// previous step's color index
prv_col_idx = 0

// for each local dispatch thread
for y = 0 to model_size.y do
    voxel_pos = vec3(global_thread.x, y,
        global_thread.z)
    col_idx = get_col_idx(voxel_pos)

    // gather color indices of neighbours
    neighbours = {
        "x0": get_neighbour_x0(voxel_pos),
        "x1": get_neighbour_x1(voxel_pos),
        "z0": get_neighbour_z0(voxel_pos),
        "z1": get_neighbour_z1(voxel_pos)
    }

    // track if face has been started
    starting_pos = {
        "x0": -1,
        "x1": -1,
        "z0": -1,
        "z1": -1
    }

    for dir in ["x0", "x1", "z0", "z1"] do
        neigh_idx = neighbours[dir]
        start_pos = starting_pos[dir]

        if col_idx != 0 && neigh_idx != 0:
            if start_pos == vec3(-1):
                start_pos = voxel_pos
            end
            else if col_idx != prv_col_idx:
            // Color changed, create face
            // for previous one
                create_face(
                    start_pos,
                    voxel_pos - vec3(0,-1,0),
                    prv_col_idx
                )
            end
        end
        else:
            if start_pos != vec3(-1):
            // empty voxel hit, create face
            // until previous voxel edge
                create_face(
                    start_pos,
                    voxel_pos - vec3(0,-1,0),
                    prv_col_idx)
                )
            end
        end
        // end of column, flush remaining face
        if i == model_size.y - 1 &&
            starting_pos != vec3(-1):
            create_face(
                start_pos,
                voxel_pos,
                col_idx)
            )
        end
    end
    // part done for all four faces
    // ends here

    if col_idx == 0:
        prv_col_idx = 0
        continue
    end
```

```
    // y direction faces
    // current index not empty
    // but previous one
    if prv_col_idx == 0:
        create_face_y0(
            voxel_pos,
            col_idx
        )
    end

    next_col_idx = 0

    // check if next_pos in bounds
    if y < model_size.y - 1:
        next_col_idx =
            get_col_idx(
                voxel_pos + vec3(0,1,0)
            )
    end

    if next_col_idx == 0:
        create_face_y1(
            voxel_pos,
            col_idx
        )
    end

    prv_col_idx = col_idx

end
```

Listing 1. Splicing Pseudo Code

APPENDIX B
GREEDY_8X8 ALGORITHM PSEUDO CODE

```
    // x0, x1, y0, y1, z0, z1
    shared uint processing_buffer[6][8]

    // for each local dispatch thread
    for each voxel:
        voxel_pos = global_thread.xyz
        local_pos = local_thread.xyz

        col_idx = get_col_idx(voxel_pos)

        // gather color indices of neighbours
        neighbours = {
            ”x0”: get_neighbour_x0(voxel_pos),
            ”x1”: get_neighbour_x1(voxel_pos),
            ”y0”: get_neighbour_y0(voxel_pos),
            ”y1”: get_neighbour_y1(voxel_pos),
            ”z0”: get_neighbour_z0(voxel_pos),
            ”z1”: get_neighbour_z1(voxel_pos)
        }

        // correct uints in processing
        // buffer arrays
        correct_mask = {
            ”x”: local_pos.x,
            ”y”: local_pos.y,
            ”z”: local_pos.z
        }

        done_bits = {
            ”x”: 4 * local_pos.z / 2
                    + local_pos.y / 2,
            ”y”: 4 * local_pos.x / 2
                    + local_pos.z / 2,
            ”z”: 4 * local_pos.x / 2
                    + local_pos.y / 2
        }

        // step down in area size
        // 8 * 8 -> 4 * 4 ->
        // 2 * 2 -> 1 * 1
        // represents amount of grouped
        // voxels in final face if
        // face can be created
        for size = 8; size > 0; size /=2:
            // done by only one thread
            if thread_idx == 0:
                // reset buffers to 0
                reset_processing_buffers()
            end

            // update whether neighbouring
            voxels are in same current area
            as voxel
            update_in_area_bools()

            for face in [”x0”, ”x1”, ”y0”,
                ”y1”, ”z0”, ”z1”]:

            // if block of 2x2 voxels not
            // processed yet / done bit
            // for face not set yet
            if not face_processed():
                // face size bigger than 1/
                // single voxel
                if (size > 1):
                    // gather processing bits
                    // of current voxel's face &
                    // its 4 neighbouring voxel's
                    // faces
                    bits =
                        gather_processing_bits(
                            face)

                    // check for color mismatch
                    // or occlusion if neighbour
                    // processing bit is the same
                    if any_color_conflict(
                        bits, col_idx,
                        neighbor_indices[face]):
                        set_processing_bit(
                            face, 1)
                    end

                    // if bit is not 1 its
                    // possible to create a
                    // face here with size
                    // of all voxels that
                    // share this processing
                    // bit
                    if processing_bit != 1:
                        create_face()
                    end

                    // no face was created before
                    // last step in for loop
                    else:
                        create_single_face()
                    end
                end
            end
        end
    end
```

Listing 2. greedy_8x8 Pseudo Code