COMP5115
Final Project
Real-Time Hierarchical Voxelization for Games

Gabriel Racz

December 06, 2024

1 Objective

In recent years, rapid advances in consumer graphics hardware have allowed game developers to create more detailed worlds through new rendering and simulation techniques. Many elements that used to require offline preprocessing such as mesh simplification [8], and hierarchical collision structures [9] are now computed dynamically at run-time using the GPU. Some techniques that were previously too computationally such as volumetric fluid simulation have now begun seeing some representation in AAA games, such as 2018's God of War [5], where a simplified fluid simulation was used to simulate wind interactions with foliage and character clothing.

Notably, all production implementations of fluid simulations for games lack true interaction between the physical game objects and the volumetric fluid domain. To simulate fluid flow around object boundaries, we must incorporate their mesh information into the fluid data structure, ideally concentrating computation around areas of higher detail. This could also be applied to any other game features that require volumetric representations of mesh surfaces, such as collision detection, rigid body physics, volumetric lighting, etc. These challenges motivate the implementation of a fast and efficient method for building a volumetric representation of a given scene at runtime. More specifically, rather than building a complete partitioning each frame, we wish to perform small updates, incrementally coarsening and refining our data structure over time as needed.

We can assume that the full resolution model of an object is not required for the applications stated above, thus our problem becomes one of incremental simplification of a mesh surface into a volumetric data structure. Previous work on GPU mesh simplification [3] [10] is able to achieve real-time rates for simple objects, but constructs the data structure in a bottom-up fashion. These methods perform the entire tree construction each frame, which although effective for preprocessing static scenes, is not compatible with the highly dynamic geometry found in games such as animated characters.

We instead look towards modern methods based on computing a sparse voxelization of a scene, as ideally our volumetric structure is guided to refined details near object surfaces.

2 Methodology

We begin by computing a surface voxelization of the scene geometry over a uniform grid, and then use the individual surface voxels to iteratively construct our hiererchical structure top-down.

Computing a surface voxelization requires marking which cells within a given volume are occupied by solid geometry, where all cells that intersect at least one triangle from a given triangle mesh can be considered occupied. Naively, we would need to test each triangle of our scene geometry against the bounding box of each cell , marking those that intersect. Due to the total number of grid cells being cubic in the grid resolution, this quickly becomes prohibitively expensive, even when accelerating tests by limiting the search space to the triangle's axis-aligned bounding box. Since our data structure should support interaction with dynamic objects, we need a method that is able to voxelize triangle meshes at real-time rates.

Various methods have been proposed to solve this naturally parallel problem on the GPU. For "sufficiently tesselated" meshes, where the grid cells are expected to be much larger than the minimum triangle size of an object, simple point-based lookups can be utilized on each of the triangle vertices as was done by DeCoro et. al to perform mesh simplification [4]. Although simple and efficient for voxelizing individual meshes with approximately uniform distribution of triangles, this approach breaks down when voxelizing entire scenes containing flat areas covered by few very large triangles. These mesh topologies are common in games where polygon counts are typically constrained and reduced wherever possible. Thus we need a method that marks all intersected grid cells as occupied without manually testing each one.

One key observation is that the process of voxelizing a triangle onto 3D grid cells is analogous to the rasterization of triangles onto on-screen pixels. If we can modify the the problem to one of 2D rasterization, we can use the built-in hardware rasterizer found in GPUs to perform extremely fast voxelization. Schwarz and Seidel showed that a ıthin surface voxelization of a triangle T can be computed for each voxel V by testing if either T's plane intersects V (the naive approach) or the 2D projection of T along the dominant axis of its normal intersects the 2D projection of V [11]. The dominant axis of T is the major axis of the scene that provides the largest surface for the projected triangle T' along this axis. Crassin et. al employ this insight, proposing a method that uses the standard graphics pipeline to perform this surface voxelization. [6, 2].

The vertex shader first transforms each vertex from model-space to world-space using a standard model transformation matrix. Then, each vertex is transformed into grid space, where the center of the 3D grid is located at the origin, and the grid spans -0.5 to 0.5 on each major axis. This relative scaling will simplify the projection operation performed in the geometry shader.

The geometry shader projects each triangle orthographically along its dominant axis. The dominant axis is selected dynamically on a per-triangle basis by finding the component of the unit normal vector with the largest absolute value. Once the axis is selected, one of the three following orthographic projection matrices is applied to the triangle to obtain its "clip-space" coordinates (assuming OpenGL-style coordinates).

$$P_x = \begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad P_y = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \quad P_z = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Each projected triangle is then fed through the standard rasterization pipeline, where it is converted into discrete fragments. In order to obtain fragments corresponding to the cell indices of our 3D voxel grid, we set the destination viewport resolution to be equal to our grid's lateral dimensions (for instance 256×256 pixels for a 256^3 voxel grid). This effectively renders each triangle as if an orthographic camera was placed on a face of the voxel grid's bounding box. Since we ensured our grid-space spanned -0.5 to 0.5 in each direction, we modify the depth value of each output vertex in the geometry shader to span the range $[0, 1]$ where 0 corresponds to the vertex being directly on the voxel grid's projection face, and 1 corresponding to the vertex being on the opposite face. The following geometry shader sample illustrates the steps described above.

```
int dominantAxis = getDominantAxis(normal);
for (int i = 0; i < 3; i++) {
    vec3 pos = gl_in[i].gl_Position.xyz;
    vec4 outPosition = orthographicProjection(pos, dominantAxis);
    outPosition.z = ((outPosition.z + 1.0) / 2.0);
    gl_Position = outPosition;
    outAxis = dominantAxis;
    outColor = inColor[i];
    outUV = inUV[i];
    EmitVertex();
}
EndPrimitive();
```

Due to the viewport resolution matching the resolution of the voxel grid, each rasterized fragment's screenspace x and y coordinates directly translate to a 3D index into the grid. To obtain the z component of the 3D grid index, we scale the fragment's depth value, which is still in the range $[0, 1]$ in clip-space, by the voxel grid's dimension in this depth direction.

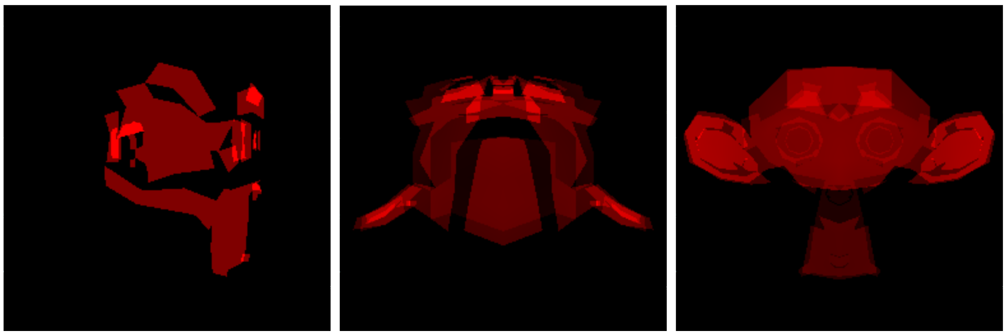
Since each triangle may be projected along a different dominant axis, we must apply a final transformation to each fragment's grid index to ensure that it is consistent with our grid structure definition. We pass each triangle's dominant axis to the fragment shader and perform the following lookup to reorient the indices.

```
float d = Constants.gridDimension
```

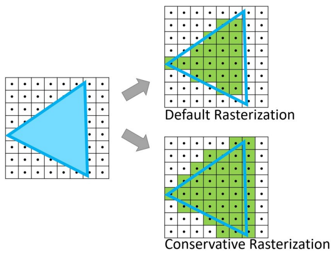
```
    vec3 temp = vec3(gl_FragCoord.xy, (gl_FragCoord.z * d));
    vec3 gridIndex;
    if(axis == X_AXIS) {
        gridIndex = vec3(temp.z, temp.y, temp.x);
    } else if (axis == Y_AXIS) {
        gridIndex = vec3(temp.x, temp.z, temp.y);
    } else {
        gridIndex = vec3(temp.x, temp.y, temp.z);
    }
```

This assumes that grid is oriented such that `grid[0,0,0]` is in the the bottom back left corner along the -x, -y, and -z axes, and `grid[d-1,d-1,d-1]` is in the top front right corner along the +x, +y, and +z axes. This orientation means our representative orthographic projections are done from left to right, bottom to top, and back to front for the x, y, and z axes respectively.

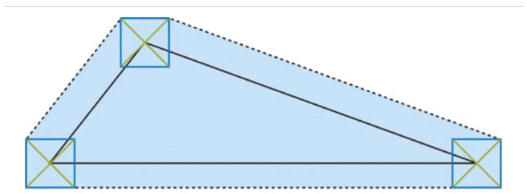
Although no image is rendered due to us writing directly into the GPU grid buffer from the fragment shader, we can visualize the voxel rasterization process by outputting the orthographic projection generated by the geometry shader. Below are some sample rasterization outputs separated by the projected axes x, y, and z respectively for clarity. The intensity of red indicates proximity to the voxel projection plane or "camera".



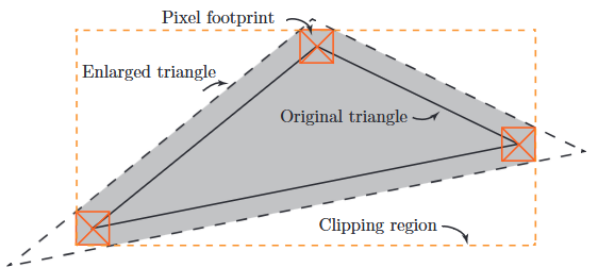
Although extremely efficient and simple to implement, the above approach does not always generate a correct voxelization [11]. This is due to the fact that only the center of each pixel, or voxel cell, is tested for intersection with the triangles by the hardware rasterizer. Triangles that intersect cells but do not pass through their centers are not properly voxelized, causing holes to appear [7].



In the context of fluid simulation for instance, these holes are especially detrimental as it would allow fluid to leak into solid objects, and alter the behaviour of the fluid flow on the boundaries of objects. To ameliorate this, Hasslegren et al. propose a method to ıdilate each triangle in the geometry shader enough such that any pixel that is touching the original projected triangle is guaranteed to have its center intersected by the new polygon [6]. The approach aims to compute a new bounding polygon for each triangle by first locking a pixel-sized cell at each of the triangle's vertices, and then sweeping these cells across the triangle edges, adding all pixels touched by the sweeping cell to the triangle. An illustration of this motivating idea is shown below [6].



Since this optimal bounding polygon is not a triangle, we instead shift each triangle vertex outwards to encompass the optimal polygon, and then discard all fragments in the fragment shader that are outside of a clipping region bounding box that approximates the optimal polygon [1].



The main idea of the algorithm is to find equations of the planes going through each of the triangle edges. Then for each edge plane, it is shifted outwards by its normal direction to intersect the corners of its two connected pixel footprints that are in the same quadrant as the normal. These quadrants are shown in the image above. Once each of the triangle edge planes has been shifted, the new location of the vertices are found by computing the intersection of the two planes incident to each vertex. The final bounding box of the triangle that is used to discard excess fragments is found by adding a half-pixel distance to each of the original triangle vertices. More details can be found in Hasselgren et al. where example code for the dilation is provided [6].

As will be demonstrated in 3, this voxelization approach is very fast irrespective of scene complexity and voxel grid resolution. However, storing the uniform voxel grid requires prohibitive amounts of relatively scarce VRAM. A single grid volume of 1024^3 cells consumes 4GB of memory, exhausting most GPU limits on storage capacity for single buffers. Below we will describe a method for using a sparse representation of the grid structure, varrying the size of voxel cells based on the proximity to mesh surfaces.

Since the performance of the hardware accelerated voxelization described scales so well with polygon counts and voxel grid resolution, it allows us to always voxelize scene geometry at the finest grid resolution and use the generated voxel fragments to iteratively update a sparse representation. Rather than directly inserting into the grid from the fragment shader, we build a list that contains only the computed occupied voxel cell positions called the fragment list. This takes advantage of the highly efficient voxel computation while minimizing its memory usage deficiencies as the surface voxels require an order of magnitude less memory to store when compared to the grid in its entirety.

Since the fragment list is such a general representation, it is compatible with almost any sparse grid representation that we chose. Separating the voxelization from sparse data structure construction is ideal as it decouples the computation of surface proximity from other ongoing rendering or simulation tasks that may also contribute to the final construction of the sparse representation.

In the same publication, Crassin and Green propose a sparse octree structure to store the voxel data at varying resolutions [1]. This data structure is based on hierarchical octree nodes where each successive refinement of a node links it to 8 children evenly dividing it. It is worth noting that this method applies to any N-ary Tree where N can be increased to promote a wider and more shallow tree which is likely to be more performant in our case due to our requirements for creating a volumetric representation over a potentially sparse scene.

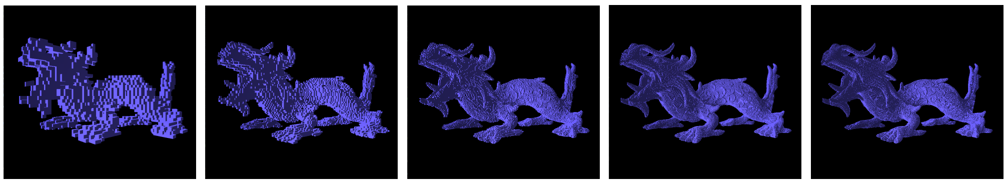
This method begins by first performing the voxelization at the finest resolution as previously described, generating the fragment list. A seperate compute shader pass is launched afterwards with one thread for each element of the fragment list. Each thread descends the currently constructed tree by finding the index of the current node's children that contains the fragment. Intersecting children indices can be efficiently computed by using a classical regular grid clipping operation on the fragment position relative to the current voxel center. Once a containg leaf node is located, it is flagged as being a potential candidate for refinement.

Once the candidate nodes are flagged, another compute shader is launched with one thread per node. Threads for nodes that are not flagged are terminated early, while nodes that are flagged allocate 8 new children inside the tree buffer.

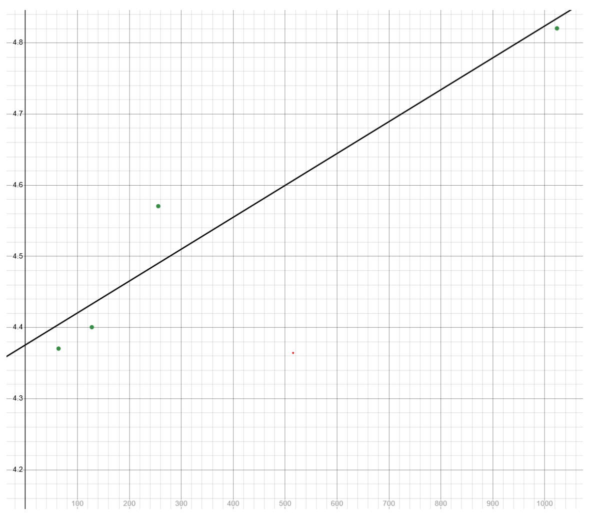
The volume rendering is performed by ray casting through the voxel volume. Each pixel generates a ray that originates at the camera's position and passes through that pixel's coordinate in world-space. Along regular intervals, the ray samples the volume as it passes through the voxel grid. When an occupied cell is encountered it either continues through and accumulates density values scaled by its distance from the camera, or exits early and applies simple Phong shading to the cell face. The rendering of the volume was not a focus of the project and thus is left unnoptimized.

3 Results

Below are the performance results for the hardware accelerated voxelization on the XYZRGB dragon mesh which containing 7,219,045 triangles at various resolutions voxel grid resolutions. Tests were performed on a NVIDIA RTX 2070 Super.



Grid Dimensions	**Average Compute Time (s)**	**FPS**
64x64x64	0.00437	229.008
128x128x128	0.00440	227.273
256x256x256	0.00445	224.719
512x512x512	0.00457	218.978
1024x1024x1024	0.00482	207.612



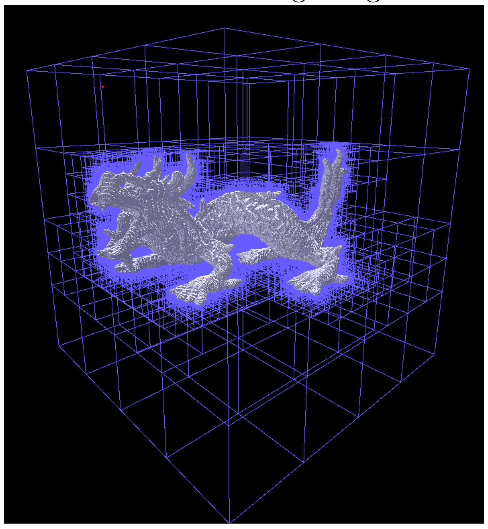
The voxelization performs well at all voxel grid resolutions, where the computation times increase by only fractions of a millisecond when increasing the grid resolution. Since the hardware rasterizer is performing the bulk of the voxel computation, the resulting computation time is linearly proportional with the grid dimension.

Below are the performance results for construction at various tree depths over the same 7.2 million triangle mesh at a grid resolution of 512^3.

Tree Depth	**Average Compute Time (ms)**
1	0.355
2	0.362
3	0.467
4	0.442
5	0.471
6	0.497
7	0.572
8	0.677
9	0.835
10	0.782

As we can see, shallower tree depths lead to less compute shader invocations and thus quicker execution. At tree depth 10, the tree is already over-sampling the input fragment list and we still maintain under 1ms compute time. These results are promising as the combined voxelization and data structure building steps complete in under 5ms which can be suitable for integration into real-time games. With further optimization, there is likely plenty of potential for even further reductions in compute time.

We can also analyze the memory consumption of the various grid representations we discusses to further motivate the need to store a sparse representation of the scene volume. This test is run on the same mesh again on a 512^3 grid. The sparse octree's depth was set such that the highest resolution nodes were the same size as the regular grid voxels.



Representation	**Memory (bytes)**
Regular Grid	536870912
Fragment List	13889024
Sparse Octree	26030592

Our sparse structure requires an order of magnitude less memory than the dense regular grid, and is close to the size of the ground truth fragment list size. With further packing of the octree and fragment list data by using 16-bit floats rather than 32, this memory footprint could be halved again.

Comparing the previous voxelization results to the Crytek Sponza scene with only 262267 various grid resolutions reveals a different outcome.

Grid Dimensions	**Average Compute Time (s)**
64x64x64	0.00016
128x128x128	0.00042
256x256x256	0.00142
512x512x512	0.00513
1024x1024x1024	0.01809

Although the lower resolutions are much cheaper than the dragon, the execution time of the raster voxelization routine quickly explodes as we increase the grid resolution. This is likely due to the number of voxel fragments being much greater than the dragon since the dragon has a much smaller surface area compared to the sparse Sponza scene. One promosing aspect is the good performance at lower resolutions, which are more realistic to be used in game applications. Further performance profiling and optimizations are needed to determine whether this change is inherent to the method or can be improved.

Due to running out of time, node deallocation was not implemented and would be a great area for future research. This would require reordering the nodes in memory after they have been allocated as currently we rely on the semi-sorted nature of the node pool for cache coherent memory lookups, as well as relative pointers between the nodes to link to their children. Also, the volume renderer can be greatly improved as it is currently the main bottleneck in the pipeline. It also only uses the uniform voxel grid for rendering, the sparse representation is purely displayed by the grid lines. Optimizing the sparse structure for ray casting would be another area for improvement.

4 Documentation

The `src/main.cpp` file contains the entire implementation for the Vulkan renderer and voxelization infrastructure. `Renderer::draw` implements each of the algorithm steps described above in the methodology. At the top of the file you will find a `Constants` namespace that controls various parameters of the algorithm. I set some sane defaults to ensure that low power machines will be able to run the implementation. In `shaders/` you will find the various shader programs organized by `raster/` for traditional graphics pipeline tasks and `comp/` for the purely compute shaders. All files prefixed with `voxel` are directly related to the uniform voxel grid whereas the `tree` files are responsible for the various compute passes to build the tree structure.

Once the application opens you have a few basic controls. Clicking and dragging with the mouse will rotate the camera orbit around the object, and scroll wheel will zoom. `Q` will toggle rendering the triangle geometry, `W` will toggle rendering the voxel volume, and `E` will subdivide the sparse voxel tree by one level. I have added `xyz.glb` mesh file for the XYZRGB dragon in case you are interested in stress testing. To use it change `Constants::MeshFile`, set `Constants::MeshIdx = 0` and set `Constants::MeshScale = 0.01`.

To build, ensure that you install the Vulkan SDK and that it is visible to your C++ compiler. The other libraries are bundled with the project and should be found by cmake. Run the following sequence to build and run:

```
mkdir build
cd build
cmake ..
cmake --build . --config Release
cmake --build . --target shaders
./Release/engine.exe # on Windows
./engine             # on Linux
```

I tested on both my Windows and Linux machines to ensure compatibility but this is my first experience distributing a Vulkan project. Please send me an email or message if the build fails for whatever reason.

References

[1] Cyril Crassin and Simon Green. "Octree-based sparse voxelization using the GPU hardware rasterizer". In: *OpenGL Insights* (2012), pp. 303–318.

[2] Cyril Crassin et al. "Interactive indirect illumination using voxel cone tracing". In: *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, pp. 1921–1930.

[3] Christopher DeCoro and Natalya Tatarchuk. "Real-time mesh simplification using the GPU". In: *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. 2007, pp. 161–166.

[4] Christopher DeCoro and Natalya Tatarchuk. "Real-time mesh simplification using the GPU". In: *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. 2007, pp. 161–166.

[5] Sean Feeley. *Interactive Wind and Vegetation in "God of War"*. Jan. 2022. URL: https://www.youtube.com/watch?v=MKX45_riWQA.

[6] Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. *GPU Gems 2, chapter Conservative Rasterization*. 2005.

[7] Benjamin Kahl. "Hardware Acceleration of Progressive Refinement Radiosity using Nvidia RTX". In: (Mar. 2023). DOI: 10.48550/arXiv.2303.14831.

[8] Brian Karis. *Nanite, A Deep Dive*. 2021. URL: https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf.

[9] Ryan Robert Kroiss. *Collision Detection Using Hierarchical Grid Spatial Partitioning On the Gpu*. 2009. URL: https://scholar.colorado.edu/concern/graduate_thesis_or_dissertations/vd66w0289.

[10] Peter Lindstrom. "Out-of-core simplification of large polygonal models". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 259–262.

[11] Michael Schwarz and Hans-Peter Seidel. "Fast parallel surface and solid voxelization on GPUs". In: *ACM transactions on graphics (TOG)* 29.6 (2010), pp. 1–10.