VIENNA UNIVERSITY OF TECHNOLOGY

PROJECT DOCUMENTATION

Engine186 GPU Rasterization-based Voxelizer Documentation

Student: Nikole Leopold

Supervisors: Johannes Unterguggenberger

December 10, 2019

1 INTRODUCTION

Engine186: A rendering engine, implemented with C++17 and OpenGL 4 is an open source project created and maintained by Johannes Unterguggenberger. Its source code can be found at [Unt18a].

A Linux port of Engine186 was made by Nikole Leopold. The Linux port source code can be found at [Unt18b]. Documentation of the port is found in another document submitted for this project.

This document provides documentation on the voxelization functionalities introduced in the course of the practicum. Familiarity with Engine186 usage is assumed.

The implementation is based on the theoretical framework provided by Crassin and Green [CG12], but not the whole technique is implemented, only the voxelization to a 3D texture based voxel grid.

2 VOXELIZATION PIPELINE OVERVIEW

The approach by Crassin and Green is significant, as it uses the GPU rasterizer to voxelize a triangle mesh. The voxelization pipeline is summarized as seen in Figure 2.1.

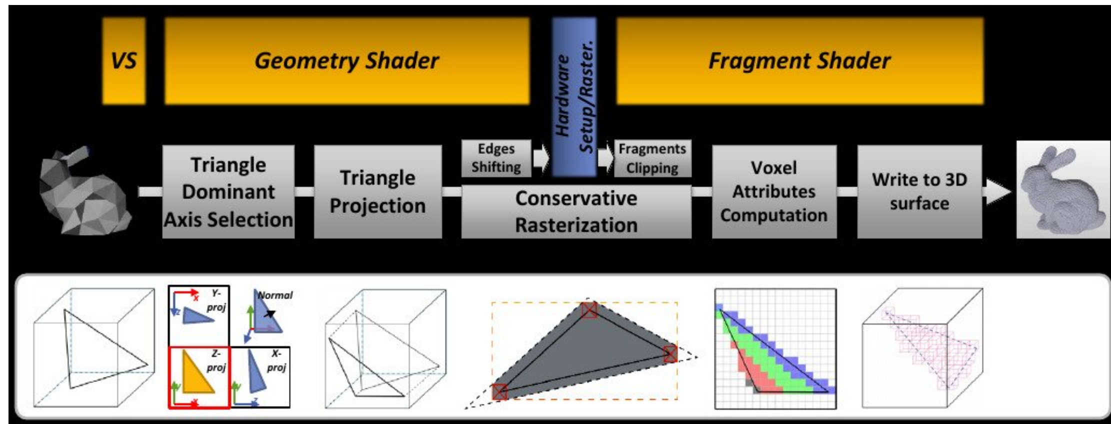


Figure 2.1: One-pass voxelization pipeline by Crassin, Green. Image from conference slides by Crassin [Cra12].

1. Triangle Dominant Axis Selection and Triangle Projection: From view direction, the rasterizer would not cover many of the fragments of many triangles. To maximize the number of voxels generated and minimize information loss, each triangle is orthographically projected along the axes of the voxel grid which are usually aligned to the world coordinate system. Of the three axes, the one is chosen which maximizes the projected area, i.e. which maximies the dot product with the triangle normal. This is done in the geometry shader.

2. Conservative Rasterization: The GPU rasterizer is setup to produce a fragment for each voxel. To avoid holes in the resulting voxels, it is necessary to use conservative rasterization. Usually, only the center of a pixel is used to test if it lies on a triangle to produce a fragment. Instead, a fragment should be produced if a triangle touches any part of the pixel footprint. One approach to achieving this is that a larger bounding polygon is calculated for each triangle, so that if the original triangle touches the pixel footprint the bounding polygon surely touches the pixel center. To do this, each triangle edge is shifted outward to enlarge the triangle. After rasterization, fragments at the corners must be clipped by a bounding box in the fragment shader. Conservative rasterization can also be achieved using the NVidia hardware conservative rasterization extension (OpenGL 4.3+).

3. Voxel Attributes Computation and Write to 3D surface: The fragment shader transforms each fragment from its local projected coordinates back to the world space to find the voxel location. It then calculates a composite of mesh data samples (e.g. diffuse color texture) of all fragments coinciding on the voxel. Since fragments are processed in parallel, it is necessary to use atomic operations, e.g. to average coinciding samples. The resulting mesh surface voxel data are stored in a 3D texture or an octree data structure.

3 Voxelizer Implementation

The implementation is done in the following files:

- `src/VoxelizationTestScene.h`,
  `src/VoxelizationTestScene.cpp`:
  The VoxelizationTestScene class contains code calling Engine186 functions to load a triangle mesh, initialize the voxelizer and allow to start the voxelization process, as well as the Engine186 main loop with standard light and camera setup and update and draw calls, and a simple UI.

- `src/Voxelizer.h`,
  `src/Voxelizer.cpp`:
  The Voxelizer class encapsulates the 3D texture and the voxelization pipeline shaders, sets up the view projection matrices for triangle projection as well as setting OpenGL state, and manages relevant variables like grid size (voxelization resolution). Variables can be set via the UI or function calls. The voxelize function takes only a reference to the mesh model as input. Rendering of the 3D texture is handled by the Tex3dDisplayer class provided by Engine186.

- `assets/shaders/voxelize.frag`,
  `assets/shaders/voxelize.geom`,
  `assets/shaders/voxelize.vert`:
  The voxelize shaders are the GPU part of the voxelization pipeline implementation, and consist of a vertex shader, a geometry shader and a fragment shader. Details are described in the following.

The Voxelizer class provides a function `void Voxelizer::Voxelize(Model& model)`. This function takes an Engine186 Model with a triangle mesh as input, which is uploaded to a GPU vertex buffer. The 3D texture is initialized or resetted. It then initializes axis-aligned orthograhic view-projection matrices for the geometry shader. Three different view matrices are used, looking at the unit cube in direction X, Y and Z. The orthographic projection matrix transforms into normalized device coordinates, i.e. scales and translates a cube of size of voxel grid to a volume [-1,1] in x and y and in [0,1] in z (no perspective divide is needed), the actual 'flattening' is done by the hardware which outputs a 2D buffer. Z-buffering is disabled to avoid loosing any triangles. UI variables like the voxel grid size are also passed to the shader program. Before the draw call, framebuffer operations are disabled via `glColorMask`, since the image load/store framework is used instead. `GL_CULL_FACE` is disabled to not discard triangles facing a certain direction. `GL_DEPTH_TEST` is disabled to not discard fragments that are behind others. `glViewport` is set to the size of the voxel grid resolution, to have as many pixels as there are voxels from each unit cube direction.

Edge Shifting and Fragment Clipping is not needed, thanks to using the NVidia hardware conservative rasterization extension (OpenGL 4.3+) `GL_NV_conservative_raster` and enabling `GL_CONSERVATIVE_RASTERIZATION_NV`.

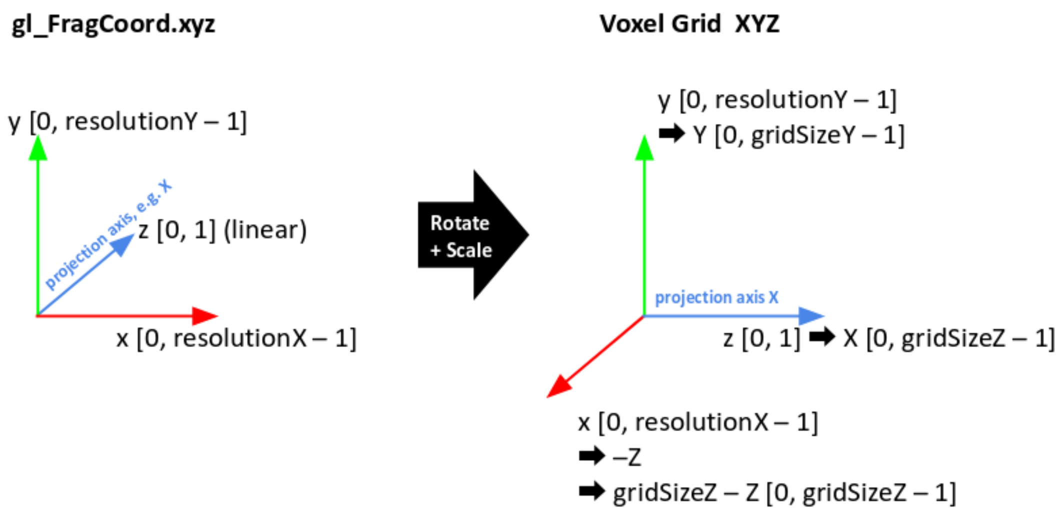


Figure 3.1: The fragment shader provides `gl_FragCoord` as a left-handed NDC space. `gl_FragCoord.z` stores depth linear in [0,1] due to orthographic projection. Knowing the world aligned projection axis, we can rotate and scale each projected fragment into the world aligned voxel grid (X,Y,Z). The left-handed coordinate system is also transformed to right-handed by flipping one coordinate.

The vertex shader merely passes the vertex locations and texture coordinates on to the geometry shader, and if needed scales the model.

The geometry shader has texture coordinates and triangle vertex data as input `layout (triangles) in;`, and outputs triangle normals, texture coordinates and triangle vertex data `layout (triangle_strip, max_vertices = 3) out;`, as well as the axis of triangle projection which is used later to determine the local orientation of the triangle in the voxel grid. To allow use of the GPU 2D rasterizer to help in voxelizing the triangles, without loosing fragments due to using the wrong view direction, before the rasterization, the geometry shader projects triangles along the axis that maximizes the area of the triangle 2D projection. The axis of projection is chosen which maximizes the dot product with the absolute value of the triangle normal, because in this direction of projection the projected area is maximized. The projection is done using the predefined view-projection matrices, without perspective divide.

After rasterization, the fragment shader computes the location of the projected triangle fragment in the voxel grid, as illustrated in Figure 3.1.

At the voxel position some model data can be sampled, for example the diffuse color texture, and stored in the 3D texture via the `image3D` interface. Ideally, the sampling would

also include a compositing step, averaging values from all projected triangle fragments that coincide on the voxel. This requires atomic operations since fragments are processed in parallel. Multiple approaches for this problem are described in [CG12], it is important to note that compositing is not implemented, an attempted implementation only works for integer images not for float data like the diffuse color. The resulting 3D texture containing the voxelized model surface can then be used for other purposes or drawn using the Engine186 Tex3dDisplayer. A screenshot is shown in Figure 3.2.

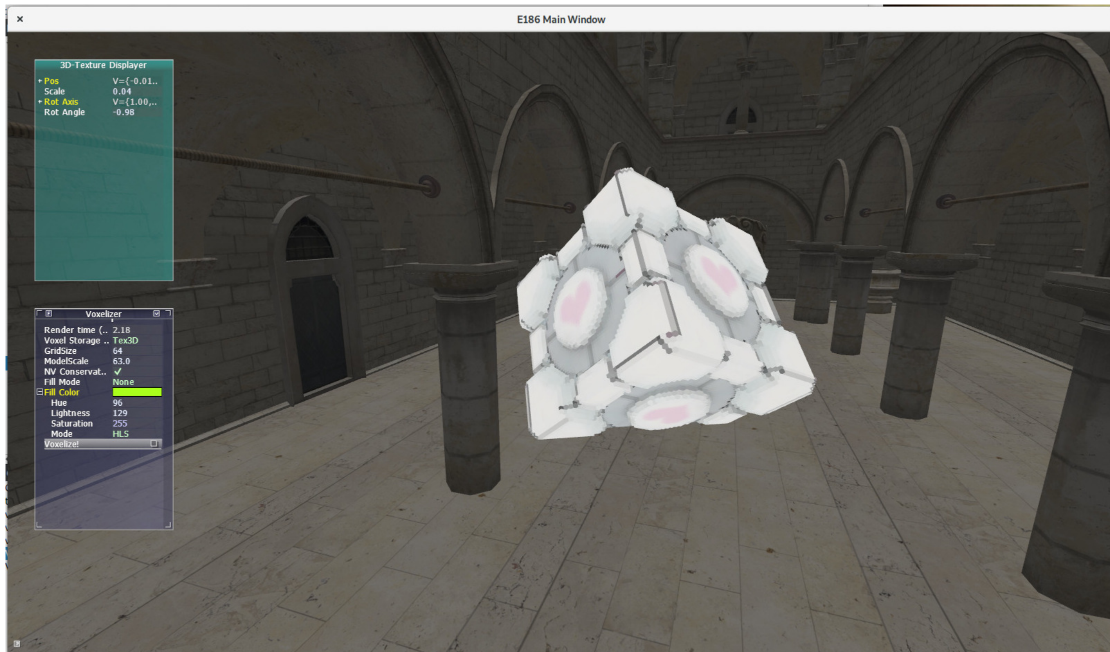
REFERENCES

[CG12] Cyril Crassin and Simon Green. Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights*, pages 303–318, 2012.

[Cra12] Cyril Crassin. Octree-based sparse voxelization for real-time global illumination. In *GPU Technology Conference–GTC12*, 2012.

[Unt18a] Unterguggenberger, Johannes. Engine186: A rendering engine, implemented with C++17 and OpenGL 4, set-up for Visual Studio 2017, 2018.

[Unt18b] Unterguggenberger, Johannes and Leopold, Nikole. Engine186 Linux Port, 2018.



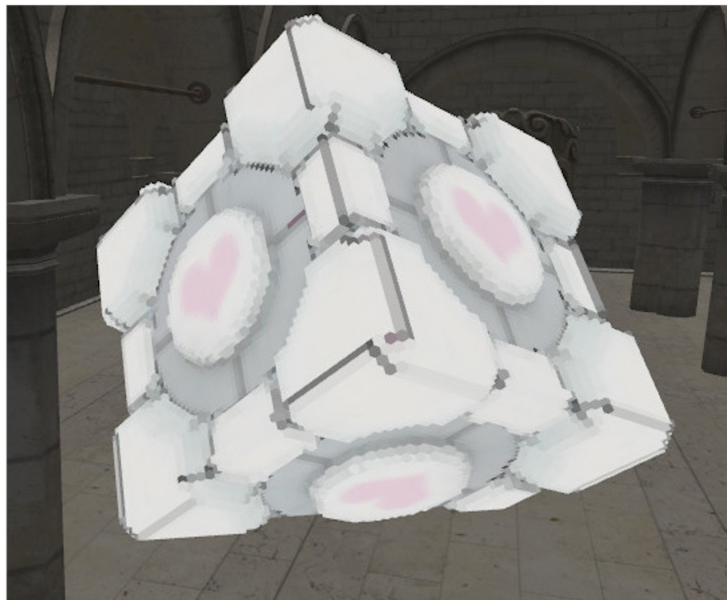


Figure 3.2: Screenshot of a test scene in Engine186 Linux port, with enlarged region below, showing the 3D texture obtained from voxelizing a mesh model of Aperture Science Weighted Companion Cube, visualized using Tex3dDisplayer. It can be seen that while there are no holes thanks to conservative rasterization, the voxel colors suffer from undersampling and lack of sample compositing.