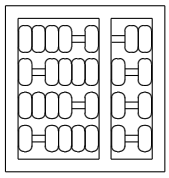

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Game Engine with 3D Graphics

Cauê Viegas Oliveira *Felipe Santos Oliveira* *Helio Pedrini*

Relatório Técnico - IC-PFG-16-13 - Projeto Final de Graduação

December - 2016 - Dezembro

Game Engine with 3D Graphics

Cauê Viegas Oliveira* Felipe Santos Oliveira† Helio Pedrini‡

Abstract

A game engine is a computational tool that consists of general purpose codes for game development, which provides features for rendering and audio, techniques for artificial intelligence and physics, among many other functionalities. In general, scenario editors and script languages are also available. Just as the automotive industry can take advantage of the design of an engine, different games can reuse the same game engine, making the development process simpler and faster. The focus of this project is the development of a rendering engine, since this is one of the most important component of a game engine. C++ programming language and OpenGL graphics package were used to build a Voxel Game Engine (or Block Engine). Height maps were generated procedurally using Perlin noise, as well as the creation of two types of camera: (i) first person (FPS camera) through perspective projection and (ii) isometric camera by means of isometric projection. In addition, the project allowed the interaction with the map to generate new blocks on those already generated.

1 Introduction

In game development, *game engine* is the framework that encompasses general purpose code that games typically require: a *rendering engine* (2D or 3D) to draw graphics, an *audio engine* for sound effects, a *physics engine* generally for collisions, and an artificial intelligence engine. Additionally, it may include a scene graph, a scripting language and many other content creation tools to alleviate the process of game making from scratch.

As such, as well as a mechanical engine design may be reused in multiple machines, a game engine may be reused or adapted in multiple games to make the process of game making cheaper and faster, since programmers will build code on top of a part that is already done. Moreover, if the engine is cross-platform, it will be easier to port video games to different platforms (for instance, porting a game from a console to a personal computer).

The term is generally credited to John Carmack, during the development of video games *Doom* and *Wolfenstein 3D*, in which he noticed there were several reusable aspects of

*Institute of Computing, University of Campinas, 13083-852, Campinas, SP
†Institute of Computing, University of Campinas, 13083-852, Campinas, SP
‡Institute of Computing, University of Campinas, 13083-852, Campinas, SP

the code that could be split from specific parts of each video game. Nowadays, powerful commercial engines such as Unity3D [1], Unreal Engine [2] and GameMaker [3] are used by many game developers who want to focus purely on game content instead of all the process of making a game.

In our project, we focused most of our effort at the rendering engine, since it is the core part of any game engine. We opted for using the *rasterization* method in rendering 3D graphics, using the OpenGL library. Furthermore, to simplify the code, we implemented a *Voxel Game Engine* – or Block Game Engine – in the same visual style of video games such as Minecraft.

2 Conceptual Definitions

This section briefly describes some relevant concepts related to the topic addressed in this project.

2.1 OpenGL

OpenGL is an Application Program Interface (API) for rendering 2D or 3D graphics and interacting with the GPU. More precisely, it is a specification maintained by the Khronos Group [4] which defines the behavior of each function. The implementation itself is typically maintained by the graphics card manufacturers. As such, each graphics card may support different versions of OpenGL, depending on the drivers.

It also accepts two ways of developing: using the *immediate mode* and the *core-profile* mode. The immediate mode is a much more simplified version, however, a more inefficient way since it does not offer much flexibility and control over the graphics, which made the team start to deprecate the immediate mode and encourage developers to use the core-profile mode. The core-profile forces the developer to use modern practices that are more difficult to learn, but that are well worth the effort, since the performance can be much better.

2.2 Rasterization

Rasterization is the process of transforming a vector graphics (curve based) image into a pixel based image (raster image). It is a faster rendering technique in comparison with other approaches such as ray tracing. It is currently the most popular technique to render real-time 3D graphics, specially in the video game industry.

2.3 Shaders

Shaders are computer programs that perform the process of *shading*, creating depth perception and producing different levels of colors by using varying levels of darkness.

OpenGL defines several types of shaders (vertex shaders, geometry shaders, fragment shaders, among others), which may be fully programmed using the OpenGL Shading Lan-

guage (GLSL). The GPU typically has small processing cores for each type of shader that will be used during the graphics pipeline defined by OpenGL.

Figure 1 illustrates an example of the graphics pipeline of OpenGL 3.3 showcasing the shaders (GL 4.0 introduced the tessellation shaders and GL 4.3 introduced the compute shaders, which are not represented).

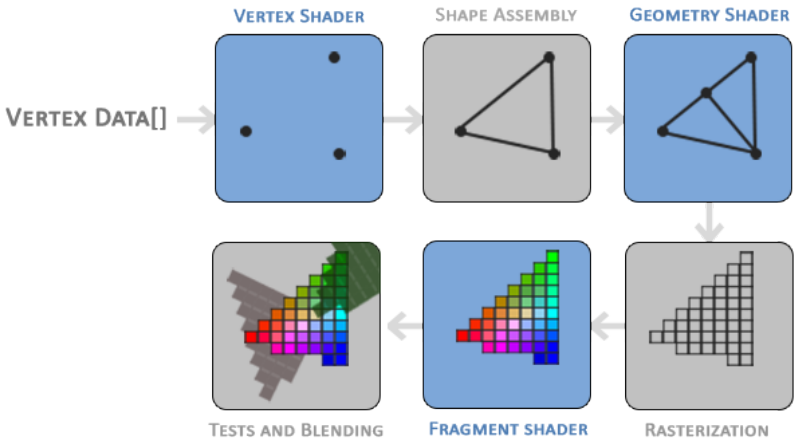

Figure 1: Graphics pipeline of OpenGL. The shader programs are represented in blue (source: Learn OpenGL [5]).

2.4 Viewport

The viewing region in computer graphics. In our project, it is defined as the window size of the program.

2.5 Normalized Device Coordinates (NDC)

The Normalized Device Coordinates correspond to a coordinate system that defines the values of the axis x, y, z to vary from -1.0 to +1.0. It is applied after converting coordinates from view space (or camera space) to screen space (which is done through a viewport transform and thus depends on the viewport size). Any pixel coordinate that falls outside this range after the transformation is clipped.

2.6 VBOs, IBOs, VAOs and FBOs

Vertex Buffer Object (VBO) is an OpenGL feature which defines a buffer used to store a large number of vertices (including its position, color, etc) in the GPU memory by uploading each vertex data through batches of data.

Index Buffer Objects (IBOs) are buffers for storing indices and reusing previously defined vertex positions, thus reducing the number of vertices needed when there are two vertices pointing to the same position.

An array of VBOs is called *Vertex Array Object (VAO)*, which is required to be binded in Core OpenGL.

The *Frame Buffer Object (FBO)* is a special buffer that can store a rendered scene in a texture. The resulting frame-buffer can be used to apply post-processing effects on the scene, such as shadows, motion blur or lens flare.

2.7 Game Loop

The core portion of the code that keeps looping until the game is closed. It is in which the rendering is done, the buffers are swapped, the game state is updated, and the inputs are polled (for instance, keyboard inputs and mouse events).

2.8 Frustum

The portion that lies between the parallel planes of a solid. In our project, it will refer to the "viewing box" that the projection matrix creates. Each pixel inside this frustum will end up on the user's screen, while the pixels outside will be clipped.

2.9 Orthographic Projection

An orthographic projection defines a cuboid by specifying the width and height of two planes (called *near plane* and *far plane*) and the distance between the planes. The frustum will be everything inside the cuboid which will be mapped directly to NDC through the orthographic projection matrix. Since it maps directly and does not take into consideration the distance of the camera to the objects, it will allow for accurate measurements, but may draw unrealistic scenarios.

2.10 Perspective Projection

The perspective projection is a type of projection that mimics the human eyes to create the illusion of 3D: objects that are farther away from the camera will appear smaller, whereas objects that are near will appear bigger. It does that by introducing a fourth coordinate (w coordinate): the further away the object is from the camera, the higher the w component becomes. When transforming to NDC, each x, y, z coordinate will be divided by w (called *perspective division*), creating the perspective:

$$Position = \begin{bmatrix} x/w \\ y/w \\ z/w \end{bmatrix} \tag{1}$$

In OpenGL, the frustum will be a *pyramidal frustum* defined by three parameters: fov (field of view), aspect ratio (calculated by dividing the viewport width and height) and the distance between the near and far planes of the pyramidal frustum.

2.11 Isometric Projection

A type of *axonomic projection* (the plane or axis of the object is never drawn parallel to the projection plane) that can be achieved by doing an orthographic projection followed by a pitch of 45° and an yaw of $\arcsin(30°) \approx 35.264°$. It can loosely be considered as a subclass of orthographic projection.

2.12 Perlin Noise

Perlin noise is a type of gradient noise developed by Ken Perlin [6] in 1983. It is widely utilized on procedural texture generation, as well as in procedural terrain generation. Here we use a combination of different instances of Perlin noise, each with different attributes, mixed together in order to generate a more realistic terrain.

It works by defining a n-dimensional grid, in which each point is assigned a random vector on the unit circle. Then, we try to find in which grid cell the point is by calculating the dot product of the gradient vector at the point and the distance to each cell corner. The final step is to interpolate the 2^n dot products computed at the grid cell containing the desired point using a function that has zero first derivative.

An instance of Perlin noise by itself does not generate a good enough noise pattern, requiring the sum of different noises to create a satisfactory pattern. The result of the sum of different instances of Perlin noise generates a *fractal noise*. Figure 2 illustrates an example of Perlin noise.

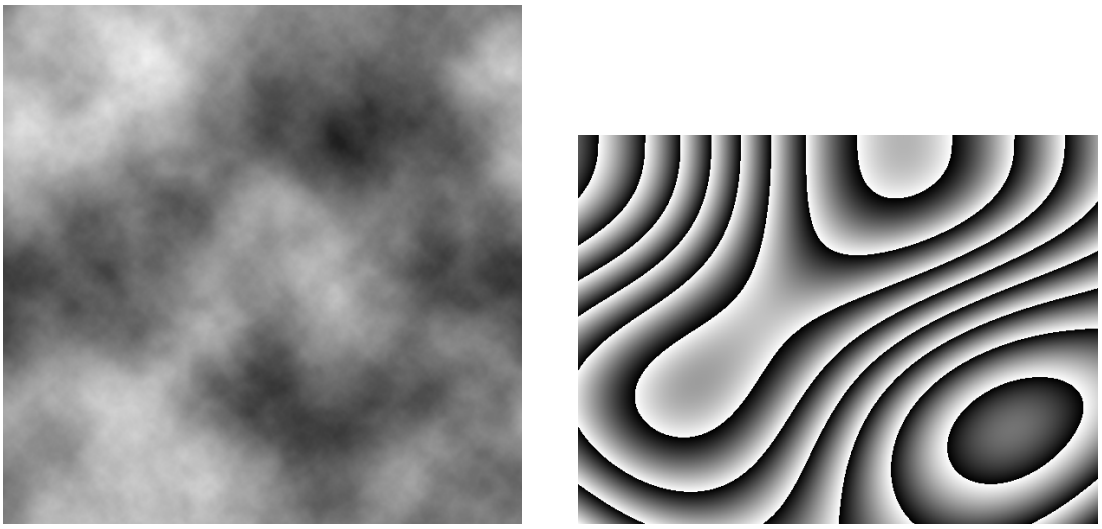

Figure 2: On the left, fractal noise generated by the addition of two instances of Perlin Noise. To the right, wood texture generated by an instance of Perlin noise.

2.13 Heightmaps

Heightmaps or elevation maps are grids or grayscale textures that contain height data. These can be pre-calculated or procedurally generated using various methods, such as the diamond-square algorithm and noise based methods, as seen in Section 2.12.

2.14 Volume

The entire engine is base on the idea that elements have volume. This is represented by the use of volumetric pixels or *voxels*, which can be represented as a mapping of a 3D point to a data type or a 3D matrix in which each position represents a point and the value the type associated with that point.

Volumes can be organized in *chunks*, structures that represent a discrete volume of the space, storing important information such as neighboring chunks, the volume mesh, and so on. In this work, voxels are graphically represented by *cubes*, due to both ease of spacial manipulation and aesthetical properties.

3 Implementation

This section describes the implementation details associated with this project.

3.1 Language and Libraries

We opted for using C++ as the main programming language for the project, since our goal was to learn more about the language and also because it is a popular language for game engines. OpenGL Shading Language (GLSL) was also used for programming the shaders.

The project was all done using Visual Studio 2015 – required to compile and run the project – and still is not ported to Linux.

The libraries used in the project are:

- *Simple and Fast Multimedia Library* (SFML): for everything related to creating the game window and handling the inputs from keyboard and mouse.
- *OpenGL Mathematics* (GLM) [7]: a header-only library used for all the mathematics done: matrices transformations, Euler angles, noise functions, etc.
- *OpenGL Extension Wrangler Library* (GLEW) [8]: used to simplify the code: instead of directly linking with the GL functions (which would be a tiresome process, since we would have to query for the location of each of the functions because there are many different versions of OpenGL drivers), we use this library to do the hard work and just return the pointer to the function.
- *Open Asset Import Library* (Assimp) [9]: used for model loading.
- *Simple OpenGL Image Library* (SOIL) [10]: used for texture loading.

3.2 Project Structure

The repository root of the project is divided into two folders: (i) *Common*, which stores the libraries and (ii) *CubeProject*, which is the root of the Visual Studio project and where the code we programmed is. Inside the CubeProject folder, we have two main folders:

- *shaders*: with all of the shaders we coded in GLSL.
- *tactical*: where the C++ files reside.

Since the *tactical* folder is the most important folder of the project, in the next subsections we will explain how the folder is structured as well as highlight the important files of our project.

3.2.1 *entity* folder

This is the folder where drawable entities reside, with *IEntity3D* inside the *interface* folder being the abstract parent class.

DrawableBox contains the code to draw a box and *Prism* contains the code to draw a prism. Both are used in the picking algorithm to highlight where the mouse is pointing to.

3.2.2 *math* folder

Folder where mathematical functions reside, such as frustum manipulation and ray cast picking.

It also contains one of the most important pieces of codes of our project: the *Perlin noise* to procedurally create the *heightmaps*. The algorithm and its files are inside the *noise* folder.

3.2.3 *render* folder

The render folder contains all portions related to *rendering*. As one of the main folders of the project and with several other important subfolders, we will explain each subfolder:

- buffer: contains the parts of the code most intimately related to the core of the OpenGL rendering: generating, binding and manipulating VBOs, IBOs, VAOs and FBOs. The *Buffer* class contains the code for VBOs, the *FrameBuffer* class contains the code for FBOs, *IndexBuffer* class contains the code for IBOs and the *VertexArray* class contains the code for VAOs.
- camera: contains files related to creating the camera. The *Camera* class is the abstract parent class which defines some functions the children must have, as well as also code obligatory to every camera – such as rotation, translation and updating the orthonormal basis vectors of the camera (up, right and front vectors of the camera). The camera uses Euler angles, since none of the cameras we implemented will have problems of *Gimbal lock* and using quaternions will not be needed. The camera is not

dependent on the FPS (frames per second) of the program (that is, it will not move faster or slower if the user has higher or lower FPS, respectively). The *FPSCamera* (First-person shooter camera) class is basically the perspective projection with some restrictions: the camera cannot roll and the pitch is restricted from $-90.0°$ to $90.0°$ – thus evading from Gimbal lock problems. The *IsometricCamera* class is the camera using isometric projection. Although in true isometric projection the yaw is locked in $arcsin(30°) \approx 35.264°$, for now we allow changing the yaw values – with the right mouse button – for debugging purposes. The pitch is locked in $45°$.

- geometry: contains a single header that defines mesh geometry operations, such as a function to add quads, triangles and calculates normals and define color of the vertices of a mesh.

- structures: as the name implies, it contains classes which are *structs* in C++ used in the rendering process: *Light*, *Mesh*, *Shader* are all files containing structs necessary for the purpose its names implies. The *Light.h* file defines the light types we are using on the engine. These are the classical three types: *directional light*, *point light* and *spotlight*, as well as a base struct for lights and a *Phong* struct. The *Phong* struct defines the three shading attributes of the Phong shading model, *ambient*, *diffuse* and *specular* colors. Directional lights, in addition to basic attributes, have a *direction* attribute. Point lights do not have a single direction, but have three new attributes: *linear*, *quadratic* and *constant* attenuation values. Finally, spotlights inherit from point lights, and have three extra attributes: *direction*, *cutOff* and *outerCutOff*. The *cutOff* and *outerCutOff* values define how wide is the light cone and the smoothness of the transition on the borders of the light circle, respectively. The *Mesh.cpp* file defines the *mesh* structure. It consists of two lists, the vertex list and the index list, and two buffers, the index buffer object and the vertex array object. Both of these buffers, as stated previously, are used to send geometry data to the GPU. The *Shader* class defines how the engine handles OpenGL shader programs. It encapsulates all the process of reading the shader source, compiling and linking the shader. It also contains a parser to identify and map uniforms and structures in the shader source as a way to speed-up the uniform value changes present throughout the rendering process. The *Vertex.h* file defines the types of vertex supported on the engine. Each type of vertex has a different number of attributes, varying from 1 to 4. These are *position*, *normal*, *uv coordinates* and *color*, in this order, and the presence of each attribute in a vertex type is represented by its size. For instance, a vertex containing *position*, *normal* and *color* is `Vertex3f3f4f`, while a vertex containing all attributes is `Vertex3f3f2f4f`. All vertex types have the *position* attribute in common, but no inheritance was used in these definitions.

The *Renderer* file/class is the main file that associates all of the above files to do the process of rendering: linking a camera, initializing and binding buffers and drawing. It also accepts two ways of rendering: *polygon mode* (filling all the polygon with the color) or *wireframe mode* (filling only the lines).

3.2.4 *utils* folder

Folder for utility classes: logging, defining a struct for filenames, reading files, etc. It also contains the *ThreadPool* class, which defines a Thread Pool Pattern to create and manage threads.

3.2.5 *volume* folder

This folder is one of the cores of the project and contains files for manipulating voxels, meshes, chunks, heightmaps, among others. We will explain each file:

- *Voxel*: contains the *voxel* struct and the *Volume* class. Used for defining a voxel and its volume, which will be used for creating the boxes of the heightmap.
- *Chunk*: class that defines a chunk, containers for the volume class previously defined. The voxel coordinates in each chunk are in "chunk space", that is, are numbered from $(0, 0, 0)$ to $(n-1, n-1, n-1)$, where n is the number of voxels and needs to be converted to world coordinates during mesh generation.
- *ChunkManager*: one of the most important files of the project, it is the main component for creating procedurally generated heightmaps. The ChunkManager creates a heightmap using the Perlin noise algorithm explained above (2.12). To change the heightmap generated, modify the values of

```
heightMapBuilder.SetBounds(15.0, 16.0, 4.0, 5.0);
```

  in *ChunkManager.cpp* to other parameters that are multiple of the values above. It can also create a simple Pyramid heightmap using the function $FillWithPyramids()$ typically used for the purpose of debugging.

- *ChunkMesher*: a header-only file used for processing how the meshing will be done. The meshing algorithm used can be a Greedy method or a naïve method, both implementations can be chosen by modifying the second parameter in the line

```
mesher::GenerateChunkMesh(*(*iter).second, mesher::GREEDY);
```

  in the *ChunkManager.cpp*, for example, `mesher::NAIVE_WITH_CULLING`. The greedy algorithm "joins" together voxels of the same type inside the same chunk, while the naïve simply extracts the visible surfaces of each voxel to form the mesh. The algorithm is currently being performed through multiple threads to accelerate the meshing process.

The greedy method concept was developed by Mikola Lysenko [11] and was adapted into the project. The method starts by analyzing each face of the world individually as if they were 2D tiles. It then starts by looking at the top-leftmost voxel of the "plane" and its type, and then checks if the neighbor to the right is of the same type. When it finds the first right-neighbor of a different type, it checks the next bottom line. If this line ends up at the same X coordinate as the last one, the two meshes merge into a single quad; otherwise,

they form different quads. The algorithms walks from left to right and top to bottom all the way, through all the faces.

On the other hand, the naïve method is quite simple. For each voxel, it checks whether each of its six faces is visible, i.e. it has no neighboring voxel occluding that face, drawing a quad if the face is visible.

3.2.6 *window* folder

This folder contains files to handle the creation and manipulation of the window, as well as the input (mouse, keyboard) from the user – using mostly SFML.

3.2.7 *main.cpp*

The main file of the project, where the game loop and game logic are contained.

3.3 Keyboard and Mouse Inputs

In order to facilitate, we will list the keyboard *hotkeys* and the mouse events the engine is currently using:

1: toggle between *Polygon mode* and *Wireframe mode*.

2: toggle drawing the normal vectors.

3: activate FPS camera.

4: activate Isometric camera.

5: activate Directional Light.

6: activate Point Light.

7: activate Spotlight.

f: toggle fog effect.

W, A, S, D: translate the camera to the front, left, back and right, respectively.

Left mouse click: create a gray box where the mouse is pointing (needs to be pointing to a box).

Right mouse click while moving the mouse: If the active camera is the FPS camera, it will change the pitch and yaw values of the camera. If in the isometric camera, it will only change the yaw values of the camera.

Middle mouse click: create a red line between the center of the camera and the point the mouse is pointing.

4 Results

In this project, we created one of the most important parts of a 3D game engine: the rendering engine. To communicate with the core aspects of OpenGL, we created a window and rendered 3D objects into it. Then we implemented the *polygon mode* and the *wireframe mode* mostly for debugging purposes – checking the meshes, checking if the voxels are perfectly aligned. Figures 3 and 4 illustrate both modes.

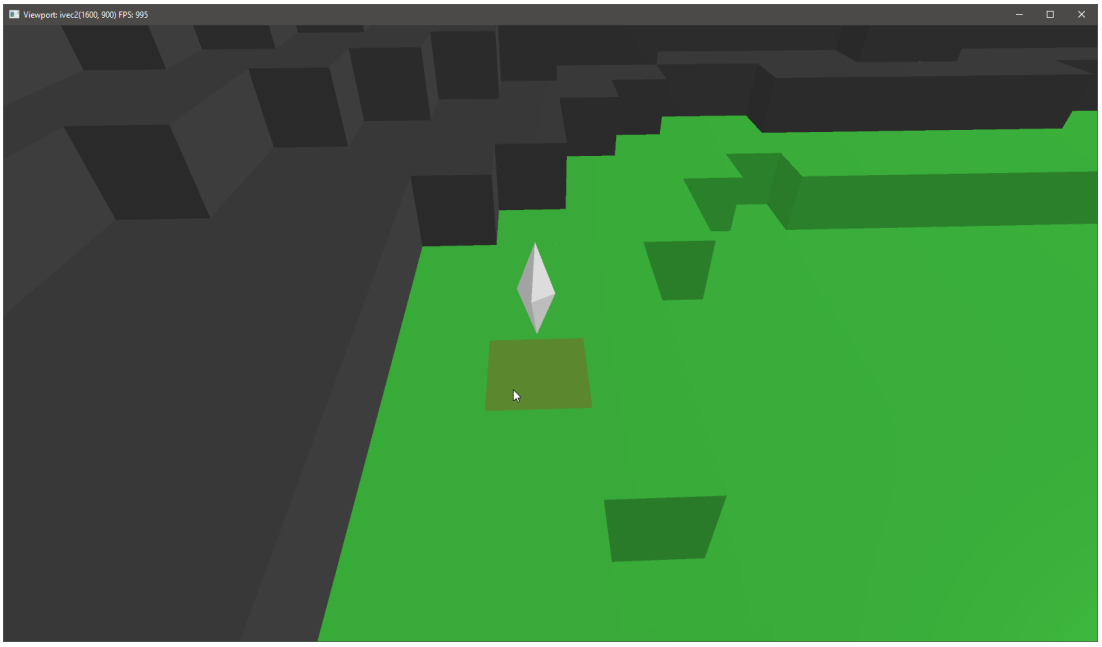

Figure 3: Polygon mode using FPS camera.

Afterwards, we were able to create procedural *heightmaps* using Perlin noise, which is an important aspect of our engine, since we had to define classes for volumes, chunks, meshes, then finally render the heightmap.

We also implemented the foundation for a map editor, as well as a very simple editor: when clicking the map, the user can create a new box where the mouse is pointing to. Figure 5 illustrates the creation of a new box in the map.

For mesh processing, we tried using two different methods: (described in Section 2.14) the naïve and greedy algorithms, then analyzing how they would affect the performance. Figures 6 and 7 show the meshing process using the greedy algorithm. Figure 8 illustrates the naïve algorithm and its differences from the greedy one.

Since we wanted to try different types of projection, we also created two different types of camera: one using perspective projection and the other using isometric projection. Figure 9 compares the representation of a heightmap using both cameras.

After that, we created basic light with three different types of dynamic – based on the camera's basis vectors and where the mouse is pointing to – light: *directional light*, *point light* and *spotlight*, as well as a fog effect. Figure 10 illustrates and compares each type of

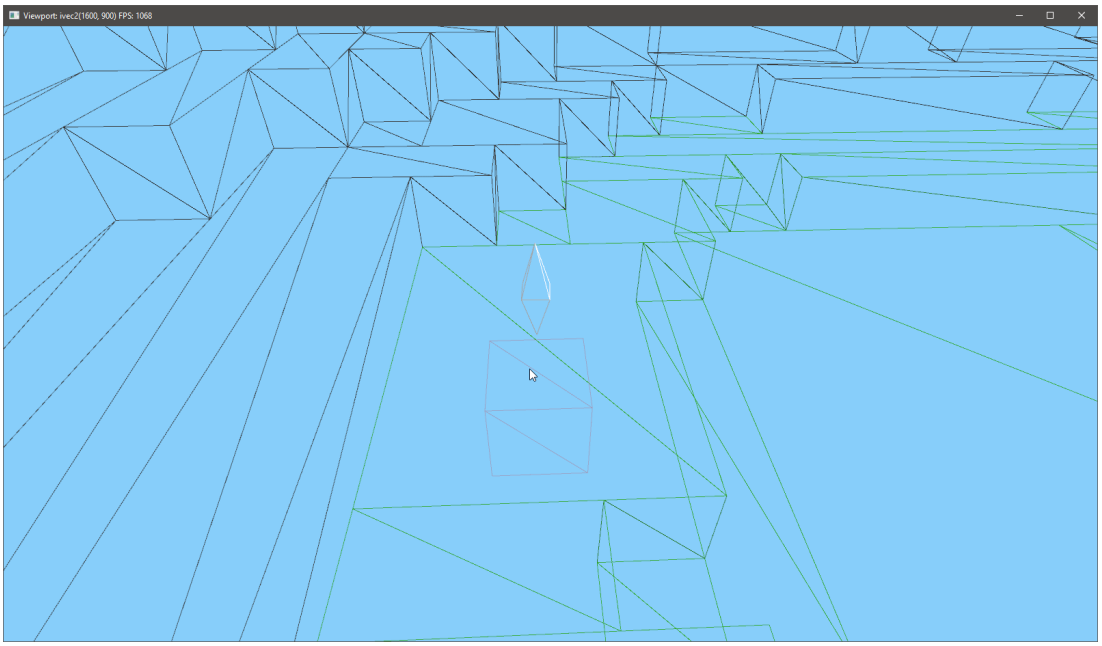

Figure 4: Wireframe mode using FPS camera.

light described.

4.1 Performance

On real time 3D applications, performance usually is measured in frames per second. In our engine, we measured performance by using a same procedurally generated scene under different volumetric parameters, for instance, chunk size and world size. We also tested the efficiency of the greedy mesher versus the naïve mesher, as well as we evaluated the advantages and counterpoints of one over the other.

We conducted a total of 6 measurements, 3 for each meshing algorithm, and each test varying the number of chunks and chunk size. We started by using a chunk size of 32 and a world dimension of 16×8×32 chunks on the first round, chunk size of 32 and world dimensions of 32×8×32 on the second and chunk size of 64 and world dimension of 16×4×16 on the second. Table 1 shows our results using the greedy algorithm and Table 2 shows our results using the naïve algorithm.

Greedy Mesher					
Chunk Size / World Dim.	Vertices	Triangles	Mesh Time	Avg. FPS	Mesh Time (1 chunk)
32 / 16×4×16	1335990	445330	11.31s	1235	0.02s
32 / 32×8×32	5297616	1765872	81.9s	150	0.02s
64 / 16×4×16	5203716	1765872	127.9s	970	0.20s

Table 1: Performance of the engine using the greedy mesher algorithm.

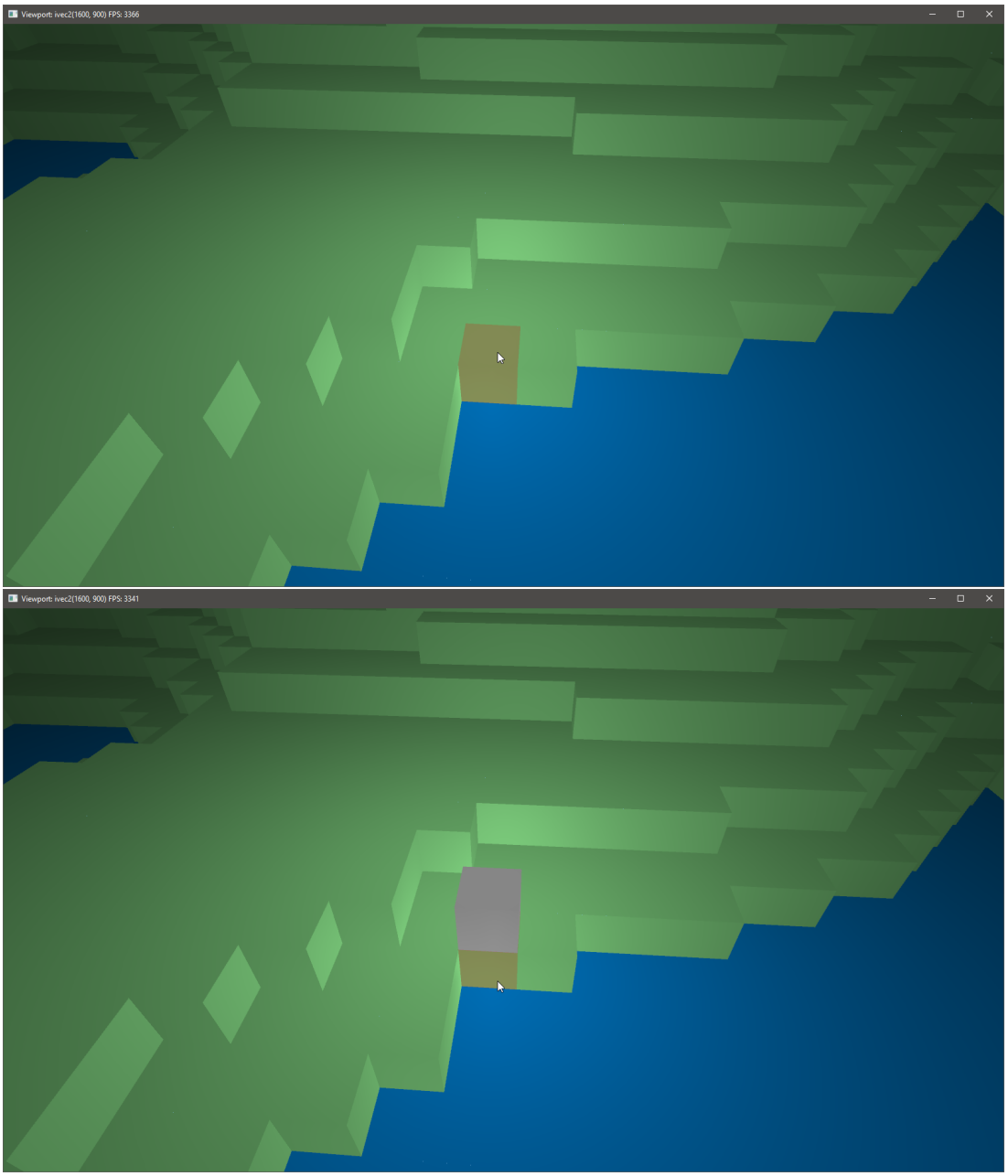

Figure 5: Picking (red overlay) a box and then creating a new one on top of it.

From the tables, we can analyze the average performance expected for the engine. First, by observing the number of vertices / triangles against the average FPS, we can conclude that the greedy algorithm performs well in this aspect, since the mesh it generates has close

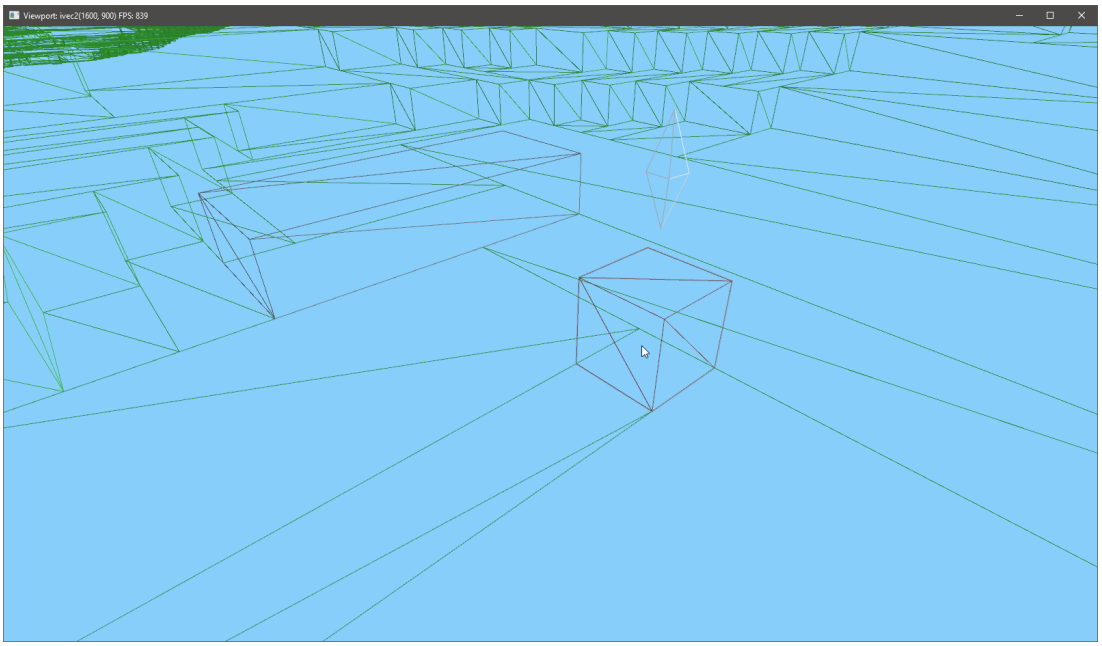

Figure 6: A single box using greedy method. By neglecting the heightmap boxes (green ones), in this case there is no difference from the naïve algorithm.

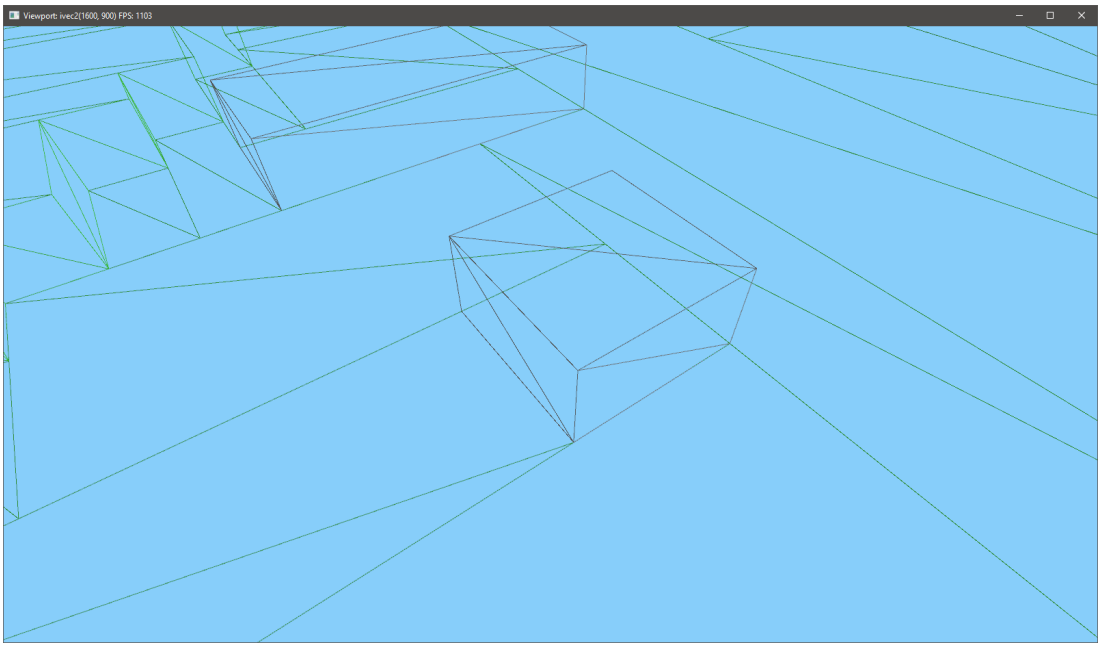

Figure 7: 4 boxes with the meshes together using the greedy method.

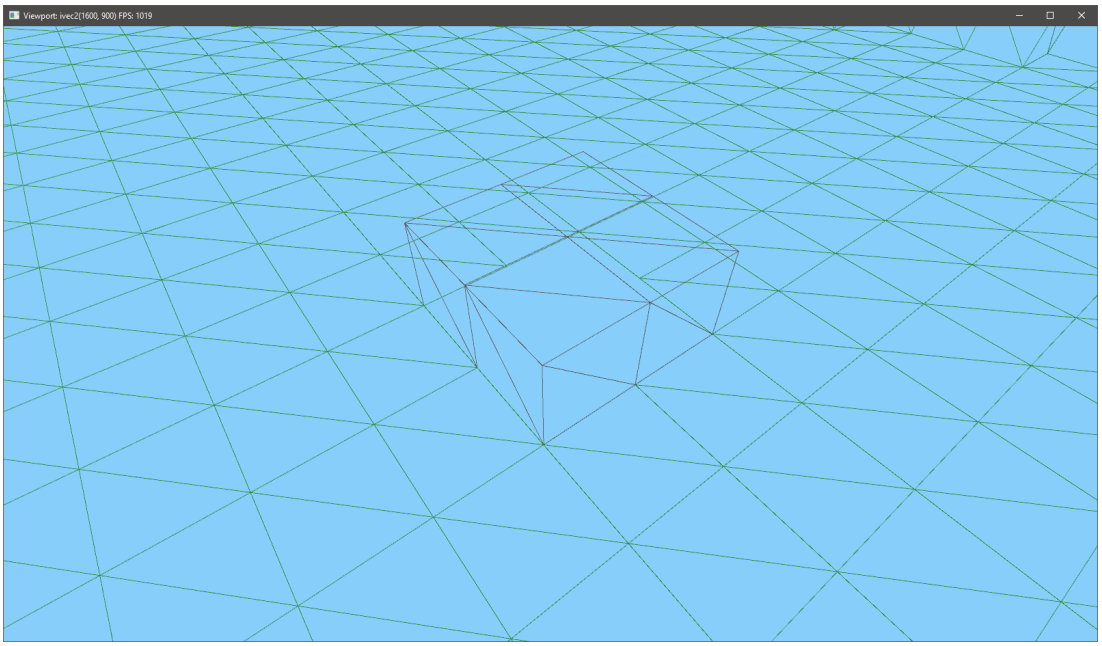

Figure 8: 4 boxes using the naïve method. Notice that neither the 4 created black boxes nor the heightmap has the meshes "joined" together.

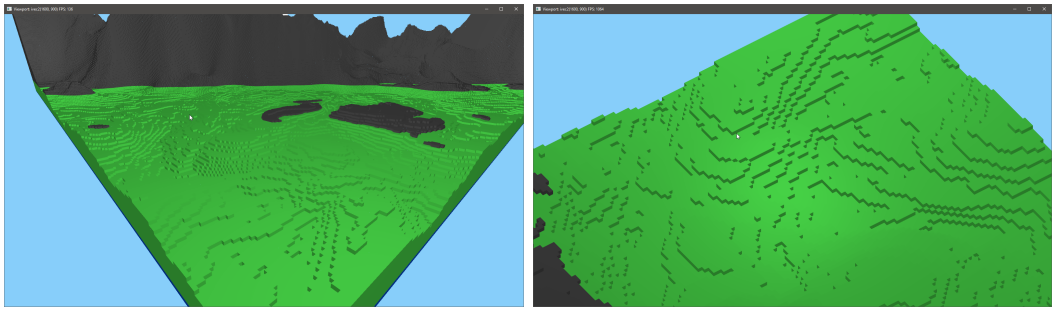

Figure 9: FPS camera and isometric camera of the same heightmap.

Naïve Mesher					
Chunk Size / World Dim.	Vertices	Triangles	Mesh Time	Avg. FPS	Mesh Time (1 chunk)
32 / 16×4×16	4766142	1588714	8.49s	849	0.02s
32 / 32×8×32	19126758	6375586	68.6s	120	0.02s
64 / 16×4×16	19126758	6375586	82.2s	330	0.20s

Table 2: Performance of the engine using the naïve mesher algorithm.

to four times less triangles compared to the naïve algorithm. This makes a great impact on the average FPS, since the lower the size of the geometry being sent to GPU, the lower the load, even with the same number of batches. We can see a significant difference in

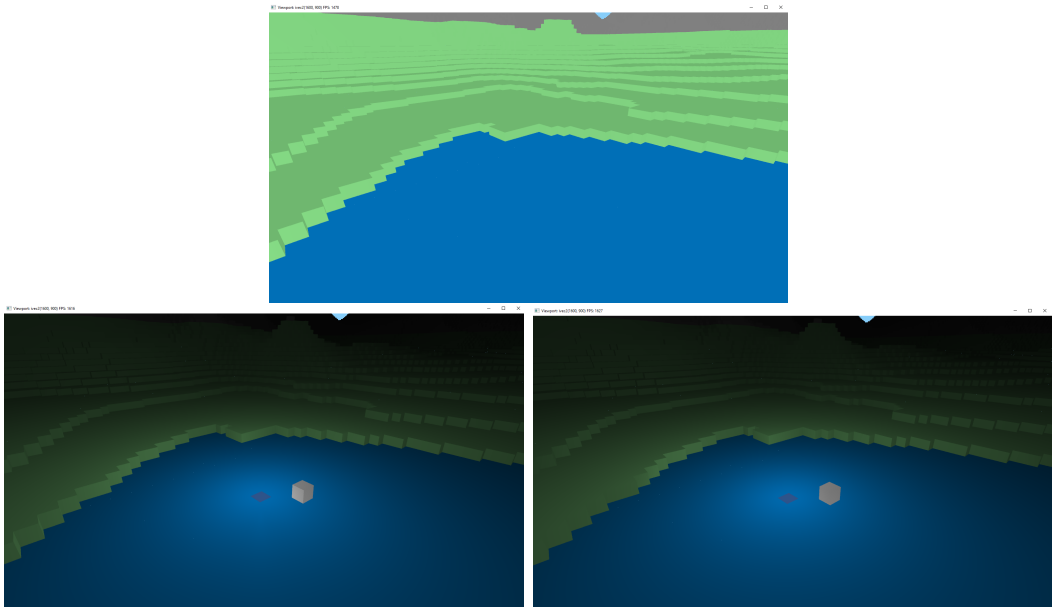

Figure 10: On top, an example of directional light. On the bottom left, an example of point light. On the bottom right, an example of spotlight. Notice how the spotlight only lights the cubes in front of it, in contrast to the point light.

the average FPS between the second and third measurements for each case, the second producing nine times less frames per second compared to the third. This shows that the current limitation of the engine is the number of different meshes being drawn at the same frame. On the second case, we have about 3300 chunks on the map, while on the third we have about 550 chunks, almost 1/6 of the former.

The second analysis we can make from the tables is concerning the impact of changing geometry in real time. Both algorithms show approximately the same response time when making changes to the existing geometry, on average 0.02 seconds for a chunk size of 32 and 0.2 seconds for a chunk size of 64. This shows that there is a sizable trade between gaining batch efficiency versus the time needed to make changes to that geometry.

Finally, the results were satisfactory, since in any of the tests we had a frame rate lower than 100 FPS, on average. On the worst case, when we had about 3300 objects being drawn at the same time using the naïve approach, we were rendering about 6.3 million triangles at a rate of 120 FPS.

4.1.1 Memory Efficiency

Memory efficiency was loosely measured throughout development. Since our engine deals primarily with volumetric data, it is expected that it will not be very memory efficient. Using as basis the three cases shown above, in the first case it would use around 700 MB of RAM depending on the algorithm and around 6 GB for the other cases. These numbers are constant for both algorithms since we do not store static geometric data - all this data

is erased from CPU memory as soon as it is sent to the GPU buffers.

5 Discussion

Overall, we have succeed to implement the most important aspects of a game engine. Creating appropriate code structure and design for the project and following established development methodologies was essential to reduce development time and alleviate time performing code review and refactoring. The use of the Visual Studio IDE also contributed with productivity since it helped us work collaboratively and maintain the project structure. It also saved us quite some time of debugging with its solid debugger.

Unfortunately, some of the features could not be done in time. Possibly the most important one, the *Map Editor*, although we were able to create the foundation for it, we could only make in time a basic block placement. Additionally, we have not migrated the software to Linux.

We also intended to use concurrent programming through in the most time consuming aspects of the code, but we could only do it in the meshing algorithm, because it is not as simple as we thought to parallelize OpenGL objects – which, again, means we were able to create the foundations for *multithreading* but could not use it to its maximum potential.

Memory efficiency is also an aspect for future work. While the basis for memory optimizations is in place, we could not do much beyond making static geometric data transient. Future work could deal with this problem by searching for a variety of strategies, such as data compression and data streaming.

Another important point is the visual fidelity of the engine. Although we are developing something similar to Minecraft and other "block engines", shaders can still be used to enhance visual fidelity. There are structures already in place to implement some of the most used techniques, such as shadow mapping, deferred rendering and other post-processing effects, but they could not be fully explored due to time constraints.

References

[1] Unity3D: (visited in 10/20/2016) `https://unity3d.com`.

[2] Unreal Engine: (visited in 10/20/2016) `https://www.unrealengine.com`.

[3] GameMaker: Studio: (visited in 10/20/2016) `http://www.yoyogames.com/gamemaker`.

[4] The Khronos Group: (visited in 10/20/2016) `https://www.khronos.org`.

[5] Learn OpenGL: (visited in 10/20/2016) `http://learnopengl.com`.

[6] Ken Perlin: (visited in 10/20/2016) `http://mrl.nyu.edu/~perlin/`.

[7] OpenGL Mathematics (GLM): (visited in 10/20/2016) `http://glm.g-truc.net`.

[8] OpenGL Extension Wrangler Library (GLEW): (visited in 10/20/2016) http://glew.sourceforge.net.

[9] Open Asset Import Library (Assimp): (visited in 10/20/2016) http://www.assimp.org/.

[10] Simple OpenGL Image Library (SOIL): (visited in 10/20/2016) http://www.lonesock.net/soil.html.

[11] Mikola Lysenko: (visited in 10/20/2016) https://0fps.net/.