Doom-Style Level Editor

Manveer Bajwa

The purpose of this project is to emulate id Software's "id Tech 1", also known as the Doom engine, which was used for games "Doom" and "Doom II: Hell On Earth". This project consists of two parts, a rendering engine and a level editor for the engine to render. The engine aims to use many of the techniques that were used in the Doom engine, while the level editor aims to mimic the format of other Doom level editors.

Manual

Rendering Engine

- W - move forward
- A - move left
- D - move right
- S - move backwards
- E - move upwards
- Q - move downwards
- Left - look left
- Right - look right
- Up - look up
- Down - look down
- M - Reload the map data

Rendering Engine

The first step of my project was to research the Doom engine, this was generally easy to find because most games are not published open-source, but since id Software open sourced the game's code on github, there has been a lot of references of how it works on the internet and in books. After reading up on the engine, I started looking into graphics libraries to implement what I had learned. I came across Simple DirectMedia Layer (SDL), and OpenGL, which of the two, I chose SDL. Before I begin, I want to note that I gathered a lot of information from the sources, but I only used what I needed.

In the beginning I formatted a plan to get the basics done, I would create a simple window and register some key inputs into the command line. I used the introduction page on the SDL wiki to find a good set of tutorials. I mainly used the SDL documentation from their wiki but whenever I could not find what I needed, I resorted to the aforementioned tutorials. After learning how to set up key parts of SDL, such as the SDL renderer, window, and event systems I had finished the first step of my plan. Now, I had wanted to write pixels on the screen, I created my own draw pixel function using SDL's draw pixel function which had parameters such as x and y coordinates and a color value to specify the pixel's color. SDL uses a coordinate plane starting at the top left, which means that the down direction is the positive y direction. I applied a coordinate abstraction, to move the coordinate system so it is in the middle of the screen by taking half of the screen width and height. To make sure that the game doesn't update constantly causing a lot of lag, crashing the program, I limited the framerate using ticks from the SDL. This way I would only draw pixels when a frame is called, of which there are only 20 in a second in the program. I then created a player structure to hold positional data of the camera, whenever a

key was pressed it would change the position of the pixels on the screen, i.e. if the player moved left, the pixels would move right, as to replicate the feeling of moving away from a point.

Before I get more into the program, I want to talk about how Doom works. In the Doom engine, there is no such thing as cameras, rather since the screen itself is treated as a camera, creating the first-person feel, it is also treated as the player. Rather than moving the player around, the world is moved and shifted around the player. All of the positional world data is moved whenever the player makes an action. If the player moves, then the world moves, if the player rotates, the world rotates, etc…. The Doom engine then takes this data, and what the player can see given their position and rotation, draws the closest walls from the player from left to right in vertical lines.

After implementing the moving of the pixels, I created world position vectors for x, y, and z. This would simulate 3D by taking the world position of the pixels and translating them to 2D coordinates on the screen x and y. Using the player's current look rotation, a pre-calculated table gives the sine and cosine values. These are useful for rotating the pixels around the player in world position. By treating the area around the player as a circle, we can move the x and y world positions of the pixels by the sine and cosine values. Now we translate the world positions of these pixels to the screen, which is 2D. We do this by taking the left/right direction, x, and up direction, z, in world position and dividing it by the forward direction y. This makes everything get translated to a certain point, which ends up being the top left, so for all our calculations at the end to change this point we add half of the screen's height and width respectively to the screen x and y positions of the pixels. Then we can draw all of the pixels in between these two pixels to create a line. We repeat this for another set of pixels at a higher z value. We can draw all of the

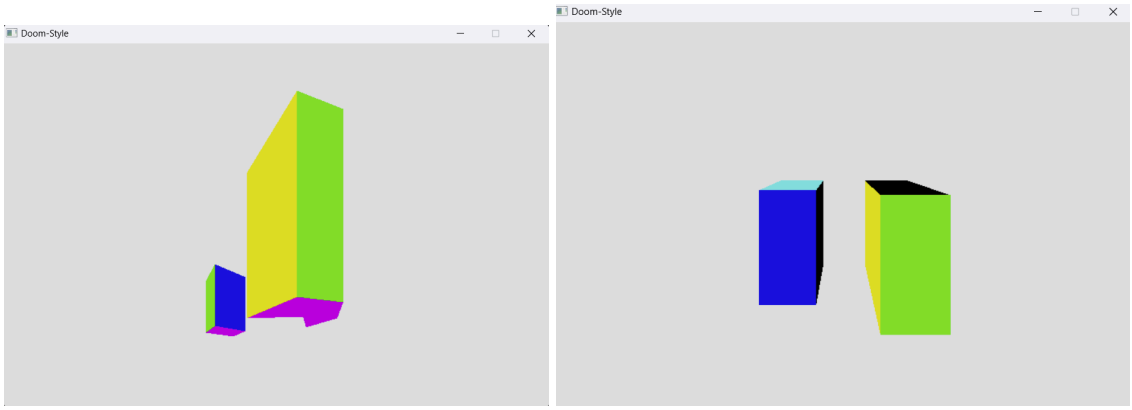
pixels going up from the bottom line to the top line to create a wall. If the wall starts to go off the screen as we move, we implement some clipping to disregard the position of any points past the screen.

This wall consists of the data of the 4 pixels/points, so I stored the data in a wall structure that had x1, x2, y1, and y2 values, alongside a color value to specify the color of the wall. In Doom, walls come together to make up sectors, so for a sector there could be a multitude of walls. Sectors are another structure that holds data which tells us which walls are in the sector, how tall each of the walls should be, and the color of the top and bottom surfaces. Sectors and walls will be discussed more in the level editor section. If the player is above or below a certain z position, we change the top line of the wall to be the top or bottom of the back wall, which allows us to draw the top and bottom surfaces of the sector.

Now we run into a problem called overdraw, since there is nothing stopping the renderer from drawing every wall at all times (other than the clipping on the sides of the screen), pixels are drawn over each other and sometimes we can see the back faces of objects, and see through objects themselves. We can calculate the distance of the walls to the players, to see the closest walls in the player's vision, and draw only those walls and ignore the rest, preventing overdraw.

Lastly, we need to load in the sectors and walls to make the process of the creating levels seamless. So far, only hard coded data was being used to test the walls and sectors, but with the level editor we should be able to export data for sectors and walls separately, and import them into their respective vectors in the rendering engine which are called every frame. All of the data points are integers for walls and sectors, so we can take the data from a simple grid and get everything we need. Using an input file stream we are able to open two files and load the data

into the vectors in an initialization function.

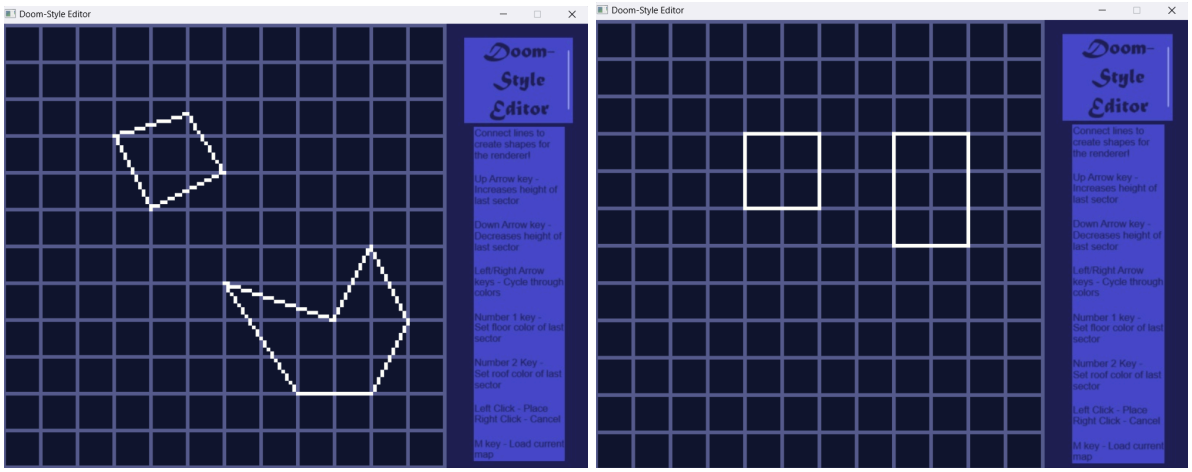


Level Editor

The second step of my project was to research Doom level editors. I first looked at the Doom engine wikipedia which told me a lot about how data is loaded to create the map. Everything is stored in something called a WAD file which stands for “Where is All the Data?”. The data consists of objects called linedefs, which there are many versions of like the one-sided linedef, or two-sided line def, these act as walls. Linedefs come together to make sectors which hold light levels, ceiling and floor values, and texture values. Sidedefs are separate from linedefs and sectors, but allow for textures to be stored for walls. In my level editor I chose to only implement two-sided linedefs and sectors, this will be explained later.

Knowing that I could get the data from points on a simple graph, and load that into the renderer I started by creating a new SDL program that acts as an editor. There are two parts of the editor that I split up using viewports, one for the graph, and one for the tool bar. I created a small SDL texture for the graph, and scaled it to part of the screen width and height, this made it so that pixels would seem much bigger when editing on the graph, making it easier to connect lines to form a sector. The toolbar was normally sized, and I created a custom texture and loaded

it into the program. It has some ui that explains what it is, and how to use it. Using what I had learned about drawing pixels from writing the rendering engine, I wrote a system to draw pixels and lines using SDL draw functions. I used the user's mouse position to draw pixels, and clicking to confirm that a line has been placed. Lines are drawn from each other until the user decides to close off the sector by placing a point back at the original starting point. This closes off the sector. There are a number of keybinds to change the color of the current wall, set the color of surfaces of the last sector, and to change the height of the last sector as well. Finally, I included a load function and keybind which loads all of the data and exports it into separate sector.WAD and wall.WAD files which can be loaded into the renderer.



What I Didn't Expect

At the beginning of the project, when I chose SDL as my graphics library for the rendering engine, I did not expect to hit a wall so soon. For some reason, I found it incredibly difficult to download SDL and implement it into a project. I had no sense of CMake files, how to build command line arguments, how to install the library, and where to put the library. I spent hours trying to figure out these things, which I could not seem to find many references on except

for in some StackOverflow forums. After installing the library and putting it into the project, I included the SDL header in my file, and tried to build it and run it. This did not work obviously because I had not included the right flags for the arguments in the build file for the compiler to look for the library files. I did not know this at the time but I thought since I was compiling with Visual C that the compiler could not find it, so this led me to downloading g++. All the references up to this point confused me rather than helped me, so I tried every option I had and learned trial by error. After a long time, I learned that there are such things as CMake files, and that when you build a file you have to include certain flags in the arguments for the compiler to include libraries in your file system. In Visual studio, there is a “tasks.json” that mimics a CMake file that is created when running a build task. In here, I learned that I can pass the compiler these flags that would allow the program to use the libraries. Finding these flags, and the order in which you put them in the command stumped me for a long time, but I persevered and completed the task, which let me finally work on the project. This was not the only thing that was a large time drain, but I wanted to talk about it here because it is the only prerequisite to do something like this project and it felt very unintuitive to someone who knew very little about these things. I also want to talk about how important it is to conserve data, especially when working with floating point numbers and integers. I ran into a problem when storing world position data, in which I was doing the calculations with integers that had some division going on. This truncated a lot of data because these values were being calculated as integers instead of floating points. It took me quite a while to figure this out, but I ended up using doubles and it fixed the problem.

What I Didn’t Do

One major thing that the Doom rendering engine is known for is the usage of BSPs, or Binary Space Partitioning. This is a technique that takes a map, splits it up into nodes by running a binary tree through the sectors, which are ordered by when they were created, then gives the renderer that map which lets it know what it should show the player when it needs to. I figured that this was too hard and too long of a process to implement and was outside the scope of the project, so I did not include it. I did not include all the different types of linedefs, aka walls, like one sided walls because the area in which the player can move is not defined in a starter sector, this would have taken more time, and limited the creation aspect of the software. I did not include the light levels of the sector, because there is no shading in the rendering engine.

Works Cited

id-Software. "ID-Software/Doom: Doom Open Source Release." *GitHub*, github.com/id-Software/DOOM.

"Where Developers Learn, Share, & Build Careers." *Stack Overflow*, stackoverflow.com/.

"SDL." *Simple DirectMedia Layer - Homepage*, www.libsdl.org/.

"SDL2/ApiByCategory." *SDL2/APIByCategory - SDL Wiki*, wiki.libsdl.org/SDL2/APIByCategory.

Sanglard, Fabien. *Game Engine Black Book Doom*. Sanglard, Fabien, 2018.

"Lazy Foo' Productions." *Lazy Foo' Productions - Beginning Game Programming v2.0*, lazyfoo.net/tutorials/SDL/index.php.

"Doom Engine." *Wikipedia*, Wikimedia Foundation, 7 Mar. 2024, en.wikipedia.org/wiki/Doom_engine.

3DSage. "Let's Program Doom - Part 1." *YouTube*, YouTube, 13 July 2022, www.youtube.com/watch?v=huMO4VQEwPc.

jdh. "Programming a First Person Shooter from Scratch like It's 1995." *YouTube*, YouTube, 5 Mar. 2023, www.youtube.com/watch?v=fSjc8vLMg8c.

"Creating a Doom-Style 3D Engine in C." *YouTube*, YouTube, 19 Jan. 2015, www.youtube.com/watch?v=HQYsFshbkYw.