Graphics Programming

Introduction:

For my project I decided to recreate a scene from Minecraft.

Blocks:

I started out with a simple plane to render objects on to but I was dissatisfied with the limitations of this approach.

I experimented with the idea of creating the blocks individually and constructing the environment based on this. I created a class that generated a single block and tried rendering a bunch of them together in a grid. My frames per second plummeted and I quickly realised that I needed to optimise how I was going to render the blocks.

I wanted each of my block faces to be made up of multiple quads so that I could assign multiple normals for better lighting calculations. For fun I decided to make each Minecraft pixel its own quad and have its own normals but decided to make the amount of divisions for each face of the block a variable so I could experiment with performance vs the visual effect of the lighting.

I decided to make use of vertex arrays. I needed to store data for my vertices, normals, texture-coordinates and indices. My block was made up of 6 individual faces so I created four 2D arrays that were size 6 (for each face) by the number of each data type I would need.

To calculate how many of each I would need, I counted the number I would need for a quad divided by 2, 3, then 4, manually, to get a sequence of numbers. I then used www.wolframalpha.com to generate an equation I could use to determine the size of the arrays I would need depending on the number of divisions I specified. Determining the size of the array beforehand allowed me to use standard C style arrays to store the data instead of vectors. Although minimal, this stopped the need for dynamic arrays and would speed up the generation of the blocks.

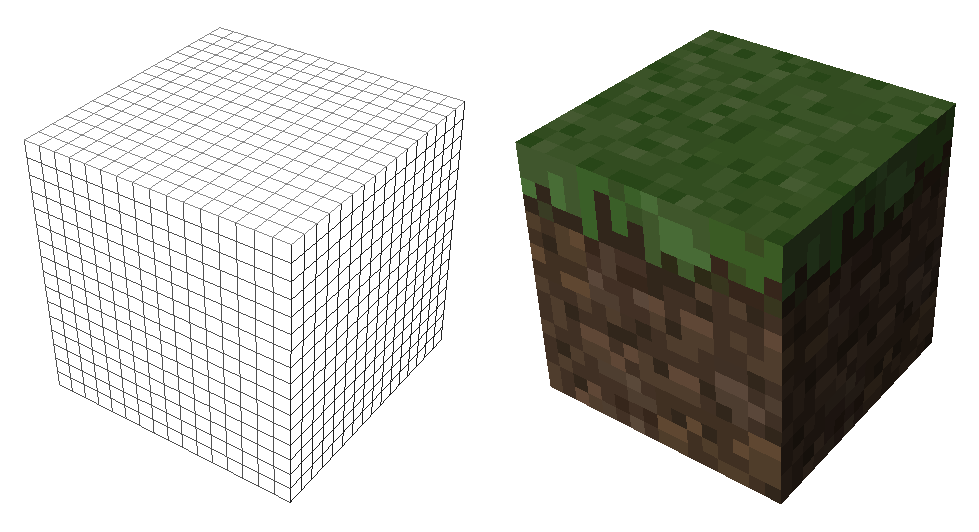
I only assigned one vertex and normal for each point on the face. The only exceptions to this were the vertices on the edges which I duplicated as I needed multiple normals, facing in each direction. Then for my indices array I re-used this data over and over again for each little quad. The texture coordinates were calculated by their position in the face divided by the amount of divisions.

The use of vertex arrays made rendering vastly more efficient as I was only having to do the calculations for generating the block once instead of every frame!

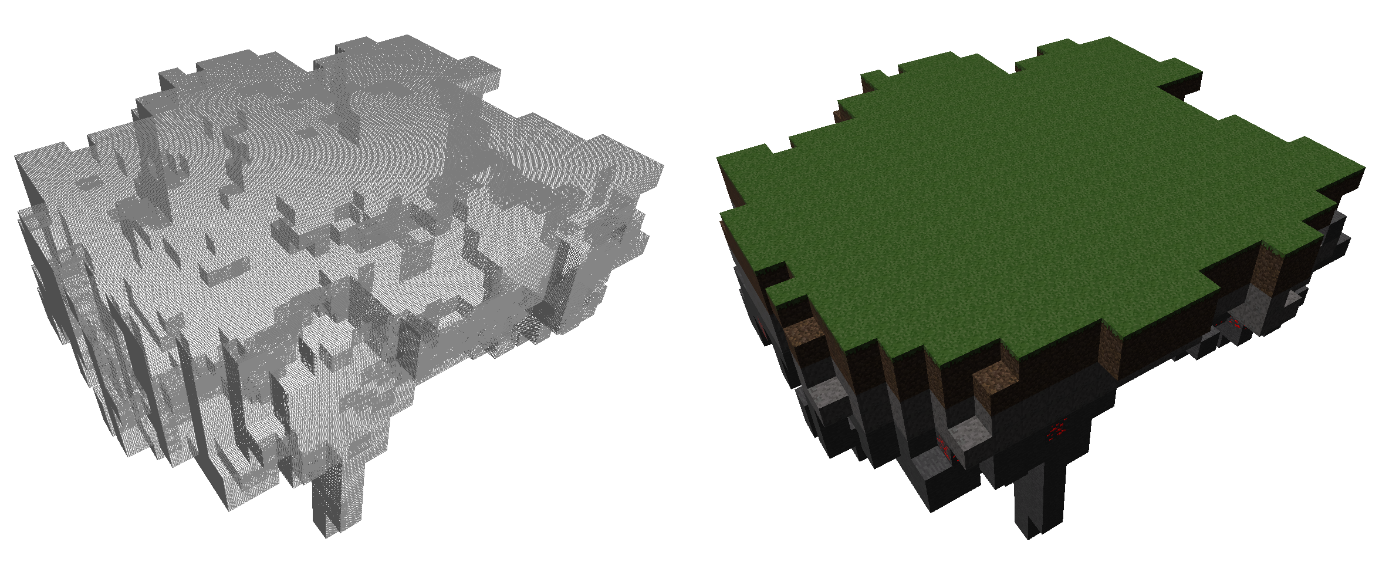
I was originally drawing each block at the origin and translating it to its position in the grid. This was expensive however as there were a lot of matrix calculations happening each frame. I solved this by passing the block class its position and taking this information into account when calculating its vertices. So now they would be positioned correctly without the need for transformations!

I realised that I was rendering all faces of the block when I didn’t need to. I noticed that I could only ever see 3 of the faces at any one time. This meant I could half the amount of rendering I needed to do! I altered my render function to take a Boolean array that signified which faces were visible. As I looped through the faces, I only rendered them if the corresponding Boolean was true.

I then created a calculateVisibleFaces() function that determined which of the faces were visible from the camera. As the blocks were all in a grid, they were all parallel to each other. They were all one unit in size and they were all drawn from the bottom left corner starting from the origin of the scene. I was able to simply compare the camera’s position with the position of the block. If the camera’s Z position was greater than the block’s Z position + 1, it meant it was in front of the block and I would never have to render the back face. If it was less than the block’s position, I would never have to render the front face. I did these checks for each axis. So now I had a block that only ever rendered its visible faces! This was a big improvement.



Then when I started rendering them together I realised they were unnecessarily rendering the faces between the blocks. These would obviously never be seen so I could make another optimisation. I updated my function to check if there was a block next to it in the direction of the face it was analysing. If there was, then it would set the Boolean for that face to false so it would not be rendered.

All of these optimisations made a vast improvement to my frame rate when rendering multiple blocks.

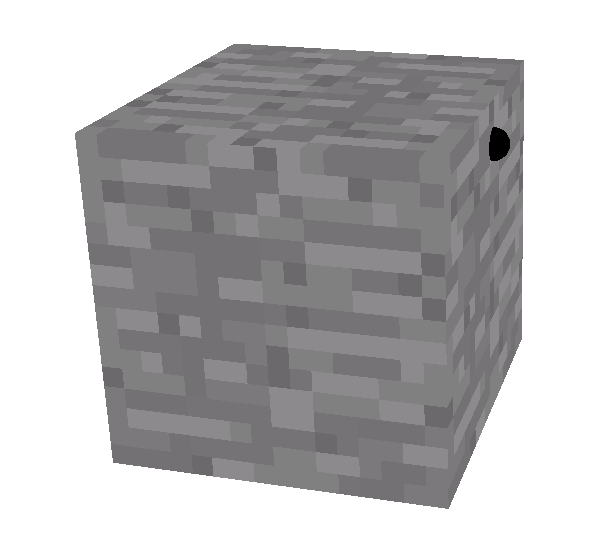
Ray casting:

After I had achieved creating geometry with blocks I was then left with the incredibly unattractive prospect of creating a scene by manually populating a 3D integer array with different values! I did not like the thought of this so I started thinking about different ways I could do it. I thought about drawing a pixel image for each layer of the grid but that still seemed pretty stupid. I thought about reading data from the actual Minecraft game and populating my array like that. I looked into this and failed miserably. I then thought maybe it would be good if I could generate them inside the application. This turned out to be relatively easy as I already had the code for generating blocks and it was not hard to write code to delete them.

I was now able to spawn blocks at the camera’s grid position but I was not able to see what I was doing very well. I was getting carried away by this point and started thinking about how to spawn them on the face of the block I was looking at. I had heard about ray casting and in our maths class we had been learning about how to calculate the point of intersection of a line and a plane so I went about trying to implement this in code!

I used the camera’s position and its forward vector to calculate the equation of the line. The equation of the planes were always very simple as they were all horizontal or vertical. I used the maths we had learned in class to calculate the point of intersection.

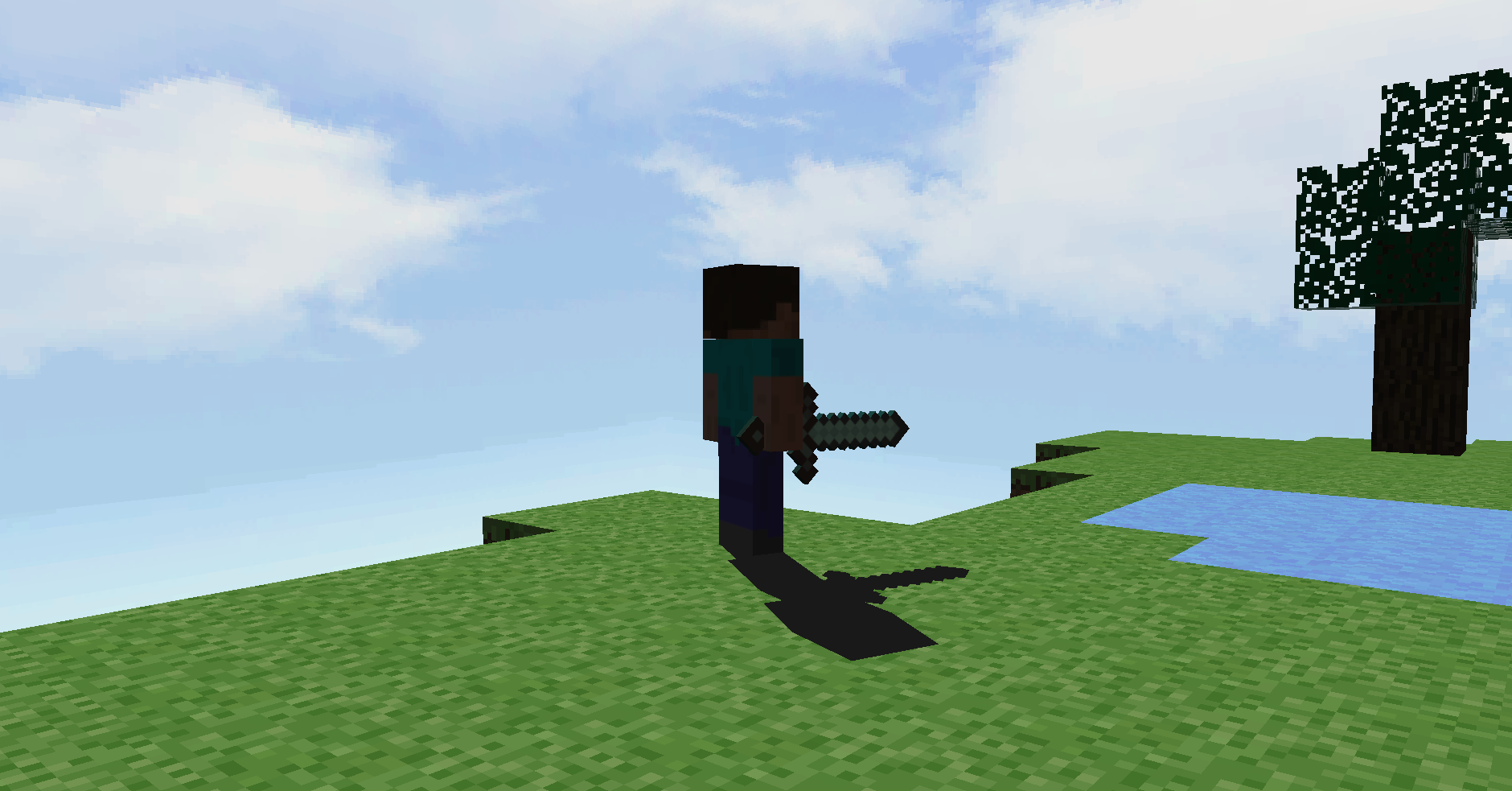
I then checked each of the faces of the block space the camera was currently in to see where the intersection was, where the magnitude of the vector was positive. (There would always be two intersections but I was only interested in the one in front of the camera.)

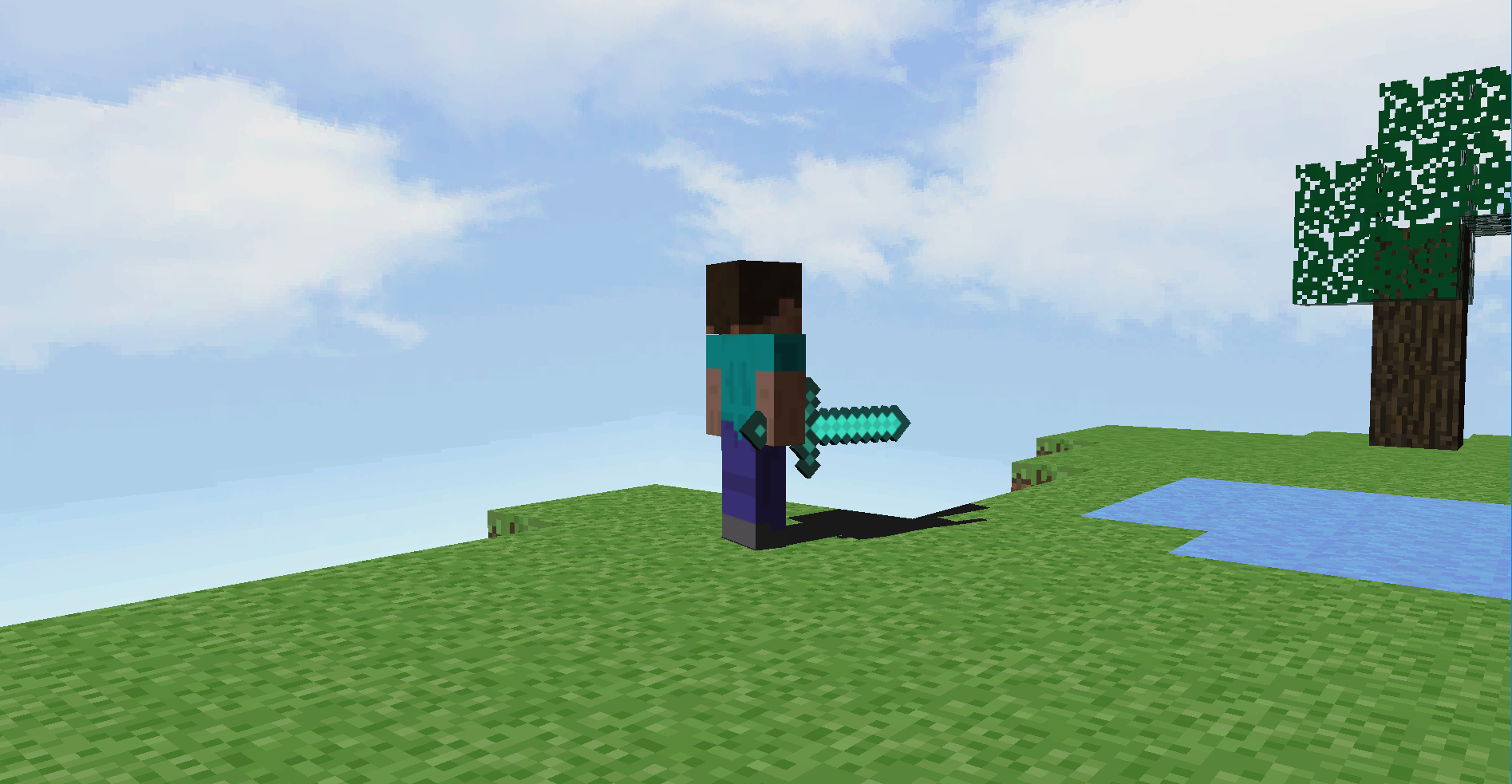
I then checked to see if there was a block in the space opposite the face that had been intersected. If there was then there was a collision and I spawned a block there! If there wasn’t, then I checked all the faces of the new block space and repeated until I found a block or I reached a certain distance from the camera.

Saving and loading:

I needed a way to save and load the grid state. I decided to create a function that would write the contents of my grid array to a txt file. I had learnt how to do this in my data structures and algorithms module. I then made a similar function for loading the data from the file and using the asci value of each character, minus an offset, to populate the grid array at the beginning of the program.

Shadows:

I decided to use planar shadows as I thought this would give the nicest visual effect while not being too expensive. The idea of creating stencils and multiple shadow objects for each face of each block seemed overwhelming. I was running out of time so I decided to compromise and make a function that would generate a stencil in the shape of the top layer of the environment. I achieved this by drawing multiple quads for the stencil in the positions of the blocks on the top layer. I called this function any time I edited the top layer of blocks.

I used the stencil inside the custom model class so I was able to render the individual cuboids and models as they were animating. I achieved this by applying the shadow matrix to them, disabling texturing and colouring them a dark grey. Originally I was using depth testing to avoid Z fighting but this caused issues with the shadows rendering in front of other objects in the scene. I abandoned this approach and simply raised the shadows a minute amount off the ground so it wasn’t visible but I got no Z fighting. The stencil worked perfectly and I got an effect I was very happy with.

Rainbow:

I created a quad for each segment of my rainbow. I added vertices on the x and y axis at the cos and sin of the angle multiplied by the radius plus the width of the rainbow. Then multiplied by just the radius. I then incremented the angle by theta and drew it multiplied by the radius and then the radius plus width. I repeated this for the amount of segments given. Theta was 2PI divided by the amount of segments.

The texture coordinates were very simple. It was just the 4 corners of the texture I was using for each segment. I changed the opacity of the texture in Gimp to make it slightly transparent. I made sure to enable blending and render it last. I disabled lighting for rendering this as it made no sense to have lighting affect a rainbow.

Reflection:

I made a special “water block” that was not textured but I instead filled with a pale blue color with some alpha to make it slightly transparent. I created a stencil for reflection in much the same way that I did with the shadow stencil. I looped through the top layer of the grid but only drew quads where there were “water blocks”. I updated this when I deleted or placed new “water blocks” in the same way.

I rendered the duplicated custom models upside down. They were rendered inside the stencil test so they were only rendered when you looked through the stencil. This created the illusion of reflection. I also rendered the skybox again upside down at the camera’s position with depth testing disabled.

Hierarchical modelling:

I used hierarchical modelling for my custom models (Steve, the cow and the pig!). I created a ‘Box’ class that handled creating a cuboid of any size and calculated the texture coordinates for any size image provided it was in a certain format. I used the matrix stack to construct the models. I translated each cuboid to its position in relation to its ‘parent’. I used a combination of translation and rotation matrices to achieve the animation of my models.

As my cuboids were drawn with their origins at the centre I had to first translate the limb cuboids to the joint, then rotate them (by an angle that I was incrementing and decrementing with time and by if the model was moving), then translate them back. This achieved a crude animation of the models that I was actually going for as it closely resembled the animations in Minecraft.

Imported models:

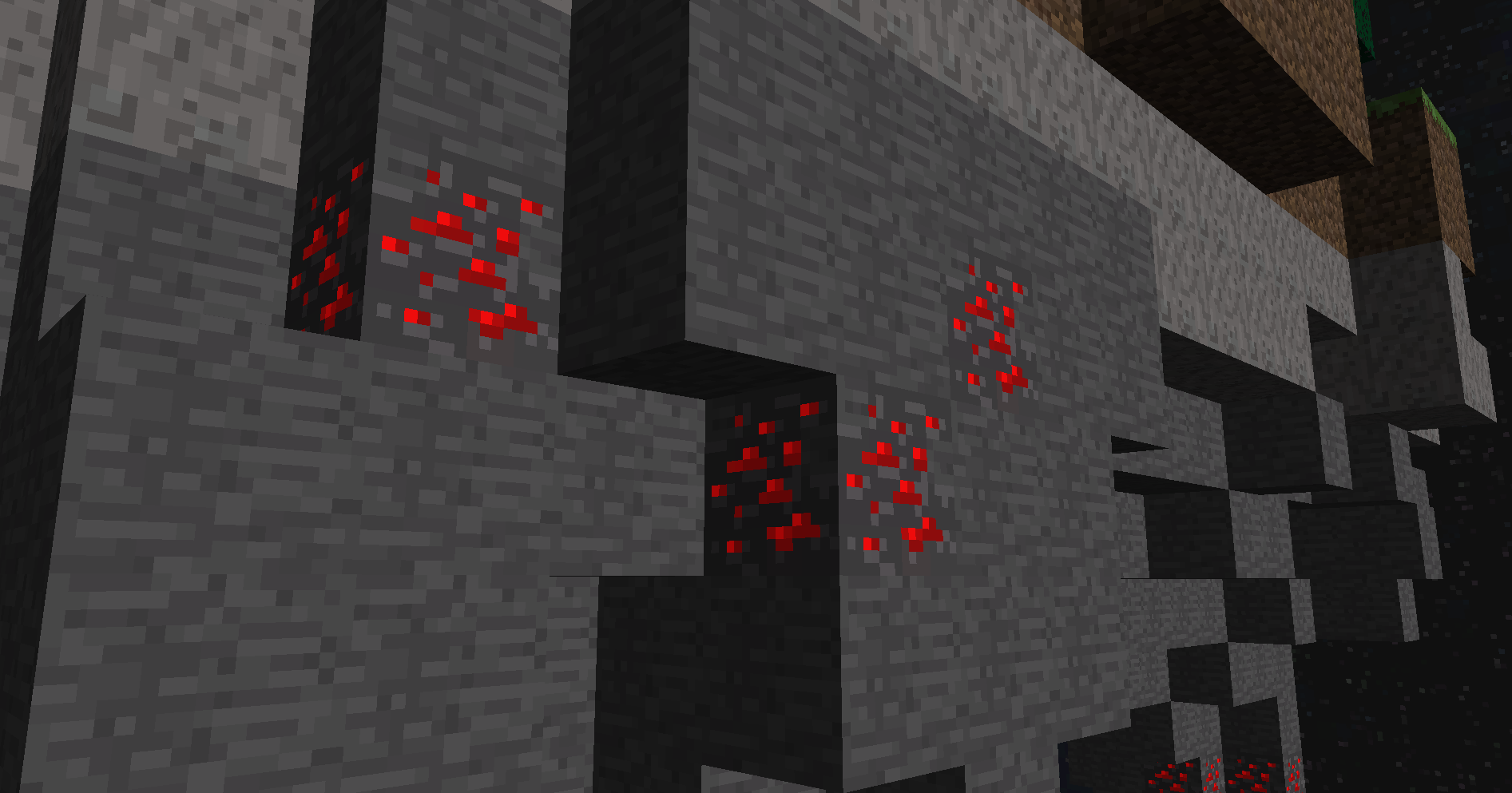
I downloaded 3 models from www.free3DM.com. They were perfect Minecraft models of the sword, pickaxe and axe. I had to import them into Blender and triangulate the models as they were originally built with quads. I scaled, rotated and translated the models into position. I did this inside my hierarchical modelling for Steve. I was able to place the tool in his hand and have it move with his arm as he walked! I was also able to include this in my shadow. I made it so you could alternate through his tools.

Cameras:

I actually only use one instance of my camera class but I use different parameters to determine how it is controlled. In retrospect it would have been nicer to have 3 separate classes but this achieves the same thing.

I have a first and third person perspective as well as a “Creative mode” where you can fly about. In first person the camera is translated to the player’s position. In third person the camera’s lookAt is set to the player’s position and the camera’s position is calculated using trigonometry and the pitch and roll variables. In “Creative mode” the camera is independent of the player and uses its own controls.

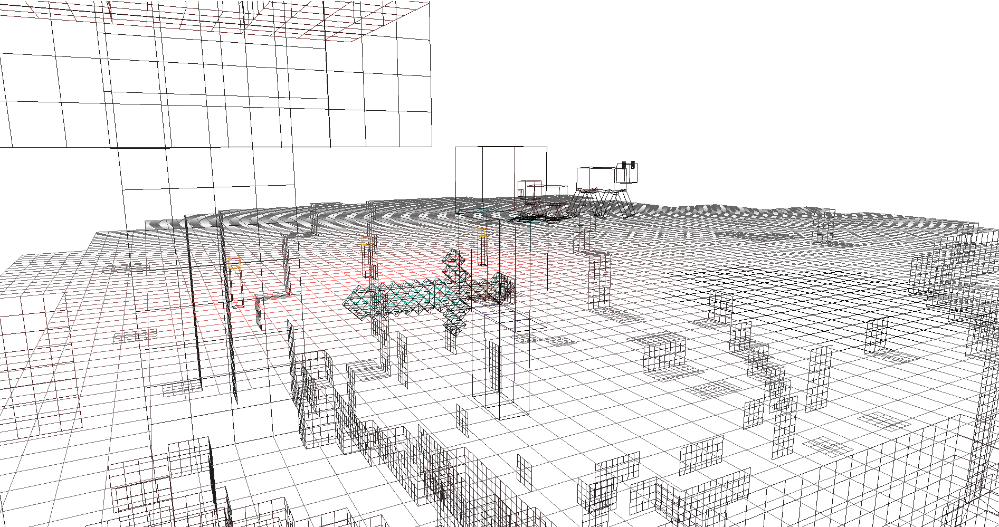
Materials:

For the “Ore” blocks I render two textures. The first layer has transparency where the ore was. The second layer is the inverse of the first with the ore in the place of the holes. I apply material components to the second layer to make it “reflect” more light and to appear shiny. This was to create appearance of the ore glinting in the light. I was very happy with the effect.

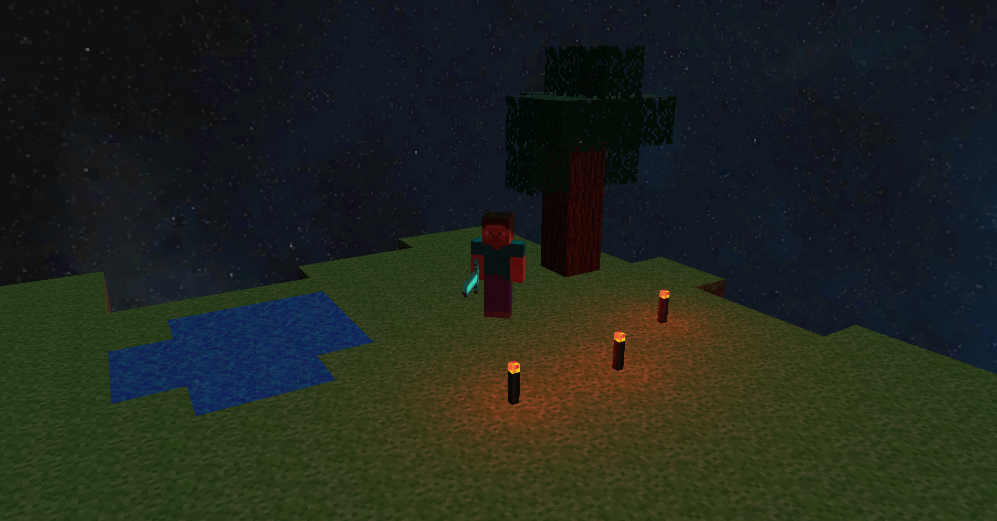
Cone:

I also made a hat out of a cone and a disk to play around with normal calculations. I created two vectors from three points on each of the quads on the cone. I then used the cross product of these for the normals. The lighting on it looked great but the geometry did not fit with the style of my scene so I made it hidden until you toggle it on with a hotkey.

Wireframe mode:

There is a wireframe mode that switches between all objects being textured and being in wireframe. This was incredibly useful when trying to optimise my block rendering.

Lighting:

I make use of directional light to simulate the sun and the moon. These rotate around the scene. I keep track of its position to use for the shadow matrix calculations. I use a spotlight to simulate a flashlight that originates at the player. In first person mode I use the camera’s forward vector for the flashlight’s direction. In third person I calculate a direction vector using the player’s angle. The hotkey to turn the flashlight on and off is ‘n’. I use point lights for my torches with a red ambient light to give a glow effect.



Controls:

The player is controlled by a combination of keyboard and mouse input. ‘s’ and ‘w’ move backwards and forwards. In first person, ‘a’ and ‘d’ strafe left and right. In third person, ‘a’ and ‘d’ rotate the player and ‘q’ and ‘e’ strafe left and right. You can also hold right click to control the player’s angle with the mouse. In “Creative mode” ‘e’ is up and ‘q’ is down.

Wireframe’s hotkey is ‘r’.

Toggling day and night is ‘p’.

Switching tools is ‘f’.

To save the state of the grid, it is ‘h’.

Dim the moonlight with ‘j’ to see the candle better.

You can press ‘y’ to equip Steve’s hat!

In “Creative mode” you can select different blocks with the number keys 1 to 9. You can delete blocks with the right mouse button and you can place blocks with the left mouse button.

To reset the camera it is ‘v’.

References:

I used [www.wolframalpha.com](http://www.wolframalpha.com) to generate an equation for a given sequence of numbers.

All the textures for my blocks and custom models were from [www.minecraft.novaskin.me](http://www.minecraft.novaskin.me).

I got my skyboxes from [www.opengameart.org](http://www.opengameart.org).

I got my imported models from [www.free3DM.com](http://www.free3DM.com).