



VOXEL TERRAIN

Development

ABSTRACT

This document provides an overview of the problems and challenges encountered during the development of the game component. Each problem or challenge is described along with the solution that was chosen and implemented. This is followed by a discussion of the advantages and disadvantages of the solution and its implementation. Some alternative solutions are also explored and compared to the chosen one.

Niklas Johansson
a24nikjo

Introduction

This document features a series of problems and challenges that were encountered during the development of the game component. The order in which they are discussed is based on the order in which I implemented them. This has the added benefit of providing a sense of what the development journey looked like as a whole.

The list of topics in this document also matches closely with the list of features that were planned for the game component. This is because those features represent the most interesting and challenging problems faced during the development. They will therefore be the main source of discussion in this document.

Mesh Generation

The first step to creating a voxel-like terrain became the mesh generation since it is hard to work on something that is invisible. The initial mesh generation was quite naive which was intentional. It is better to get something up and running before trying to make it perfect. The naive way is essentially to create six quads forming a cube for each block in the world. Each quad also has a texture applied to it that depends on the block type. This results in the desired voxel-like terrain that consists of cubes.

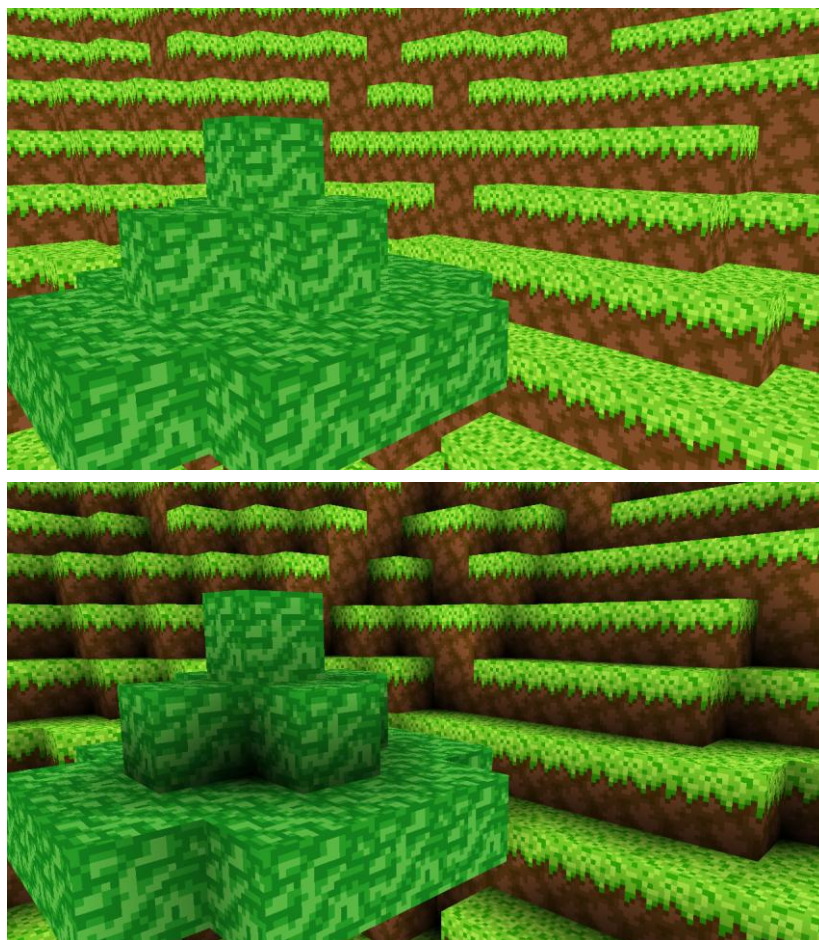
The main disadvantage of this approach is that cubes that are adjacent still create faces between themselves. These result in an unnecessary number of triangles being rendered. The solution to this is to check if neighbouring blocks are air or not and only create faces if the neighbour is air. The hidden faces then disappear which does not change the visual appearance of the mesh but decreases the impact it has on the performance. This removal of faces that are not visible can be called “face culling”. Culling is described in the book “Real-Time Rendering” (Akenine-Möller, Haines & Hoffman 2008, pp. 660-661).

Due to my previous experience and knowledge of rendering, I find this to be an obvious first step to improving the naive way of generating the mesh. The only downside is some additional checks which might increase the generation time. This however only affects the setup time while the performance gain will be continuous while the mesh is being rendered.

Ambient Occlusion

After I had successfully implemented the mesh generation, the next step became ambient occlusion. The problem with the plain mesh is that it is very hard to distinguish the geometry of the blocks without some form of shading. Ambient occlusion, which is described in the books “Real-Time Rendering” (Akenine-Möller, Haines & Hoffman 2008, pp. 373-385) and “OpenGL Shading Language” (Rost & Licea-Kane 2009, pp. 380-385), is a great way to solve this. It shades geometry with ambient lighting which makes it more distinguishable and adds depth. This makes ambient occlusion a perfect fit for the problem at hand.

Of course, a lighting system would also help with this problem, but it is far more complex to implement. Due to time constraints and the desire to quickly get something decent working, I decided to go with ambient occlusion over a lighting system. When researching the topic, I came across an article (Lysenko 2013) describing how one can implement ambient occlusion for voxel games. This turned out to be quite simple. The amount of light that a vertex should receive is directly calculated based on how many neighbouring blocks are solid. The fragment shader then darkens pixels based on the amount of ambient light it receives. This effect can also be tuned by choosing how much of it to apply. The result of applying ambient occlusion can be seen below. The top image has no ambient occlusion applied and the bottom one has the maximum amount applied.



The advantage of ambient occlusion is the increased sense of depth and the more distinguishable geometry. The only real disadvantage is the slight increase in the time it takes to generate the mesh. This is due to ambient occlusion being calculated beforehand and stored as a vertex attribute. The increase in vertex data is also a factor that needs to be weighed in. Overall, ambient occlusion improved the look of the terrain immensely.

Chunking

Now that the blocks of the world can be rendered, it was time to scale things up. To do this the world has to be split into manageable pieces. This is a core concept in making large worlds and is called chunking. Each piece is called a chunk and can be modified and rendered independently. To implement chunking, all I had to do was break up the grid of blocks into multiple equally sized pieces. These pieces of the grid represent the blocks that belong to each chunk. Each chunk also gets its own mesh which means that the whole world consists of multiple meshes instead of one giant one.

Chunking is almost necessary if you are going to support large worlds. It allows pieces of the world to update, render and be modified independently. The alternative would be to have the whole world as one object and one mesh. This would not be feasible as the time required by the mesh generation alone would make the game unplayable.

I did not have the time to implement it, but chunking also allows for the world to be multithreaded. Multiple worker threads could cooperate by, for example, building meshes for chunks in parallel. The downside of chunking is their border. Some things like ambient occlusion rely on neighbours. This means that when ambient occlusion is calculated, all neighbouring chunks must be loaded. This adds additional complexity that must be handled.

World Generation

The world was now larger, but it looked a bit boring since block types were selected completely at random. While it technically could be played, it would not be very interesting. The problem with random numbers is that they are very random and the terrain looks like a complete mess rather than any landscape. To solve this, more sophisticated solutions must be explored.

One traditional approach is to use noise and fractal noise to determine elevation. Both concepts are explained in the book “OpenGL Shading Language” (Rost & Licea-Kane 2009, pp. 435-443) and come with some interesting properties. Different types of noise will, of course, have different properties and in turn different advantages and disadvantages. One general advantage of using noise compared to other more exotic approaches like erosion or genetics is the speed at which it can be generated. The downside is that the result might look a bit uniform and boring. However, since I have previous experience with noise and a perfect-looking world was not my main goal of this project, the noise approach was chosen.

The type of noise that was chosen for this project was Perlin noise (Perlin 1985) which is explained by the article “Simplex Noise Demystified” (Gustavson 2015). It is a type of gradient lattice noise that has a highly desirable property. Given the same position, the resulting noise value will always be the same.

This is perfect for the type of terrain and world that this project uses. Chunks can be generated independently in any order and the result will always be the same given the same seed. The Perlin noise algorithm does technically not handle seeds, but one can easily be injected into the hashing phase of the algorithm. Another benefit of using Perlin noise is the smoothness and continuity of the noise which makes it a good fit for generating landscapes.

The process of generating the chunks then becomes a matter of using these noise values to select the block types. The way I used the noise was to first determine the elevation of the terrain at each horizontal position of the chunk. This value is then used to select the block type by checking how far below the elevation each block is. This distance is used to create layers by assigning intervals to different block types. Finally, to make the world a bit more interesting, some caves were carved out of the terrain. This was done using a 3d version of the noise where blocks that have a noise value higher than a specific threshold are cleared.

Building

With the addition of a more appealing world generation, the next logical step was to make the terrain modifiable. This is a core feature of the intended game and without it, the game would not be playable. To implement it, some prerequisites had to be dealt with. Firstly, chunks had to support changing block types. This was already supported due to the types being stored in an array. To change a block type, one simply replaces an element in the array with a new value. This however does not actually do anything since the mesh is only generated once. To remedy this, I regenerated the mesh of a chunk that has been modified. A simple list of “dirty chunks” was used and the meshes of those chunks would be rebuilt each frame. After the meshes had been rebuilt, the list was cleared.

The foundation of being able to build was now in place. The only thing missing is being able to point at a block and after some input, change that block’s type. This would allow the player to use the mouse to point and click to add and remove blocks. There is however an important step to this that needs to be solved before adding the input controls. How can the block that the player is looking at be determined?

The answer to this is ray casting. It simulates a ray shooting out from a point of origin in a specific direction. The result of a ray cast is then the point at which the ray collided with an object. If the ray reached its maximum length before it hit something, then the ray cast returns an empty result. A ray can then be cast from the position of the camera in the forward direction. The result of the ray cast can be used to determine if a block is looked at and which one.

The naive way of doing a ray cast is to take very small equally sized steps along the ray and check if a collision has occurred. This is very ineffective in a voxel world since most of the points along the ray will be within the same block space. A better way is to take advantage of the fact that the ray is cast in a voxel grid. This is exactly what the algorithm presented in the article "A Fast Voxel Traversal Algorithm for Ray Tracing" (Amanatides & Woo 1987) is doing. It uses clever thinking that results in a significant reduction in the amount of collision checks required. Only one check per block space that the ray enters is required. This results in a very fast ray cast with the only real downside of being slightly more complicated.

Collision

Now that the terrain both looked ok and could be modified, the player controls and physics became the next focus. Since this is mostly unrelated to the game component, only the collision between the player and the terrain will be discussed. This part is quite easy to implement since the player consists of a single AABB and the terrain is a grid of uniform AABBs. To check if the player is colliding with the terrain, only the block spaces touching the player AABB need to be checked. If any of these block spaces contains a block, the player has collided with the terrain.

I implemented basic physics by moving the player along one axis at a time. The collision detection then calculates the maximum amount that the player can move along an axis before a collision occurs. This value is used to cap the move amount which results in the player ending up flush against blocks that the player collides with. The reason for still letting the player move despite a collision is to prevent vibrations. If the player is stopped and then starts to move again, chances are that the distance to the block is shortened. This can make it look like the player is vibrating towards the block since the distance is shortened even after the first collision.

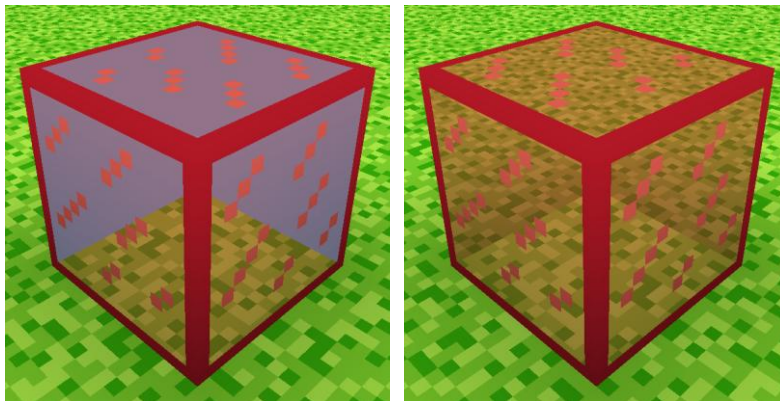
The problem with being completely flush against blocks is that floating point numbers are not perfect. When calculating the position that the player should have after a collision, the number might get rounded slightly. This can result in the AABB of the player touching the block instead of being flush against it. Obviously, this becomes a big problem as the player can now enter solid blocks. This also makes the physics behave in unexpected and undesirable ways. The solution is to have a slight margin between the player and the blocks when snapping the position. This prevents the player from entering solid blocks due to any errors from floating point numbers.

Another approach could be to ditch floating point numbers entirely and use integer physics. This would make the physics deterministic and more reliable. The player could simply be snapped to the collision point by adding either plus or minus one. This does however create problems of its own as the integers must be converted to floats at some points and the support for complex integer calculations could be sparse.

Transparency

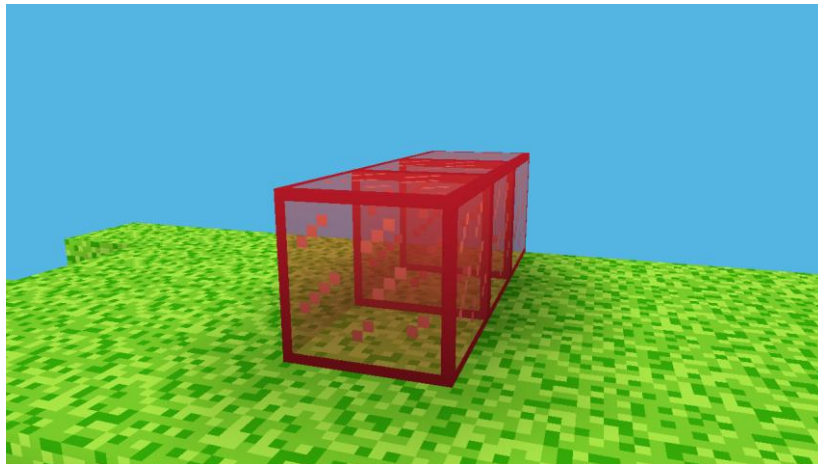
With a solid foundation to build upon, the next step is to add more complicated things. The first such feature is transparent blocks. These are more complicated to implement than one might think. This is due to the strict rules that must be followed when processing transparent surfaces. The problem of rendering transparent surfaces is discussed in the book “Real-Time Rendering” (Akenine-Möller, Haines & Hoffman 2008, pp. 134-141).

The rules mandate that all transparent fragments of a pixel must be processed and that they then must be blended in the correct order. This rules out depth testing since only one fragment would contribute to the final colour and the rest would be discarded. Even if I disabled the depth testing used so far by the rendering, it would not solve the problem. This would simply blend the fragments in the order that they were sent to the graphics card. This might not be the correct order which is sorted by depth. To fully solve the problem, the geometry would have to be sorted based on the distance from the camera. This order would, of course, change if the camera moved and the geometry would have to be resorted. The images below demonstrate how the current rendering process delivers different results depending on which angle the block is viewed from.



To overcome these challenges, the concept of “order-independent transparency” was introduced. It is a way of doing the sorting and blending of transparent surfaces on the GPU using shaders and fragments instead of sorting the geometry on the CPU. This results in the ability to send geometry in any order to the GPU and the shaders will figure out the correct final colour. The book “OpenGL 4 Shading Language Cookbook” (Wolff 2018, pp. 258-270) details how this can be achieved with OpenGL and GLSL. The result of implementing the algorithm found in the book is correctly rendered transparent surfaces.

The advantage of using this approach is as previously stated, the ability to ignore sorting on the CPU side. This gives great freedom in how the geometry is stored and sent to the GPU. The downside is some added complexity on the GPU side and the limited number of fragments that can be processed. I do however think these are minor in comparison to the freedom gained. Finally, the result of rendering transparent surfaces with order-independent transparency can be seen in the image below.



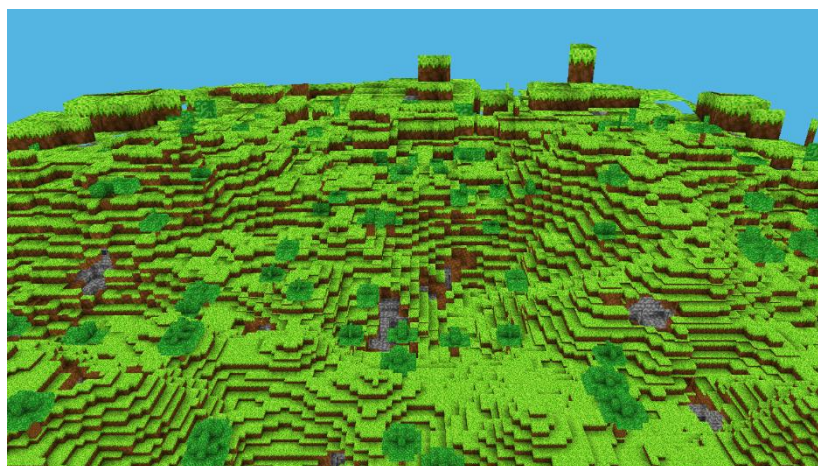
Level of Detail

The next problem that I wanted to tackle was “level of detail” or LOD. The concept is described in the book “Real-Time Rendering” (Akenine-Möller, Haines & Hoffman 2008, pp. 680-693) and is a rendering optimization. It works by rendering geometry further away from the camera at a lower level of detail. This reduces the number of triangles that need to be rendered and contributes to improved performance.

I wanted to apply this concept to the chunks so that chunks far away are rendered with fewer triangles. When I researched the topic, I came across the article “A level of detail method for blocky voxels” (Lysenko 2018). It describes how one can use POP buffers (Limper et al. 2013) to achieve LOD in voxel games. This approach places the model in a uniform voxel grid and when lower levels of detail are required, the vertices are snapped to the points on the grid. The size of the grid is determined by the desired level of detail and doubles for each level. This essentially becomes an octree where each level combines the smallest cells into larger cells. Some triangles will become degenerate when their vertices are snapped. This is due to at least two of the points becoming equal. These triangles can be skipped when rendering which achieves the goal of rendering fewer triangles.

The traditional way of doing LOD is to store multiple versions of a mesh at different levels of detail. The version to use is then selected based on the distance from the camera. The POP buffer instead stores only one version of the mesh but in a special format. We know that some of the triangles in the mesh will become degenerate after snapping at a specific level. This means that they should only be allowed to render if the level is more detailed than that level. The format of the POP buffer cleverly utilizes this fact and stores the triangles in a certain order. The triangles are sorted in the order that they would appear as the level of detail increases. To then render the mesh, one simply chooses the number of triangles that should be rendered based on the desired level of detail. The only thing left is to snap the vertices to the grid of the correct size in the vertex shader.

The result is a LOD system that only requires a uniform to be set and the total number of triangles to be rendered from the buffer to be modified. The system requires no additional intervention. It is a great way of doing LOD for voxel games since the world already consists of voxels. The main advantage is the minimal amount of processing required to display different levels of detail. Another advantage is the memory that is saved from only having to store one version of the mesh. The downside is that extra time is required to sort the buffer and unless some clever sorting techniques are used, any modification of the mesh requires a resort. The article which describes the use of POP buffers in voxel games also provided a way of interpolating the level of detail across the world. This makes the LOD linear across the world instead of jumping from integer to integer. My implementation of LOD was extended with this concept and the result can be seen below. For a live demo of this, see the separate video provided alongside this document.



Chunk Culling

The final feature that was implemented was the culling of chunks. The concept of culling is again explained in the book “Real-Time Rendering” (Akenine-Möller, Haines & Hoffman 2008, pp. 660-661). In this context, culling means that chunks that will not be visible to the camera should not be sent to the renderer. This reduces the amount of geometry that needs to be processed and in turn, improves performance. The implementation simply needs to check if each chunk intersects the camera frustum and if it does not, then it should not be sent for rendering. The result of this can be seen in the live demo contained in the separate video provided alongside this document.

References

- Akenine-Möller, T., Haines, E. & Hoffman, N. (2008). *‘Real-Time Rendering’*. Third Edition. A K Peters, Ltd.
- Amanatides, J. & Woo, A. (1987). *‘A Fast Voxel Traversal Algorithm for Ray Tracing’*. Available at: https://www.researchgate.net/publication/2611491_A_Fast_Voxel_Traversal_Algorithm_for_Ray_Tracing
- Gustavson, S. (2015). *‘Simplex Noise Demystified’*. Available at: https://www.researchgate.net/publication/281031452_Simplex_Noise_Demystified
- Limper, M. et al. (2013). *‘The POP Buffer: Rapid Progressive Clustering by Geometry Quantization’*, Computer Graphics Forum, 32(7), pp. 197–206. doi:10.1111/cgf.12227.
- Lysenko, M. (2013). *‘Ambient occlusion for Minecraft-like worlds’*. 0 FPS, 2013-07-03. Available at: <https://0fps.net/2013/07/03/ambient-occlusion-for-minecraft-like-worlds/> [Accessed: 2025-04-19].
- Lysenko, M. (2018). *‘A level of detail method for blocky voxels’*. 0 FPS, 2018-03-03. Available at: <https://0fps.net/2018/03/03/a-level-of-detail-method-for-blocky-voxels/> [Accessed: 2025-04-19].
- Perlin, K. (1985). *‘Image Synthesizer’*, Computer Graphics (ACM), 19(3), pp. 287–296. doi:10.1145/325165.325247.
- Rost, R.J. & Licea-Kane, B. (2009). *‘OpenGL Shading Language’*. Addison-Wesley.
- Wolff, D. (2018) *‘OpenGL 4 Shading Language Cookbook’*. Third Edition. Packt Publishing.