Blockycraft

Jonathan Beverly
jrbeverl

January 26, 2017

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this project was to create a Minecraft inspired world that utilized core functional techniques and demonstrate it with an interactive demo game.

The interaction demo is focused on demostrating key graphical components of the world that are pivotal to a Minecraft block world. Specifically these include collision, multiple blocks, world generation and consistent visuals.

This project was interesting and challenging because it involves using varied techniques to produce a final composite result. I learned how to effectively use Perlin Noise to achieve realistic world generation as well as techniques to describe the lighting and shadows of a dynamic world.

1.2 Overview of Features

The interactive demo is navigated using first-person controls, similar to first-person shooter games. A user interface allows the user to see the currently block status. Texture mapping allow for the blocks in the world to be of varying types.

The world is generated using Improved Perlin Noise in order to keep the world dynamic. Octave Perlin is used to control the presence of rolling hills or high altitude mountains.

Static collision is creating by detecting collision of the player with the scene, and is fairly straightforward. Dynamic collision was not completed, but it was intended that objects would exist in the scene that the player could collide with causing an effect similar to a knockback.

Transparency exists in the block objects to enable the glass and leaves block types. It uses a specific colour (magenta) to identify components that are transparent.

Keyframe animation exists within the clouds in order to move them in an interesting manner. Each of the clouds has their own unique set of keyframes to keep the cloud movement interesting.

Sound was added using SDL Mixer - sound effects are synchronized with movement in the game. Specifically the type of block the player is currently walking on determine the footstep noise. If the player is above blocks the 'overworld' song will play, if underground it will play a different song.

Additional light and shading factors were added into the final result in order to improve the visual aspect of the interactive demo.

These features can all be enabled at once to provide the full Minecraft experience. Most of them can be toggled by pressing a number key associated with the objective.

Chapter 2

Manual

2.1 Running the Program

2.1.1 Dependencies

In addition to the premake4 build system, the following libraries are dependencies of this application:

- GLEW - The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library.
- GLU - The OpenGL Utility Library
- GLFW3 - Multi-platform for simple, platform-independent API for creating windows, contexts and surfaces, reading input, handling events.
- Lodepng - LodePNG is a PNG image decoder and encoder.
- SDL - Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to graphics hardware.
- SDL Mixer - SDL mixer is a sample multi-channel audio mixer library.

2.1.2 Compilation and Execution

The program uses the **premake4** build system to handle the creation of compilation targets. To build the program, navigate to the top level directory and create the program makefile (*premake4 gmake make*). To build the specific project navigate to the source directory (*src*) and create the program makefile (*premake4 gmake make*). This process is outlined in the README of the source.

To run the program after building, simply run '**./Blockycraft**' from the command line. To quit the program press escape.

2.2 Controls

2.2.1 Mouse

The base controls for the game work similar to Minecraft. By moving the mouse in to change the angle of viewing in the 3D world. Note that the mouse is captured, so in order to regain control of the mouse cursor, you can quit the program by pressing escape.

2.2.2 Keyboard

To navigate the 3D environment, the application follows Minecraft's movement scheme. Using the 'WASD' keys the player can move about the scene.

The key 'TAB' makes your character fly in the scene, allowing viewing the scene from a different perspective.

The key 'SPACE' makes your character jump, allowing easier movement through the scene.

2.2.3 Special Keys

- Escape - quits the program.
- Z/X - Increases/Decreases the amount of visible global light
- J/K - Increases/Decreeases thge frequency value used in the Octave Perlin Noise
- C/V - Increases/Decreases the radius of chunks present in the scene.
- F/G - Shifts the current selected block either left or right.
- P - Toggles the particle visibility type
- O - Toggles the orientation component of the Cloud animation.
- L - Toggles the currently targetted block (by Crosshairs) to become a Light Emitting block.

2.2.4 Toggle Settings Keys

The number keys toggle specific features in the scene.

- 2 - Toggles the visibility of the user interface.
- 3 - Toggles the UV Mapping of objects within the scene.
- 4 - Regenerates the world and resets the current scene properties.
- 7 - Toggles the application of transparency.
- 8 - Toggles the visibility of clouds in the scene.
- 9 - Toggles the visibility of particles in the scene.

Chapter 3

Technical Components

3.1 The Scene

The scene is composed of a set of Blocks that are collected into a set of Chunks. Each Chunk acts as a 32x32x32 region that stores 32768 blocks of the scene. This spatial separate is for the purposes of making rendering of the scene easier.

See also Appendix A for a visual of the voxel world.

3.2 User Interface

The user interface is defined by two components. The first component is a cross-hair that represents the current sight vector of the player. The second component is a block is the bottom-left corner of the screen representing the currently selected block. The selected block is the type of block that will be placed when perform the mouse right click interaction.

The first component involves making us of the player rotational coordinates (x, y) that determine the current sight of the player. These two values are used to calculate the sight vector that is used to determine the block the player is currently interacting with through mouse interaction.

The second component can be altered by the keys F and G. These allow the type to be changed, allowing different kinds of blocks to be placed in the scene.

The user interface can be enabled and disable using the number key 2.

3.3 UV Texture Mapping

All blocks in the scene are textured using UV Mapping. All the block textures are currently defined in a spritesheet called Textures. This asset was acquired from the ‘Minecraftforum.net‘ submitted by the user CaptainJoe102 under the resource name ’Minecraft PE Original Texture Pack’.

When a cube is created using the ’construct-cube’ function (see cube.cpp), the UV values of the cube are set. This value is computed by determining the current (x, y) position the cube texture face is in the spritesheet. Using this the exact UV ratio position is computed by factoring in the total size of the spritesheet (2048) with respect to the size of each of the texture blocks (16 pixels). This is how the UV coordinate of a texture is computed with respect to the spritesheet.

The texture block indices are provided by a BlockDefinition structure (see block.h) that defines the index in the spritesheet of the specified face. These indices are defined by the method getBlockDefinitions (see block.h) that quickly constructs the mapping of block type with the positions within the spritesheet.

See also Appendix G and H for the visual representation of the world without UV Mapping.

3.4 Perlin Noise

The world in the interactive demo is generated through Improved Perlin Noise combined with Octave Perlin Noise. The implementation of Improved Perlin Noise (see pnoise.h) is a faithful interpretation of the java reference implementation provided in the SIGGRAPH 2002 Paper by Ken Perlin. Although some slight modifications have been made, the code is faithful to his reference implementation.

In my implementation, during each initialization of the PerlinNoise structure, it reinitializes the permutation vector (p) in order to keep the world changing between each world reset. The world can be regeneration with the number key 4.

For the implementation of Perlin Noise, the algorithmic work is done by the 'noise(double x, double y, double z)' method. In this method five key steps are performed:

- Construction of the Unit Cube.
- Relative X, Y, Z within the Unit Cube.
- Determining 8 cube corners for gradient vectors.
- Calculation of pseudo-random gradient vectors.
- Blending results from the gradient vectors.

Following through with the process, we first divide the coordinate into unit cubes to find the coordinates location within the cube.

On each of the 8 unit coordinates we generate a pseudo-random gradient vector. In Ken Perlin's *Improved Noise*, the gradient vectors are not completely random, these are actually picked from the point in the center of the cube to the edges of the cube. This results in a total of 12 potential vectors.

For each of the gradient vectors we take the dot product between a gradient vector and the distance vector provided by the input coordinates. With these final values, we linearly interpolate between the 8 values to get a weighted average between the 8 grid coordinates.

Two points of interest in this code is the 'fade' and 'grad' functions. The fade function implements a 'ease curve' in order to provide a smoother transition between gradients. This is because linear interpolation tends to produce an unnatural effect.

The grad function handles the process of calculating the dot product of a randomly selected gradient vector and the 8 location vectors discussed above. This is done by Ken Perlin using bitwise operations, but the code can be redesign to be less obtuse.

On a final note, the final result produced by the algorithm can remain relatively flat or similar to rolling planes. More ideally we wish to be able to provide a wider breadth of world types (mountains). To do this Octave Perlin was implemented where an amplitude and frequency were added. The frequency determines how 'squished' together the result is, while the amplitude determine the heights of peaks.

For the implemented code the amplitude is at a constant of 32 (the height of a chunk), while the frequency is variable based on user input. The initial low value (0.01) produces a rolling plains, while higher values (0.06) will produce more mountainous terrain. At sufficiently large values the world will produce an almost city-like landscape.

See also Appendix B for visual examples of the Perlin Noise results.

3.5 Static Collision

The collision of the player with blocks in the scene is primairly handled by a single method (see main.cpp). The method is player-collide and works by first determining where the player is in the world. With this information a relative position within the current block coordinate can be determined.

From here, we will compute based on the current position if the player is currently colliding with any of the blocks surrounding the current block coordinate. This means we check collision in all 6 cube directions, as well as upwards based on the height of the player (constant of 2).

If no collision occurs, then the player position is not altered. If a collision does occur then the player position is adjusted to be equal to the block center position with an adjusted padding. This padding is vital to ensuring that the player does not collide to closely with blocks in the scene.

The padding component discussed above can be visually scene by running towards the right or left edge of a block. If you hit in just the right position, the player will be adjusted to the block neighbour, allowing you to move past the block.

3.6 Dynamic Collision

Dynamic collision was not finished due to poor time management with respect to the graphics components. However, this was the current thinking with respect to its implementation.

Within the scene would be objects such as a sphere that would move about the scene. These objects would move about the (x, z) plane randomly based on any positioning algorithm. The y position would be determined based on an average height of the neighbouring blocks (9 blocks). In other words, the object would always try to remain higher in the y-coordinate than any nearby blocks in order to avoid collision.

When computing collision with the player, a check would be performed to determine if the player and object were within the same spatial chunk. If they were, a simple bounding sphere calculation would be used to determine collision. In the event of collision between the object and block, the objects y position would increment rapidly, causing it to fly into the sky much like a power-jump.

3.7 Transparency

Within the spritesheet any transparent pixel is set to magenta (RGB - 1.0, 0.0, 1.0). This was done so that within the shader it could compare the sampled pixel value based on the UV Mapping with the constant value of the magenta. If this comparison was true, then the shader could discard the pixel knowing that it would be a transparent pixel.

The transparency can be toggled by the number key 7. See also Appendix C for a visual display of transparency.

3.8 Keyframe Animation

Within the scene are a collection of flat white rectangle clouds. Each of these clouds in initialized when the scene starts with a set of positional and orientation keyframes. Using linear interpolation, the position of a cloud at a given moment can be computed.

After initializating in *uploadCloudDataToVbos* each of the clouds with their keyframes, they are rendered by *render-clouds*. The necessary linear interpolation is computed for each cloud, determining the current and next keyframes.

The timing component of the clouds is incremented by a speed factor with respect to the passage of time between frames. This ensures that the movement of clouds is bound to time as opposed to number of frames.

The cloud coordinate (x, y, z) is linearly interpolated to determine where to draw the cloud, as well as the orientation factor of the cloud. These are provided to the world transformation matrix for positioning of the cloud.

The visibility of clouds can be toggled by the number key 8. The orientation component of clouds can be toggled by the key ‘O‘. See also Appendix D for a visual representation of the clouds.

3.9 Particle System

The particle system works by first allocating the graphics buffer for the ’dust’ particles (through *upload-DustDataToVbos* in main.cpp). Once the buffer is allocated, we can render the particles in the system in the method *render-particles*.

In the method *render-particles* we iterate through all the particles in the scene to first determine if any have expired. If they have, they are removed from the list of particles.

Remaining particles are computed by determine the position in the world. The particles are scaled and also move in a random direction based on a factor of time. This ensures that the particles are moving away from the center of the generated cube. An important method is '*billboard*'. For the billboard effect to take place with the particles, we need to remove a rotational component of the world-view matrix. In this case we are considering a 4x4 matrix, where the bottom row follows the identity matrix, and the right column contains the translation vector. The 3x3 matrix within the 4x4 matrix handles the rotational component. If we set this value to the identity matrix, we will remove the rotational component. However, we wish to preserve the first columns properties, so we compute the $x^2 + y^2 + z^2$ of the first vector. Using this scalar factor we multiply it against the 3x3 identity matrix, to factor scaling into our final result.

When the key 'P' is toggled, it switches the particles to a single large block that has a longer expiration period. This is in order to show that the particle works using billboarding techniques, as this is not clearly visibile when working with particles in their normal form.

The visibility of particles can be toggled by the number key 9. The visual state of particles can be toggled by the key 'P'. See also Appendix E for a visual representation of the visual states of particles.

3.10 World Sound

An important objective of the interactive demo was to provide realistic sound based on the current position of the player. This comprised of two components. The first involved the presence of an overworld and underworld song based on whether the player was above or below a block. The second component involved determining the current block that the player was moving over, to play specialized footstep sounds.

This objective was possible with the 'hit-test' method. By firing a hit ray directly down we are able to detect the type of block the player is currently over. If movement conditions are met, then we can play the footstep sound. This is based on a pseudo-random index to determine the footstep variant to use.

The 'hit-test' method also enables firing a hit ray directly up, in order to detect whether the player is considered 'underground'. If the player is underground, then we can instruct certain music to play. Otherwise we default to the overworld music.

The 'hit-test' method works by determining where the player currently is in the world, as well as the hit ray direction vector. From this, the code iterates through nearby chunks, as well as the current, if within reasonable distance range as the ray can intersect with cubes in other spatial chunks.

The hit-check works similar to that of player-collide, in that we round the position to the nearest logical block determining if the type is collidable. If not, we continue to shift the ray forward based on the direction vector for each logical block segment (length of a block).

The footstep sounds were provided by the software resource 'Minecraft Sounds - Footsteps' by a user called Daenth from the online forum 'blockland'.

The footstep audio cannot be disabled at present.

3.11 Bonus Objectives

Three graphical bonus objectives were attempted in order to provide more realism to the world.

3.11.1 World Lighting

The world was given a static lighting component that factored into the world. This lighting value is known as 'time of day', and provides the overall lighting factor of the scene. In this method the value of total lighting factors in the composite of light.

It is important that the world light is not too bright or too dark, as such the potential contribution value is capped within the range of [0.2, 0.5] for the light values. This value is then factored in with respect to the diffuse colour, in order to produce the final colour output.

The final lighting value of the scene is a composite of diffuse, ambient occlusion and daylight with the fragment light from light emitting objects factored in for each step.

All computation with respect to world lighting is performed in the '*block.fs*' fragment shader.

The world lighting value can be incremented/decremented by the keys Z and X.

3.11.2 Light Emitting Cube

The light emitting component of the world makes use of a light-map for storing the light contribution of an individual cube. This can be seen with 'light-maps' and 'lights' variables in a ChunkGenParams and Chunk (see world.h). These variables are used to store lighting factors about the world.

The key code to distribute the light is handled by 'light-fill' in main.cpp. In this code, it handles the process of distributing the light throughout the scene by spreading out in the 6 cube directions. This creates a diamond-like light emitting cube, but follows with the Minecraft style of lighting.

The 'light-fill' method is part of an important piece of code called '**compute-chunk**'. This method performs a bulk of creating the graphic buffer for drawing a spatial chunk. As it handles the setting of light, texture mapping, shaders and ambient occlusion. The specifics of this method will be covered in more detail later.

A cube in the world can be toggled as a light emitting cube using the key L. See also Appendix F for a visual representation of Light emission.

An excellent resource for understanding light fill algorithm more is Seed of Andromeda. See the acknowledgements for this resource.

3.11.3 Shadows and Ambient Occlusion

The bulk of shadow and ambient occlusion code is handled in the method '**compute-chunk**'. This method makes use of the '**occlusion**' method found in occlusion.h.

Specifically when creating the graphics buffer for a chunk, we have to follow a process of computing variables for each step. For better understanding of the following, the Appendices G and H will provide a visual help to understanding shadows and occlusion in the environment.

- Determine if the scene has any lighting
- Determine the opaqueness of blocks (what is visible?)
- If a light exists, compute the light strength in the scene
- Determine visibility of each face of a block in the scene
- Compute the ambient occlusion based on light and shading
- Construct the cube from the compiled entries

The initial steps are fairly straightfoward. First we iterate through the chunks to see if any have any light data within them. If they do not, we have no lights. Next we iterate through all the blocks in the scene, determining which are opaque so that they will be draw.

The next steps involved propagating light throughout the scene as defined in 'Light Emitting Cube'. Essentially we iterate through all the entries in the cube, and if necessary propagate light from a light emitting position.

As we are constructing the buffer based on vertices in the scene, we first need to determine which sides of the block are visible based on the neighbouring blocks. Once this is known, we can then begin to compute the shading and lighting factors as they influence the visible primitives.

In the case of lighting, for each cube we need to consider all 3D neighbours with a distance of 1. This means we need to consider a 3 by 3 by 3 cube around the current block (27 cubes total including self). The shading component here is interesting as for each of the surrounding neighbours, we determine the shading

factor cast on it by blocks above it. This is done by iterating up a certain height (y-coordinate) and if any block is visible (opaque), we factor in a shading value based on the height.

With these variables we then compute a occlusion factor. In Minecraft-like worlds, the calculation for occlusion is significantly easier as the effect that the environment can have on the blocks is more limited in terms of number of rays. Specifically, we only need to consider what kind of contribution nearby blocks have to the currently vertex based on positions. In this world we have to consider when blocks are present on the sides of a vertex, and when a corner block is present. These three blocks greatly influence the ambient occlusion value for the vertex.

In the method 'occlusion' you can see this process of performing a lookup to find neighbour positions based on the face and vertex we are currently working with. The lookup tables (*neighbour-lookup-table*) work by indexing the cube face (cell) and vertex as the first two dimensions. The final dimension is for determining the specific cube positions that influence the specified vertex. The occlusion provides a curve factor but for the purposes of my implementation is linear. This is meant to control the shading factor brought about by the environment influence (the position of cubes).

When determining the light and shading values, we perform a similar lookup to occlusion but we must also factor in the lighting factor for each of the vertices in a given face. As each of these contribute in a manner to each of the vertices in the face.

A cube can be placed at a higher level in the scene to show the effect of shadows on the below cubes.

An excellent set of resources for understanding lighting is the blog 0FPS, LetsCode35, and a wrute-up by Florian Bsch. These references are available in the acknowledgements.

See also Appendix G and H to better understand the example cases mentioned above.

3.12 Implementation

3.12.1 Code and Considerations

In the code the 3 dimensional list are treated as single dimension arrays. The index for the single dimension array is computed based on an equation of the form $y * 2w + x * w + z$ as the width and height component are equal.

Instead of using a proper **List3D** implementation that abstracts away these technicalities, I use the existing arrays and a macro for indexing.

- Settings of the world are defined in settings.h, but not all values are recorded in the settings.
- The flag for toggling UV mapping is helpful for visualizing the lighting and shading of the environment.
- Testing and verification was all manual which means there are some strange bugs with respect to ambient occlusion of the code.

Chapter 4

Conclusion

4.1 Moving Forward

There is lots of room for improvement for the almost all components of the code. Especially with respect to the development of lighting and structuring of 3 dimensional data with the current code.

The spatial chunks are not multi-threaded, meaning that drawing them in the scene takes a very long time for sufficiently large number of chunks.

A lot of the code is written to be functional as opposed to expressive. As a result, a lot of the code can seem to be obfuscated by the current implementation. Examples of this are magic numbers or numeric assignment with no clear reason as to why it is occurring.

4.2 Acknowledgements

I would like to acknowledge a collection of blog posts by voxel world enthusiasts who provided the basis of understanding for a wide variety of concepts.

- Florian Bsch - `http://codeflow.org/entries/2010/dec/09/minecraft-like-rendering-experiments-in-opengl`
  `#ambient-occlusion`
- Let's Code - http://www.sea-of-memes.com/LetsCode35/LetsCode35.html
- 0FPS - https://0fps.net/2013/07/03/ambient-occlusion-for-minecraft-like-worlds/
- Seed of Andromeda - https://www.seedofandromeda.com/blogs/29-fast-flood-fill-lighting-in-a-blocky-voxel-game-pt-1
- ByteBash - http://bytebash.com/2012/03/opengl-volume-rendering/

Bibliography

[1] Perlin, K. 2002. *Improving Noise.* Computer Graphics 35(3).

[2] Perlin, K. 2004. *Implementing Improved Perlin Noise.* GPU Gems

[3] E. Hastings, R. Guha, and K. O. Stanley,. *Neat particles: Design, representation, and animation of particle system effects,* Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG), 2007

[4] P. Shanmugam, O. Arikan. *Hardware accelerated ambient occlusion techniques on GPUs.* SIGGRAPH 2007.

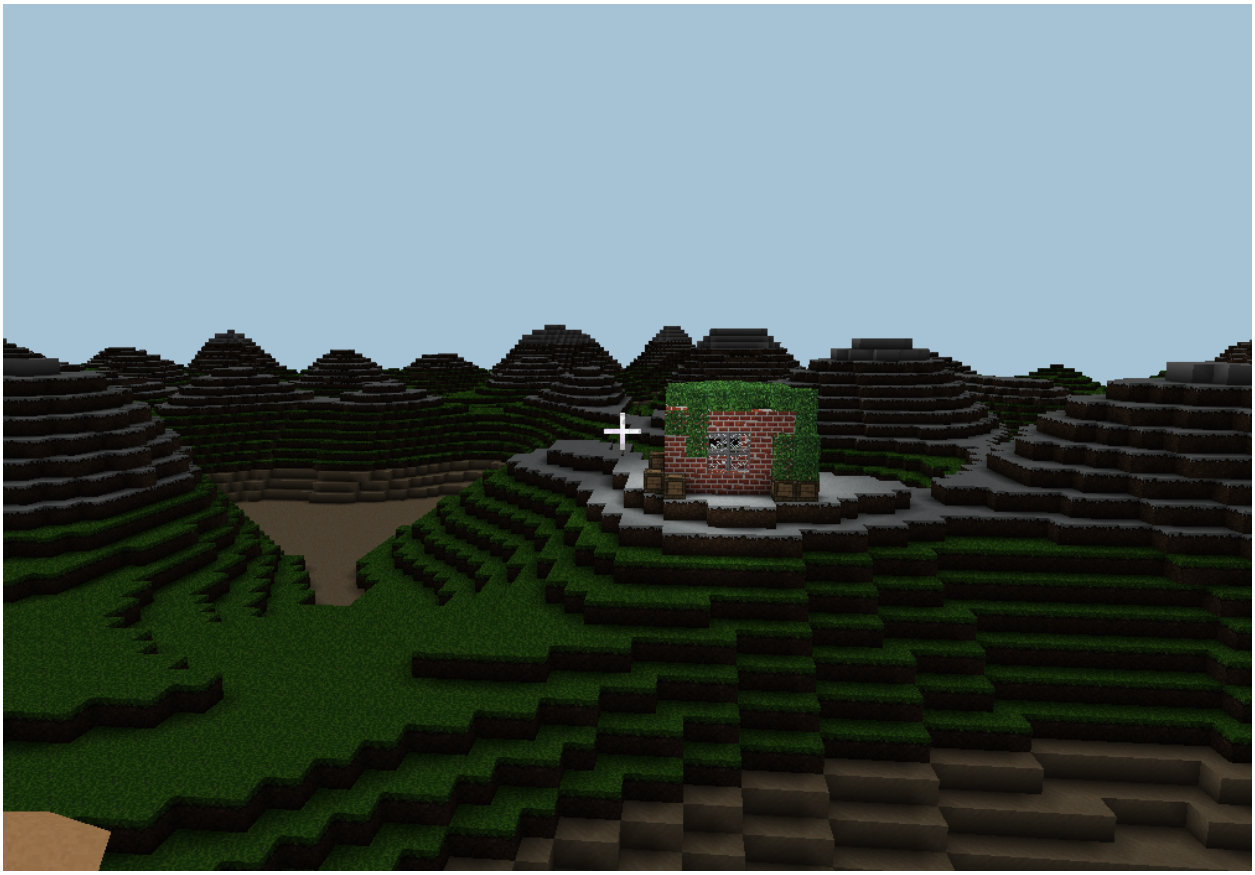
[5] Wilder, Michael W., *An Investigation in Implementing a C++ Voxel Game Engine with Destructible Terrain* (2015). Honors Research Projects. Paper 217.

[6] E. Arneback, A. Lunden, et al *Bloxel - Developing a voxel game engine in Java using OpenGL.* GUPEA 2015.

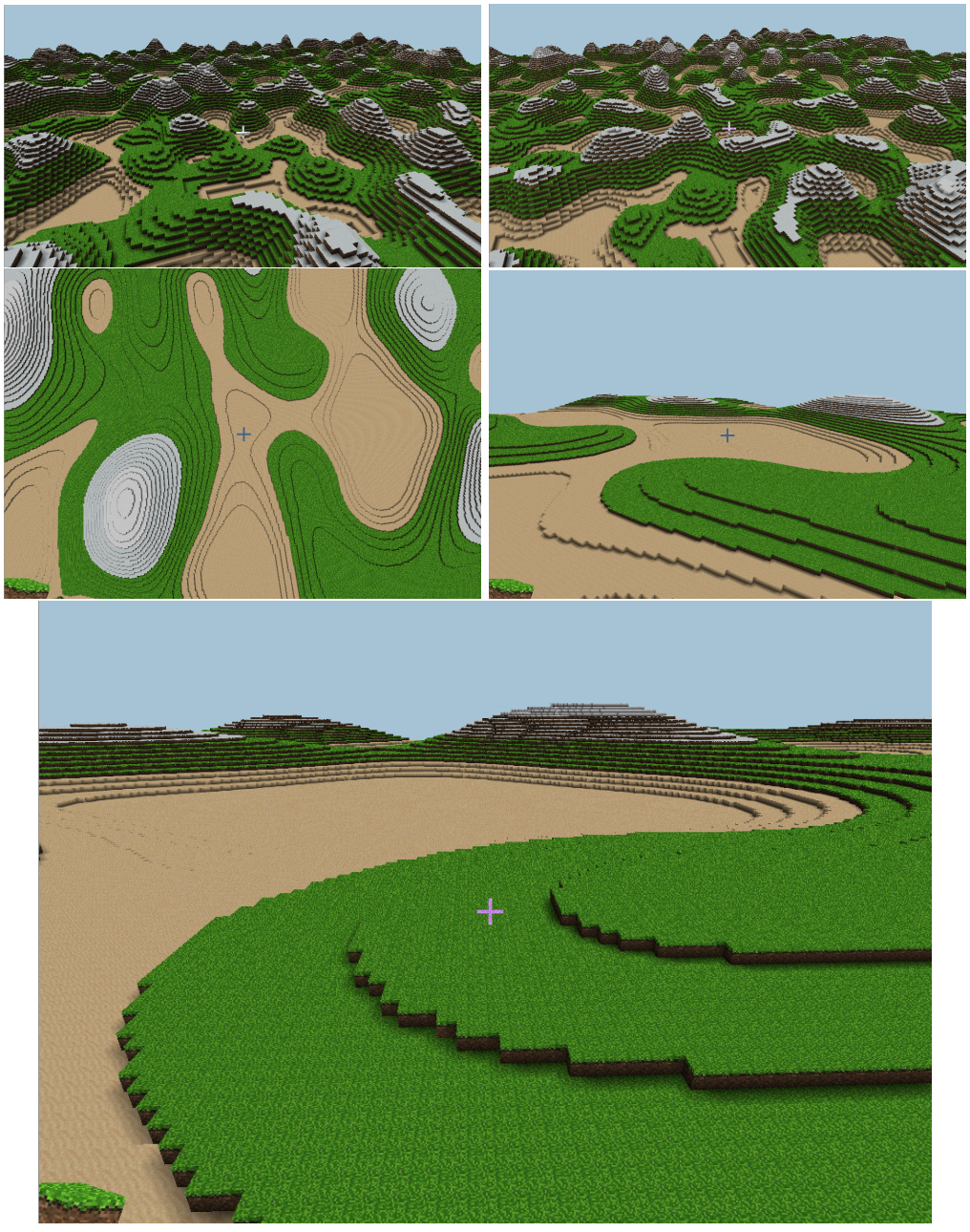
Chapter 5

Appendix

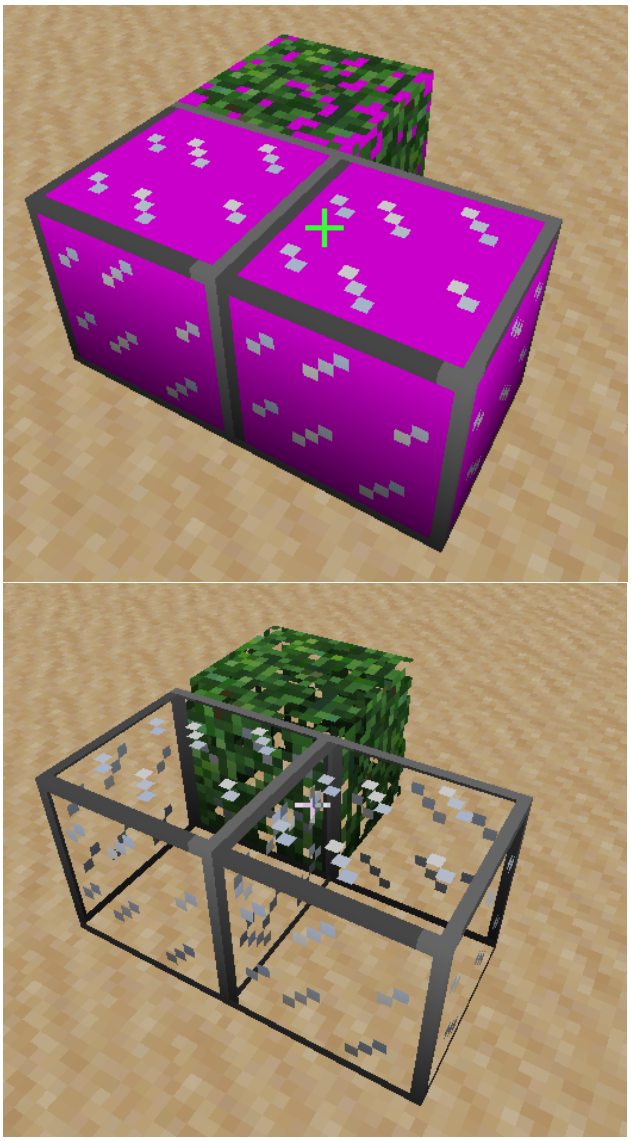
.1 Appendix A - World



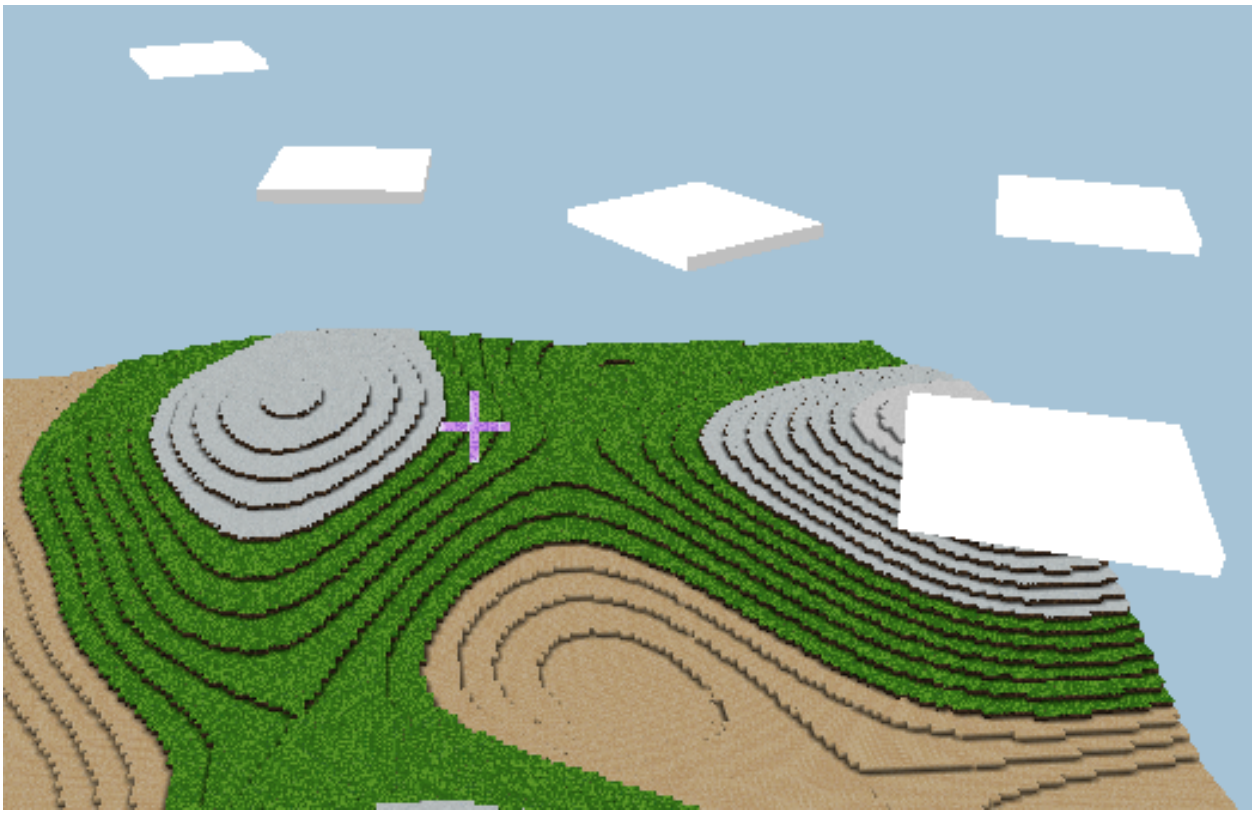
.2 Appendix B - Perlin



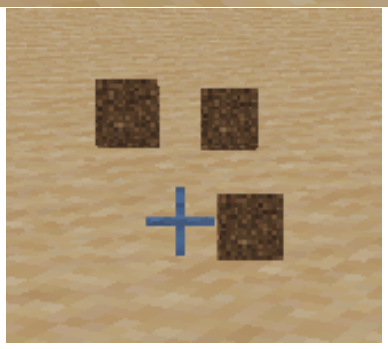
.3 Appendix C - Transparency Toggle



.4 Appendix D - Clouds

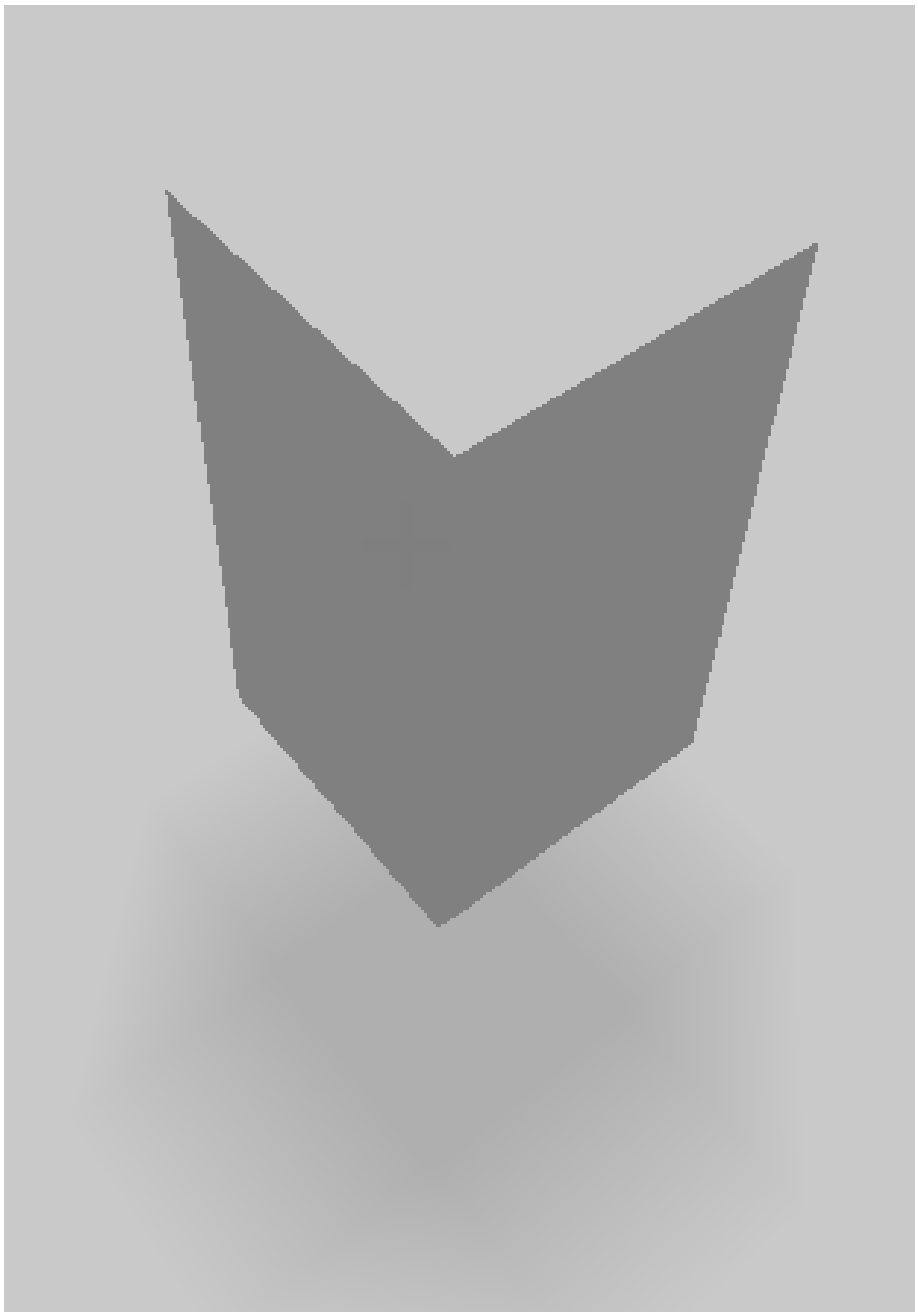


.5 Appendix E - Particles



.6 Appendix F - Light Emitting

.7 Appendix G - Height Shadows



.8 Appendix H - Ambient Occlusion