Abound

CSCE A470 Capstone

Anthony van Weel, Antonio Pennell

5/2/2024

Acknowledgements

We would like to acknowledge Dr. Kenrick Mock for mentoring us throughout making Abound, Andrew Tyler for making the font used in the game, Patrick de Arteaga for making the music, and our family and friends for play testing.

Abstract

Abound aims to be a 2D sandbox role-playing game inspired by the niche game Starbound. What you see are the beginnings of a years-long endeavor to learn from the successes and failures of Starbound and build a game that achieves the goals it ultimately failed to meet. For the course of this project however, Abound is a simple 2D sandbox game created with a custom game engine.

Table of Contents

Introduction 5
Requirements 6
- Implemented 6
- Not Implemented 7

Design 8
- Architecture 8
- Rendering Pipeline 8
  - Shaders 8
  - Sprite Rendering 9
  - Text Rendering 9
- Context Handling 10
- Input Handling 10
- World 11
- World Generation 11
- Physics 14

Software Development Process 15
- Planning 15
- Testing 15

Conclusion 17
- Results 17
- Discussion 17

References 18
User Manual 19
- User Interface 19
  - Main Menu 19
  - Pause Menu 20
  - Settings Menu 20
  - Audio Settings 21
  - Hotbar 21
- Gameplay 22
  - Movement 22
  - World Interaction 23
    - Pickaxe 23
    - Blocks 23

Heuristic Evaluation 24
Code Contributions 26

Introduction

Abound is a 2D side scrolling sandbox game where the player controls a player character and can create and destroy the procedurally generated terrain in order to create their own custom structures and landscape. The goal of this project is to create a 2D sandbox game with a custom game engine. As such it is built upon Open Graphics Library (OpenGL), Simple DirectMedia Layer 2 (SDL2), FastNoise2, and FreeType. These libraries combined provide the starting point for creating a custom game engine.

Requirements

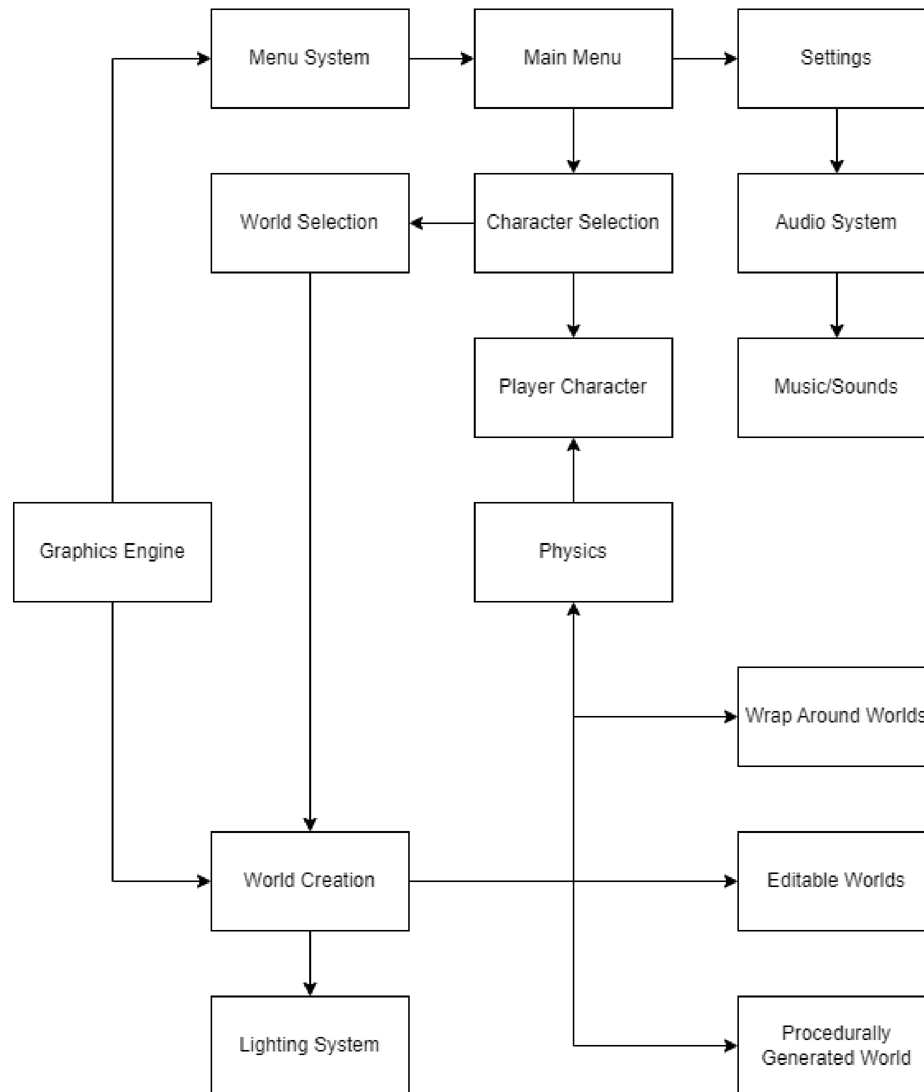

Figure 1. This graph outlines the requirements and in what order they should get done in. This also served to help keep track of all progress made.

Initially, Abound's requirements were split into two sections, a required and a stretch goal section. However, due to the lack of time the entire stretch goal section was dropped with the exception of implementing parallax.

Implemented

- A graphics engine was implemented to allow rendering images to take place.
- The world, perhaps one of the most vital features of the game, due to the entirety of the game play taking place here.

- Wrap around worlds, while not common in sandbox games this was one feature we wanted to add to make the world feel more like something you could fully explore.
- Procedural generation, the initial plan was for each world to be different, but due to lack of development time, we decided to hard code the world seed so only one world is generated. The reason for this is that sometimes the player would spawn in rock which caused undesirable movement at the start.
- Fully destructible terrain was implemented to allow the player to edit the world using the blocks in their hotbar.
- Physics allows the player to traverse and be limited by the terrain.
- Player character makes it so that the player must take into consideration the player size when building, leading to the development of more consistent structures.
- Menus were implemented as they are present in nearly every video game created allowing us to implement a credits screen, along with options to customize the player's experience.
- An audio system was implemented to make the game more immersive by playing ambient music.

Not Implemented

- The lighting system, while initially planned and prototyped, was determined to be too time consuming to implement before the deadline.
- Character selection was unable to be implemented due to the menus taking longer than expected to implement.
- World selection was unable to be implemented due to the menus taking longer than expected to implement.

Design

Architecture

The architecture for Abound uses a monolithic architecture commonly found in games. There is a central game class that has the audio system, rendering engine, and input handler singletons declared within it. We decided early on that each object would adhere to the principles of single object responsibility so each object would only be responsible for things related to itself. The only exception to this is the game class which mostly acts as a control center for all of the components of the game. Towards the end of the project we did start to run into issues with this architecture. Namely, passing information from various parts of the game to others started to become unwieldy, causing us to oftentimes have to go and pass in references to other classes purely to ensure a single function of the class has access to the data it needs.

Rendering Pipeline

To start rendering using OpenGL, you must first create a context. This contains the frame buffer that everything renders to and is eventually displayed on screen, and it contains everything OpenGL uses. From here, we enabled blending and transparency to allow transparent textures to be properly displayed. After that the vertex array buffer is created. The vertex array buffer has two important pieces of information. It contains the render target positions, and the texture sample positions. Each of these coordinates exist within the range of 0 to 1 due to how GPUs handle UV mapping with U and V denoting the axis on which the textures are sampled. So if you wanted to sample only the top left 32x32 square of a 64x64 texture, you would use the UV coordinates 0 and 0.5 to denote the start of the texture and the middle of the texture. Once the vertex buffer is created, it must be bound, enabled, and then have a vertex attribute array created so arguments can be passed to shaders. From here, you can create the textures and shaders. Once the textures and shaders are created, you can start rendering things to the screen. If you wish to learn more about the OpenGL rendering pipeline, we recommend visiting the OpenGL wiki at https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

Shaders

From the beginning we wanted to include shaders as part of the rendering engine. This allows for tricks to be used that make rendering objects more efficient, but also allow for effects like shifting colors of a sprite, or even distorting it. Unlike your traditional program, shader programs are compiled at runtime. To create a shader you need to tell OpenGL what type of shader you want to make, then you must allow it to assign a shader ID. This ID is what you pass to the GPU whenever you want to render an object so it knows which set of shaders to use to do so. Once the shader is compiled, you must attach the ID to the shader which sends it to the GPU, and then link the shader so OpenGL knows that that shader ID is taken.

Sprite Rendering

Once shaders are initialized, and textures are loaded you can begin drawing things to the screen. To start rendering a texture, first bind the shader you wish to use. Once you do that you will need to initialize a 4x4 identity matrix. Then you take the position you want to render the texture at and you translate the matrix by it, but since we lack a 3rd dimension in our game, we ensure that the z component of the translation is 0. We then go and scale the matrix by the scale of our texture so it renders as the correct size. After that we set the projection. The projection translates world space to screen space so we don't need to worry about moving our world around the camera whenever the player moves. Once all of that is done, we set the attributes of the shader to the model, projection, and color so the shader can use them. Then we activate the sprite and bind it before finally binding the vertex array and drawing it.

Text Rendering

When it comes to text rendering we initially wanted to implement the entire submodule from scratch, underestimating the complexity of the task. Ultimately, we ended up utilizing the freetype library, which when combined with openGL, handles the complex vector atlasing and to some extent the shaders for us. With the freetype library the primary consideration became managing the font on a glyph level.

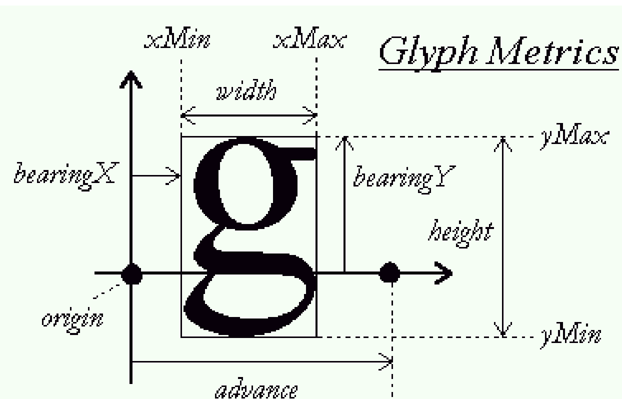

Figure 2. Glyph attributes denoting different properties and how they effect glyph offset [1].

On font initialization we load each glyph and the relevant metrics such as advance, bearing, and height into a text character struct that is then stored within an array indexed by the character it represents.

The bearing and advance metrics are the most important to pay attention to within our use case. Bearing X tells us how far the right our glyph is in relation to the origin point on the left which could be considered where the cursor would begin. Bearing Y similar to bearing X informs us of the height of the glyph in relation to the horizontal midline from our origin, this is what ensures that Y and y for example are indeed different heights and that the hook for the lower case y dips below where the uppercase would stop.

The advance attribute informs us of how far to the right our cursor needs to move in pixels to ensure even spacing between characters. This is also especially important for determining the pixel width of a provided string to inform the X offset of the textures ensuring that the string is drawn with the intended alignment in reference to its background. Upon each call to the draw text function a string is passed that is iterated through, using each character to index the relevant glyph metrics which are used in conjunction with a shader for drawing onto the screen.

Context Handling

When beginning to implement the menus and considering the inventory a problem presented itself we had not initially considered, context handling. How do we ensure that during the game buttons behave differently? For example, pressing WASD should move the character in normal circumstances, but do nothing in the pause or the future inventory menu. Initially we simply had buttons trigger different contexts, through the assignment of a context enum.

```
if (clickInObject(UIElements["startButton"])) {
      context = GAME;
}
```

The above code snippet is an example of this ‘naive’ implementation, as when the topic of submenus was discussed it became clear that a single variable was not sufficient as tracking of previous context states was necessary. Ultimately our ‘final’ design turned our single variable into a stack, this allowed us to, on press of buttons that should impact the context, push the new state and simply pop it later to return to the previous menu screen or back into the game itself in the case of the pause menu. The below snippet exemplifies the difference.

```
if (contextStack.top() == SETTINGS) {
        if (clickInObject(UIElements["backButton"])) {
            contextStack.pop();
        }
    }
```

Input Handling

Our input handling utilizes an event API provided by SDL to read in input events and pass them to our input handler which maps specific keys to functions related to those keys. We utilized the ‘unordered_list’ hashmap to, on startup, map SDL keycodes for the keys we wanted to use to the appropriate functions for said keys; the snippet below shows this mapping.

```
    keyMap[SDLK_ESCAPE] = &InputHandler::keyboard_escPressed;
    keyMap[-SDLK_ESCAPE] = &InputHandler::keyboard_escReleased;
```

Upon the indexing of the keyboard and mouse button maps, the functions would be called allowing for fewer lines of code and a simplified control structure at the cost of increased time spent debugging issues which will be expanded upon in the lessons learned section.

World

The world is procedurally generated and split into 16x16x2 chunks. Each chunk consists of a 16x16 foreground layer and a 16x16 background layer. In order to render the world, a check is done starting from where the player is offset by half the width of the screen in chunks. The world checks all the chunks that should be displayed on screen and sees if they are fully generated. If the chunk is not fully generated it generates the chunk. Furthermore, each chunk is rendered front to back. So if there is not a block present in the foreground, it will render the block in the background, and if there is no block in the background then it won't render anything.

The player is considered to be a property of the world that everything else is oriented around. It is also the only object in the world that is affected by collision. The player also serves the unique purpose of being the object that the camera follows at all times.

World Generation

The world generation runs on the main thread and is responsible for lag spikes that are encountered while playing. The world generation generates one chunk at a time and does so in five stages.

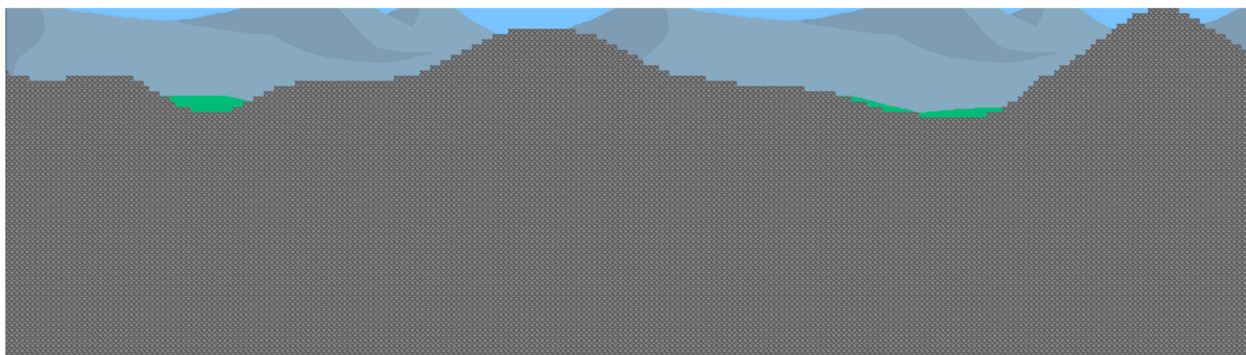

Figure 3. Phase 1 of world generation

The first stage of world generation is generating the heightmap for the chunk. To do this a 2D perlin noise is sampled twice, once from the current position in the world, and a second time from the current position in the world offset by the width of the world. This is done 16 times per chunk in order to generate the height. From here, the ground level is displaced by the values in order to generate a smooth wrap around world. Everything below the displaced value is turned to stone, and everything above it is turned to air.

```
float World::genFloat(float x, float y, float xPeriod, float yPeriod,
float scale) {
    return (
        (x / (worldSize * chunkSize)) * FBMNoise->GenSingle2D(x / xPeriod,
y / yPeriod, seed) +
        (1 - x / (worldSize * chunkSize)) * FBMNoise->GenSingle2D((x +
(worldSize * chunkSize)) / xPeriod, y / yPeriod, seed)
        ) * scale;
}
```

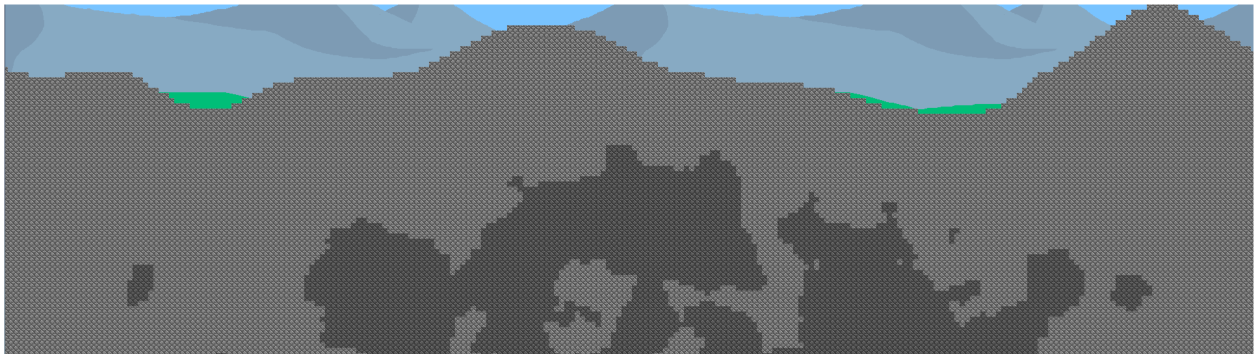

Figure 4. Stage 2

The second stage of world generation is generating caves. To do this 2D perlin noise is sampled for every single tile in the chunk, and then has turbulence applied. This serves to distort the noise so it does not form smooth continuous caves. After this, a gradient from 0 to 1 is multiplied by the caves with 0 being at the surface of the world, and 1 at the bottom of the surface layer to ensure that no caves spawn at the surface of the world. This results in the caves gradually fading in the deeper down you go.

```
float World::turbulence(float x, float y, float size) {
    // Sample 3D Noise using distorted coordinates
    float value = 0.0;
    float initialSize = size;

    while (size >= 1) {
        value += genFloat(x, y, size, size, size);
        size /= 2.0;
    }

    return value / initialSize;
}
```

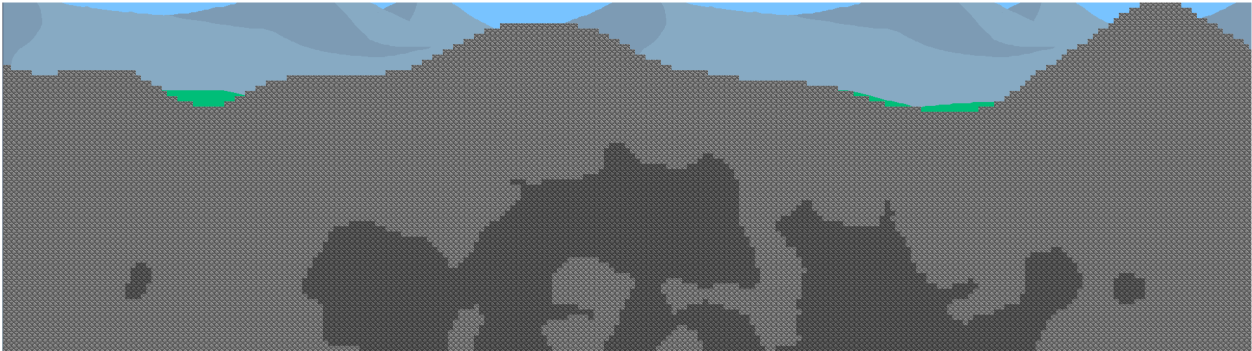

Figure 5. Phase 3 of world generation where cellular automata is applied

Phases 3 and 4 of world generation having cellular automata applied to them. Cellular automata works by having each tile check its neighbors. If a tile has six or more air neighbors it will turn into air, or if it has six or more air neighbors it will turn into air. The drawback of this, is that for each chunk that cellular automata is applied all surrounding chunks have to be generated to the previous phase. This results in each chunk causing their neighbors to be partially generated at runtime.

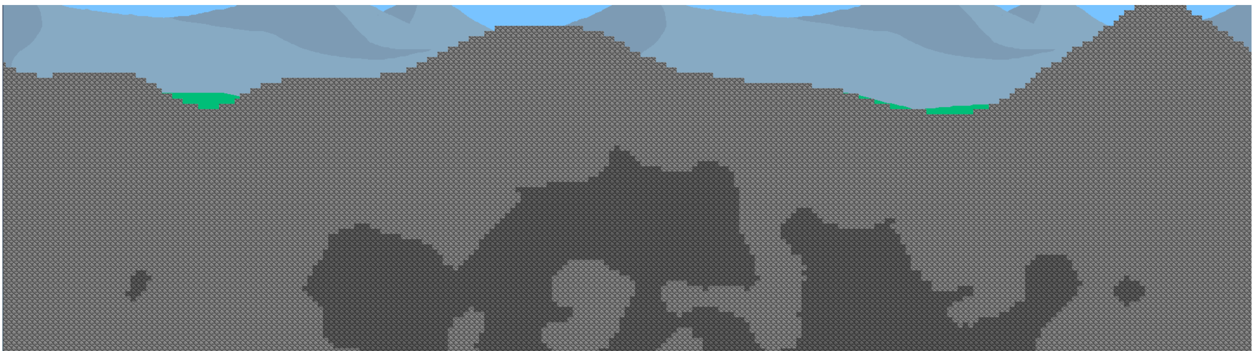

Figure 6. Phase 4 of world generation where cellular automata is applied.

Figure 7. Phase 5 of world generation

The final stage of world generation is generating the surface layer of dirt and grass. The first thing that happens is all excess stone generated by the cellular automata is trimmed away so the surface returns to its original height. Next, the surface is sampled once again similar to how it is in the first phase, except it is sampled with a slight offset. This results in the displacement being slightly different from the surface. Once the sampled values are obtained, they are used to generate the dirt by having everything above that value minus an offset for the dirt depth be turned to dirt as long as it is stone. Then, once that is done the entire surface layer is turned into grass.

Physics

The physics in the game is implemented using axis aligned bounding boxes (AABB) collision. This style of collision is very common due to its simplicity. This works by checking if two boxes overlap each other and then displacing one of the boxes for the difference of the displacement.

The only issue with this is what happens when you have thousands of tiles the player can potentially collide against? The solution for this problem is to not check against all the tiles. We know for a fact that the player can only collide with the blocks immediately surrounding the player, so those are the only blocks that are checked against. Due to a physics bug, we don't check the layer of blocks directly at the head. As otherwise the player gets pushed into the ground when running into head height obstacles. Instead we check the blocks immediately surrounding the top of the head against the player which results in no clipping through blocks.

Figure 8. The player's collision boxes and the tiles it is being checked against.

Software Development Process

Planning

We used a combination of waterfall and iterative development for planning our project. The diagram in the Requirements section served as the basis for the general order in which features would be developed and implemented, however, there were aspects not initially considered identified during development that were required to continue(eg. The need for a context system). In the cases of unforeseen requirements and those which required more artistic considerations, such as UI and Rendering, we transitioned to iterative development researching, designing, and implementing as necessary.

Testing

To test the game we employed quality assurance testing with the aid of family and friends. This method of testing was chosen over test driven development as the majority of the features implemented in the game are purely visual or rely on user input that is not easily replicated without the use of external programs connected to a debugger.

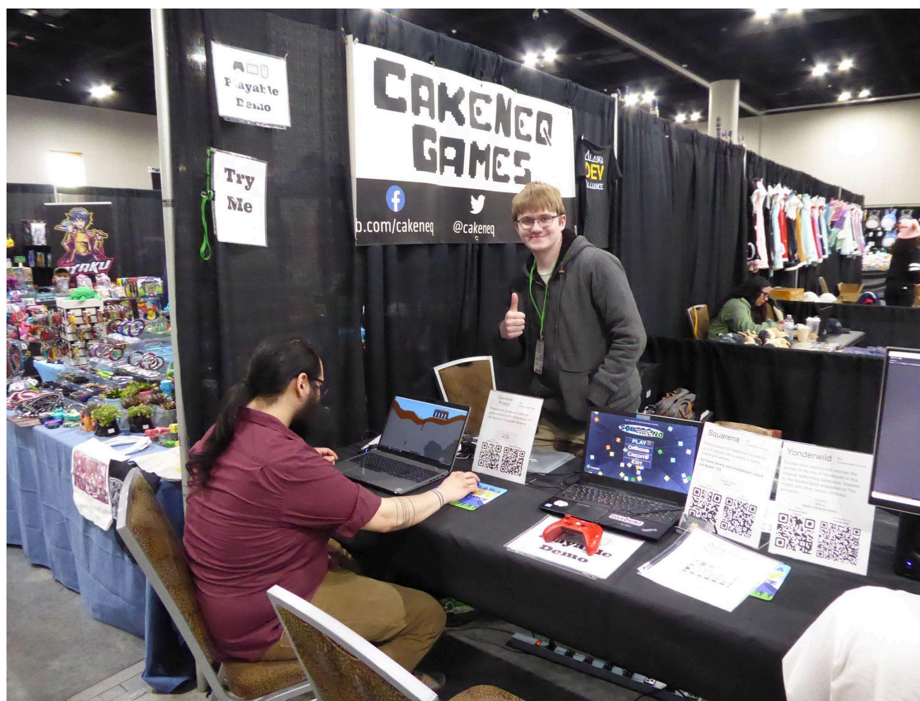

Figure 9. Matt playtesting Abound at Arctic Comic Con while Anthony van Weel poses for the camera [2].

A public playtest also took place at Arctic Comic Con as part of the Alaska Developer's Alliance Anchorage Game Creator's booth sponsored by Arctic Shield. This proved to be insightful as we were able to discover additional bugs that arose from players trying to break the

game. In addition, it allowed us to observe how people from a wide variety of backgrounds and age groups approached the game and its controls without being given instructions. Here we found that people who were unused to games defaulted to using the arrow keys. In addition, we found that after adding arrow key support most people were able to pick up the game without needing instructions on how to play it. The greatest cause of confusion was being unable to click the hotbar in order to select a different item.

Bug #	Bug	Solution
1	Middle clicking with the mouse crashes the game.	This was fixed by adding a catch for unexpected user input.
2	Right shift makes the pause menu repeatedly flash onto the screen.	Cause of the bug is unknown.
3	Left control makes the player repeatedly jump.	Cause of the bug is unknown.
4	9 on the numpad makes the player move left.	Cause of the bug is unknown.
5	Scrolling down on the hotbar using your mouse does not work.	Added bidirectional mouse scrolling for changing hotbar items.
6	Player can fly by pressing the up arrow key.	Added ground check when jumping with this key.
7	Fullscreen does not have the hotbar at the correct position on the screen.	Caused by Windows 10 zoom level, attempted a fix, but it failed. Since the cause of the bug was not due to the game the temporary fix is to change the zoom settings on Windows 10.
8	Colliding with a block from the bottom right corner just right will send you flying through the wall.	This bug is currently not fixed due to difficulty in reproducing, and lack of time.

Figure 10. A table indicating different bugs we found during development and their solutions if we found one.

Conclusion

Results

Overall, we hit 12 out of the 15 goals planned. The only goals we didn't hit were the player selection screen, world selection screen, and lighting. Looking back, we underestimated just how much work creating a custom game engine from scratch would take, specifically when it came to rendering. If given another month, it is possible that we would be able to implement the lighting, and world menu, but both of those would require reworking sections of the game. Many of the design decisions taken were taken in the interest of saving time over ensuring the engine would be fast, modular, and expandable. As such we were able to build the game engine and game within three months.

Discussion

Although individual components were planned for implementation our initial planning was surface level (ie. we planned what would be implemented not necessarily how for all required items.). This resulted in underestimating the time needed to research, design, and implement some components slowing down the overall progress. As an additional side effect more focus was allocated toward implementing a working function over an efficient one in some cases (eg. world rendering).

When developing components we each primarily worked on our own components with the exception of when a bug came up that we were struggling to resolve. This allowed us to avoid slowdowns caused by differing implementations of the same component, but was also a tradeoff that limited our ability to quickly assist each other as we did not always understand why or how we were implementing each other's components.

The current implementation of the input system is non-dynamic and does not lend itself to easy reconfiguration for user customization. A solution was thought up that involved implementing a more robust event manager class, however it was too close to the deadline to be implemented.

References

1) de Vries, J. (n.d.). Text rendering. LearnOpenGL. https://learnopengl.com/In-Practice/Text-Rendering

2) Arctic Comic Con picture taken by Wyatt White

User Manual

User Interface

Main Menu

Figure 11. Main Menu screen

Upon launching the game you will be greeted with the main menu, simply left clicking on any of the buttons will interact with them. The start button will spawn you in the world, please see the gameplay section for more information. The options button will take you to a settings menu where you will be able to adjust audio and video settings, currently only basic audio settings are implemented, the video button settings does not have any behavior. The credits button will take you to a credits screen where you can see the authors and contributors. Exit will close the game.

Pause Menu

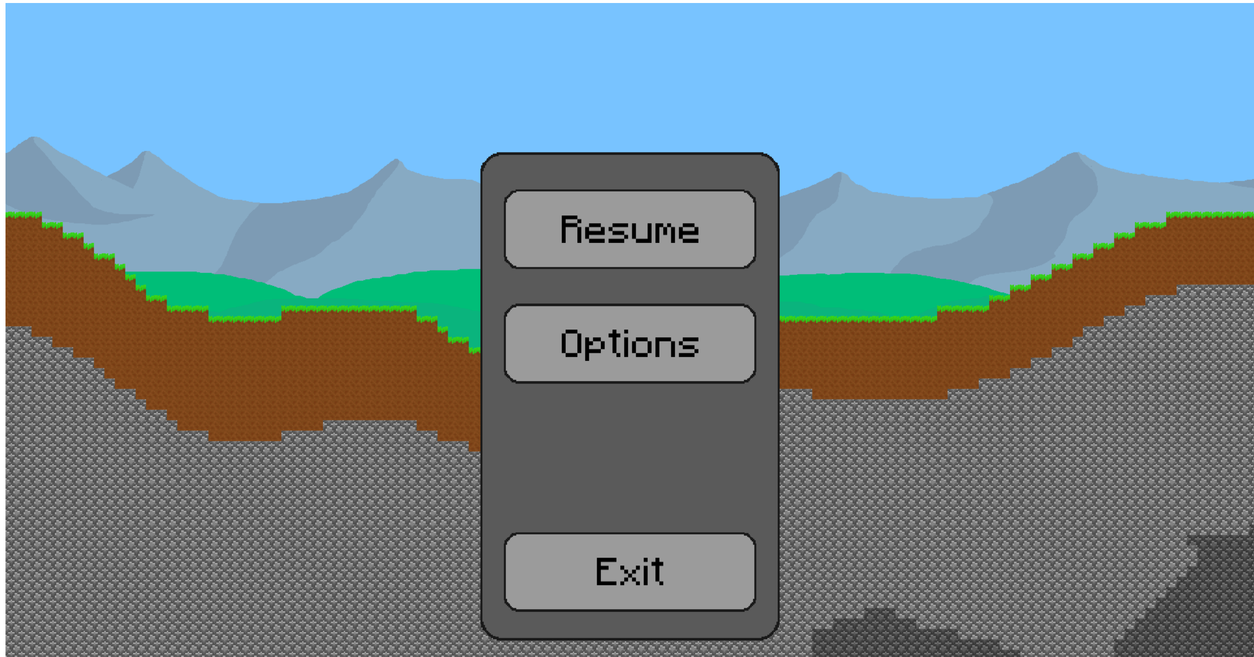

Figure 12. Pause Menu screen

The pause menu can be opened during gameplay by pressing the ESC button on your keyboard, from there you can resume gameplay, open the options menu and exit to return to the main menu.

Settings Menu

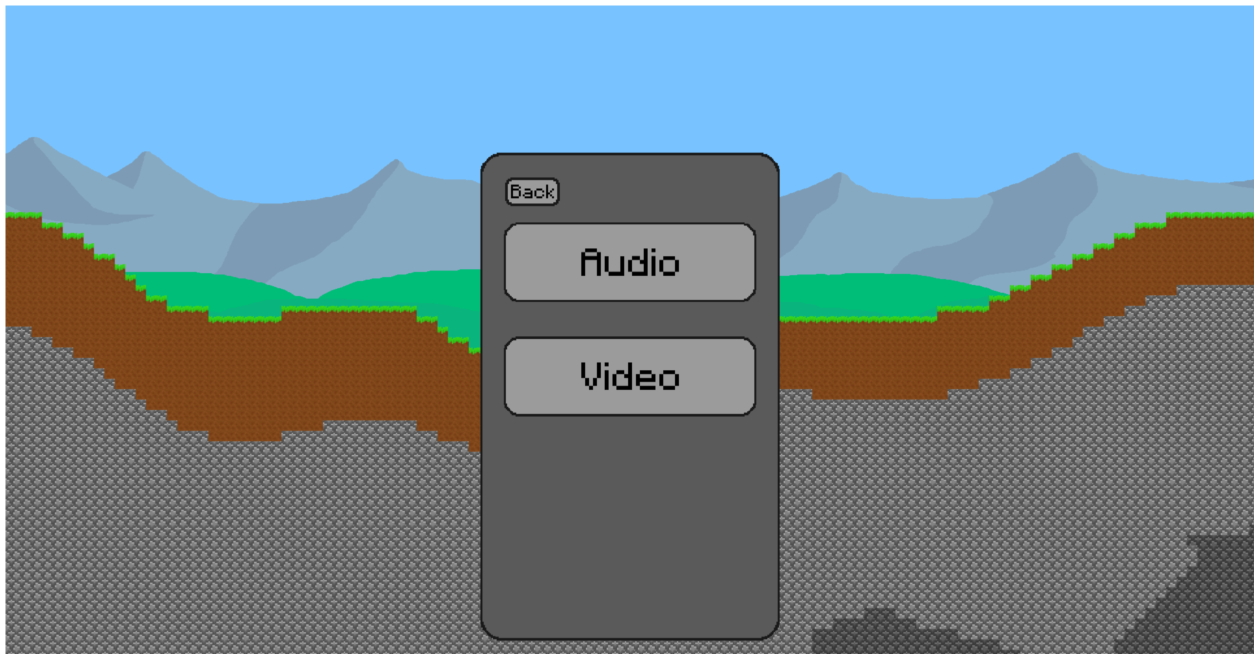

Figure 13. Settings menu

There are two settings categories within the settings menu, video(not implemented) and audio. Left clicking with the mouse on either buttons will bring you to the respective settings options.

Audio Settings

Figure 14. Audio settings menu

There are two audio settings available for adjustment, music volume and sound effects (SFX) volume. Left clicking right the mouse on either of the left arrows will decrease their respective volumes and vice versa. The back menu in the top left corner can be used to return to the previous menu context.

Hotbar

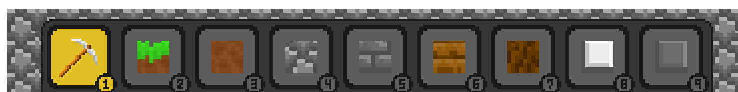

Figure 15. The player's hotbar

The hotbar is available during gameplay and determines the right and left mouse click actions within the world. The current selected item can be changed with the scroll wheel, or via pressing the corresponding number on the number bar.

Gameplay

Movement

Figure 16. An in-game screenshot

The player character is controlled via the A, S, Left Arrow, Right Arrow, and Spacebar. A, S as well as Left Arrow and Right Arrow move the character left and right respectively while the spacebar is jump.

World Interaction

World interaction is limited to the placement and destruction of blocks.

Figure 17. An in-game screenshot demonstrating the player having interacted with the world.

Depending on the selection within the hotbar various actions can be performed:

Pickaxe

With the pickaxe (First item on the left) background and foreground block can be removed

Blocks

Any of the items from 2-9 on the hotbar are blocks. Blocks can be placed in the foreground with a left click while having the item selected and the background with a right click.

A yellow selector appears over the grid the cursor is hovering, this is where a block will be placed or destroyed.

Heuristic Evaluation

Severity (H M L Q)	Filename	Location (line #)	Description of Defect	Solution/Justification
M	World.cpp	182-199	store in data structure like list or map, easier to manage and reference	This makes world gen take much longer, string lookups are inefficient.
H	World.h	full file	No deconstructors	None yet, troubleshooting issues with crashes from freeing the memory.
L/funny?	World.h	87	void generateChunk(Chunk* **chonk**, int x, int y, int phase); but in World.cpp it's chunk	Forgot to update header file when renaming variables
L	World.cpp	97, 106...	Same variables being used for different entity comparisons - local function name vs class name	Will rename variable to be more clear
M	World.h	28	the seed variable is not initialized	Added initialization to constructor.
L	World.cpp	Full File	world.cpp is very long, consider breaking into mutliple files	Didn't have enough time to break the world generation into its own class
L	World.h	22, 23, 37-39, 49, 59	Magic numbers	World Generation Parameters feel a bit hand wavy no matter what
L	World.cpp	full file	No function documentation.	None yet, time contraints have forced a focus on implementation.
M	World.cpp	6	updateActiveChunks() clears all rendered chunks- possible to only clear offscreen ones?	Deffered due to complexity.
L	World.cpp	253-259	Formatting is inconsistent with other lines of code	Issue is only present in google docs.

			(e.g. newlines in args)	
L	World.cpp	149	scaleBetween() return expression really long, unsure what's being returned at a glance; could assign to a new variable & then	Deprecated
L	World.cpp	multiple	add coords class with x & y attributes	Deffered due to refactoring required to implement.
L	.h	46,48	S vs s in Surface	Forgot to change
L	.cpp	182-198	consider all caps for named constants	Unchanged.
L	.cpp	339-354	magic number 6; simplify code somehow?	Forgot to change
	World.h	74-75	Using a default constructor to call another constructor in the same class, possible initialization error with World(int seed)	Intended behavior
L	World.h	27, 77	C++ has a smart pointer class that you can use instead of raw pointers -- can prevent memory leaks	Researching appropriate use-cases for our implementations.
L	World.h	multiple (29, 30 for ex.)	Might be better to use constexpr instead of int/float for some variables	Some of these variables may seem like they could, but aren't implemented as consts due to allowing screen size to be readjusted

Figure 18. Heuristic Evaluations

Code Contributions

Audio System: Antonio
Chunk: Anthony
Game Class: 70% Antonio 30% Anthony
Input Handler: 90% Antonio 10% Anthony
Tile : Anthony
World: Anthony
BoundingBox : Anthony
Character: Anthony
Player: Anthony
.frag & .vert files 90 Anthony 10 Antonio
Parallax: Anthony
Rendering Engine: 90% Anthony 10% Antonio
Shaders: Anthony
Sprite: Anthony
UIElement: Antonio