Introduction

In this assignment you will build your own version of one of the most popular indie games ever: Minecraft. If you are not familiar with Minecraft, you can check out the classic version for free at http://www.minecraft.net/classic/play (requires Java). The game is based on a practically infinite block world in which players may mine (remove) almost any block. Players may also place mined blocks back into the world at any unoccupied location. During the daytime the world is peaceful, but at night mobs of enemies appear and attack the player.

The reason we are studying Minecraft is not because of its popularity, but instead because its underlying engine can be implemented in a matter of weeks. By the end of the assignment, you will have a fully playable Minecraft clone, complete with mining blocks, an infinite world, and simple enemies.

Demos of each checkpoint are available by running cs1972 demo minecraft\{1,2,3\}.

Support Files

Since you’re continuing to work with the same engine, there is no support code for this project. However, we have provided some assets that may be useful, which can be found at /course/cs1972/asgn/minecraft.

There, you’ll find a a 256x256 image named terrain.png. This is a texture atlas: a grid of 16x16 textures. You will be using this image to texture all the blocks in your game world. Additionally, there is a texture atlas to skin a pig, and 6 textures to make a minecraft-themed skybox. Feel free to use all or none of these assets: there are plenty of resources online with higher-res textures, different skyboxes, and more interesting NPC skins.

Week 1 - Due Feb. 16

Minecraft1 is all about setting up your basic world representation. The world will be divided into 32x32x32 chunks, each of which will be divided into 1x1x1 blocks. At the end of this week, you will have a small procedurally generated world.

Design Check

- How will you represent a block? What about a chunk?
- Write up an interface for your Voxel “manager” class.
- What will your procedural generation algorithm do? Is it engine-side or game-side?
Voxel Engine Requirements

The majority of your work during these weeks will go towards your “voxel” engine. This collision engine will support the full range of necessary features for a block based world. However, this week, you will just set up the basic world representation.

Primary Requirements

- Chunk-based world representation (implemented as a “manager”)
- Block defined as a 1x1x1 cube with (minimally) the following characteristics:
  - Transparency - should the engine render block faces adjacent to this block?
  - Passability - can entities pass through this block?
- Chunk defined as a CxCxC group of blocks, where C is a single parameter determining chunk dimensions. If you prefer, you may have three parameters for XxYxZ size chunks

Secondary Requirements

- Chunks render only block faces that are adjacent to transparent blocks, with the exception of chunk boundaries, which may always be rendered

Game Requirements

Like warmup1, you don’t really have enough functionality to make a real game this week. However, you do have enough in place to procedurally generate your terrain, and this will be your main game task for this week, besides showing us that your engine requirements are working.

Primary Requirements

- Movement and view controls must be the same as warmup
- The world must have no gravity, so the player must have an additional controls to change their height
- The world must consist of a 2x1x2 grid of size C = 32 chunks

Secondary Requirements

- The terrain in the world must be procedurally generated
  - Chunk composition is algorithmically determined with the complexity of at least a \( \sin \) wave, but we strongly recommend trying Perlin/Value/Simplex noise
  - Terrain contains at least two visually unique blocks besides air/empty blocks
- There is no FPS requirement this week. This overrides the standard 30+ FPS requirement.
Week 2 - Due Feb. 23

Minecraft2 focuses on adding collisions to your “voxel” engine, but also features a number of rendering optimizations that will help you double your framerate while at the same time populating the world with 25 times as many chunks! Talk about fancy...

Design Check

- Describe the view frustum culling test.
- Describe at a high level how you will generate the vertex data for your per-chunk dynamic VBOs.
- How will you support texture atlasing for chunks? What is engine-side and what is game-side?
- In what order will you do collision detection?
- Draw and walk through a 2D sweep test (like the example from the slides).

Common Engine Requirements

Many rendering optimizations don’t apply to a specific type of game, and this week’s requirement is no exception.

Primary Requirements

- Camera or graphics object supports view frustum culling of axis-aligned bounding boxes

Voxel Engine Requirements

Unfortunately, one of our rendering optimizations only makes sense for a voxel structured world. Additionally, our collision detection depends on a block-based world.

Primary Requirements

- Per-chunk dynamic VBOs

Secondary Requirements

- Collision detection and sliding response with collided blocks for arbitrary size axis-aligned bounding boxes

Tertiary Requirements

- Texture atlas support for per-chunk VBOs (must use the texture coordinates in the VBO)
Game Requirements

With collisions implemented, your game will be a demo of the rendering and basic physics system you’ve created! The player won’t be able to interact with anything, but they will be able to explore a much larger procedurally generated world. Be sure to stress test!

Primary Requirements

- Movement and view controls must be the same as warmup (including gravity and jumping)
- The world must consist of a 10x2x10 grid of size C = 32 chunks
- Player must be vertically larger than 1x1x1 block
- The game must run at 40+ FPS on department machines: this requirement overrides the standard 30+ FPS requirement

Secondary Requirements

- Player must be larger than 1x1x1 block

Tertiary Requirements

- All terrain is textured from a single atlas

Week 3 - Due Mar. 1

Minecraft3 adds raycasting to your “voxel” engine for the purpose of interactivity. With this, the engine will be complete, and you’ll be able to add some real gameplay, and make your world nearly infinite. We’d like to remind you that even though your game may look like minecraft now, there is no reason you can’t make it a first person shooter, a racing game, or a puzzle game. Be creative!

Design Check

- Draw and walk through a 2D grid-based raycast (like the example from the slides).
- What is your criteria for determining if a chunk should be loaded or unloaded?
- When will you perform the actual loading or unloading of chunks?
- How will you implement addition and removal of blocks?
- What will your enemies be, and how will they be involved in your gameplay?

Voxel Engine Requirements

Raycasting is essential for interaction in a world, so make sure your raycasting is both correct and robust! Additionally, a world that loads itself makes for a much better play experience.
Primary Requirements

- Analytic voxel-based raycasting
- Addition and removal of block methods
  - Addition or removal of blocks updates the appropriate VBO’s

Secondary Requirements

- Automated chunk streaming
  - Chunks within a certain distance of a selected focus point are automatically loaded into the game
  - Loaded chunks outside a certain distance are unloaded from the game

Game Requirements

This week is pretty heavy on the game requirements, but most of them just make sure your final product is something to be proud of. Put some effort into it: this will hopefully be one of the coolest projects you’ve implemented at Brown!

Primary Requirements

- Addition and removal of blocks
  - Ray shot from the camera eye extending along the look vector
  - Visual highlighting of the nearest intersected face
  - Targeted block can be removed
  - Block can be added to the empty block adjacent to the targeted face

Secondary Requirements

- Continuous chunk streaming
  - Chunks within 2 chunks of the player must be in memory
  - Chunks further than this distance must be unloaded from memory (you do not need to save changes to unloaded chunks)

Tertiary Requirements

- Simplistic enemies
  - Spawn either at the beginning of the game or at intervals during play
  - Collide and interact with both the world and the player
– No complex AI required, but cannot remain stationary or jump in place

• It must be possible to restart the game without quitting the program

• The game must have at least two screens

Handing In

Hand in the entire directory tree for your project, including both your engine and game code. You must also include a README file that describes how to verify each requirement, and an INSTRUCTIONS file that describes how to play your game, as specified in the Global Requirements. To hand in, run `cs1972_handin minecraft n` from the top level directory of your project (which should be where your Qt pro file is), where `n` is the checkpoint you are handing in. Please do not hand in the build files from your project.