IN3026  
Milestone 2  
Report

Table of Contents

[Overview 1](#_Toc1)

[Mechanics & Features 2](#_Toc2)

[Building 2](#_Toc3)

[Controlling Units 2](#_Toc4)

[World Generation 2](#_Toc5)

[Persistent State 2](#_Toc6)

[Controls 2](#_Toc7)

[Milestone 2: Implementation 2](#_Toc8)

[Prefab 2](#_Toc9)

[World Generation 2](#_Toc10)

[Audio Manager 3](#_Toc11)

[Clicking Manager 3](#_Toc12)

[Persistence Manager 3](#_Toc13)

[User Interface 4](#_Toc14)

[Physics 4](#_Toc15)

[Artificial Intelligence 4](#_Toc16)

[Simplification, Refactoring and Cleanup 4](#_Toc17)

[TL;DR: What is Missing 4](#_Toc18)

[Assets Used 4](#_Toc19)

[Dependencies Used 4](#_Toc20)

# Overview

The game idea pivoted slightly from the original plan. I used the available assets as inspiration of how to alter the game as a whole. Since the model assets I was able to find used hex squares, I ended up using those as the primary base terrain for the game.

The original idea was to make a game that runs continuously, but it seems like a pretty complex because the game pacing might be all over the place. Instead I decided to go with a turn based system instead, somewhat closer to Civilisation.

The primary objective of the game is to defeat the enemies, before the enemies defeat you. As part of that you have to build your citadel out with resources gathered around you.

## Mechanics & Features

### Building

### Controlling Units

### World Generation

As part of the main menu, the user can start a new game or load a current save. A

### Persistent State

As part of the main menu, the user can start a new game or load a current save. Loading a current save presents a list of files to choose from, which will then be validated

*Overview of the project, description of gameplay objectives and mechanics*

# Controls

*Keyboard/Mouse controls*

# Milestone 2: Implementation

*Design and implementation, including screenshots, code snippets, FSMs, UML*

### Prefab

As part of milestone 2, I created a prefab manager – similarly to how many other game engines create pre-defined ready assets to be dragged into a game layer. This is my take on implementing something similar, where assets can simply be placed in the world by defining a position as well as overrides for other aspects (if desired).

The manager removes most of the complexity by handling most of the logic on the engine side. This means if a prefab is placed in a world, it will automatically be called to render using the engine definitions, as well as handling all abstraction and definitions. It also stores and provides a manner of using the defined assets.

All of my assets are defined in game/managers/prefabs.h which defines 72 building variants (including colours), multiple tiles, colour themes and decorations in a centralised and generic manner that is easy to access. It uses the engine prefab manager to bind all of these assets as prefabs, and then they can be accessed during the layer to load into memory and bind it to render and be recognised as an prefab instance in the world.

I have chosen this approach since most engines do this, and in my case, I would have a lot of repetitive code defining similar or the same assets. This allows me to define all of my assets in a generalised manner, as well as providing a clean way of removing and abstracting a lot of code and logic from the layer (since it’s already pretty long and complex as is).

### World Generation

The world generation is based on a given size and seed. The seed determines the look and feel of the map such as the height map of all the tiles. The size essentially scales that seed and provides more detail (more tiles) in between – sort of like displaying a high resolution image on a small resolution screen (low size) looses details whereas a higher resolution screen retains them, in this case infinitely scalable (as limited by processing power).

The world generation creates a quantity, based on the size, of desired points and their heights. It then looks to gradually smoothen the terrain in between those points to create believable terrain that has potential to have dips (which are filled with water), as well as spikes like hills and even mountains. Bilinear interpolation with uniform distributions are the primary maths involved, with a lot of other steps to improve the output of the noise generated. This creates a pretty believable and natural looking terrain.

[ADD IMAGE OF RELEVANT CODE]

As part of this, I did look into other forms of terrain and noise generation, such as perlin noise. I am aware of dependencies and modules that provide an out of the box working and fitting system for this. However, I decided on using a much simpler approach where I could learn more C++ and maths, and fully understand the code I am using.

[ADD IMAGE OF GENERATED WORLD]

Unfortunately, due to time constraints, I was unable to fully take it to the next level I wished to take it to by using all the tiles provided in the pack and all of the variants, as well as implementing y-height the way I wished to. Since this is a really complex aspect of the game, with an extra week or two, I believe it would be possible to create a very impressive world generator that would handle all of these aspects. This is also fully expandable to fit future needs of the game as new features are added.

### Audio Manager

TBD

The current engine audio manager was extended with functionality. This functionality relates to playing tracks simultaneously and switching between the two with a smooth crossfade. This allows the music to ramp up as required by the game. Some of the code has also been simplified and cleaned up a little bit.

Unfortunately, due to limited time, I was not able to bind all of the music tracks in a neat file for easy access like I did with the prefabs. I also did not have time to restructure the audio manager in full – I believe a lot of it can be simplified down, and cut out about half of all the lines in the file for easier maintainability.

### Clicking Manager

TBD

### Persistence Manager

TBD

As the name suggest, the persistence manager manages the persistent state of the game - i.e. saving and loading. The data structure is held in a json format which stores the seed and size of the map. This is used by the world generator to regenerate the world exactly how it was. It also stores the resources, buildings, troops and other values loaded into other managers at game initialisation. In terms of saving, the persistence manager retrieves all these values, and dumps them into a json file it creates.

In the current state, the game does not support autosaving. This could very easily be added on as the game development progresses, to store 3 files in the background, and the newest version, and automatically overwrite them when autosaving. Due to the game state as a whole, I did not implement autosaving, since the game isn’t complete, and primary aspects of the game are more important for player retention.

[ADD IMAGE OF EXAMPLE SAVE FILE]

### User Interface

TBD

### Physics

TBD

### Artificial Intelligence

TBD

### Simplification, Refactoring and Cleanup

The project has been re-factored multiple times as I went along developing it. I have laid out the project in a way that I believe all similar functionalities have been grouped together. One good example of this is the world generation, which consists of the grid, everything related to the hex as well as the world generator itself. I believe this makes it easy to maintain the code, and expand it as the game improves.

Throughout development, I also simplified and optimised minor aspects of the game. One example of this is cleanup of enums, updating data structures to more efficient ones, as well as ensuring consistency throughout the project (due to inconsistencies caused by use of two IDEs and prior exploration with CLion).

# TL;DR: What is Missing

* Autosaving
* Audio Bindings
* Tile Variety
* More Complex Terrain Generation
* Terrain Height
* Stuff

# Assets Used

*All assets are referenced in the README.md under the “Usages” heading, with details of the type, source and licence details. Some new assets were added to the table, and the “Date Accessed” column has been added.*

# Dependencies Used

Only the engine has been used with minor alterations to simplify certain aspects of code and stylise them to work better for the given game. This means that the only dependency is the AGT engine/template, including all its original dependencies.