IN3026
Milestone 2
Report

Table of Contents

Overview

The game idea pivoted slightly from the original plan. I used the available assets as inspiration of how to alter the game. 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 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 must build your citadel out with resources gathered around you.

Due to a lack of time, not all planned mechanics and features have been largely implemented. Greater detail can be found in the Implementation heading under this document.

Intended Mechanics & Features

Building

A hex tile can construct a or destroy a building using the methods created. This is the primary feature to allow the player to expand their citadel, create buildings which produce resources, or enable certain actions.

Controlling Units

A player can click and control the flow of units to other tiles. Units have a movement range as well as an attack range. Some units might be utility units rather than attack units and serve different purposes.

World Generation

The world generation is used to create a unique map based on seeds to keep the game replayable and interesting.

Persistent State

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

Controls

The intended controls for the game were primarily mouse clicking, but due to a lack of time, I've only used keys since they're easier and faster to implement while focusing on other aspects of the game.

The menu controls are W/S or UP/DOWN as well as SPACE and ENTER to select as labelled.

The game layer itself doesn't have any working controls. As part of testing while developing I have bound the "F" key to make a specific unit attack a certain location – but this is to be replaced by the mouse clicking.

All debug controls range from F1 – F24 keys. The "DEBUG" flag can be used to disable these for production. At present:

- F1 – Toggle wireframe
- F9/F10 – Crossfade music tracks between A/B

Milestone 2: Implementation

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 my assets are defined in game/managers/prefabs.h. This defines 72 building variants (including colours), multiple tiles, colour themes, units and decorations in a centralised and generic manner that is easy to access. It uses the engine prefab manager to bind all 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 a 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 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 long and complex as is).

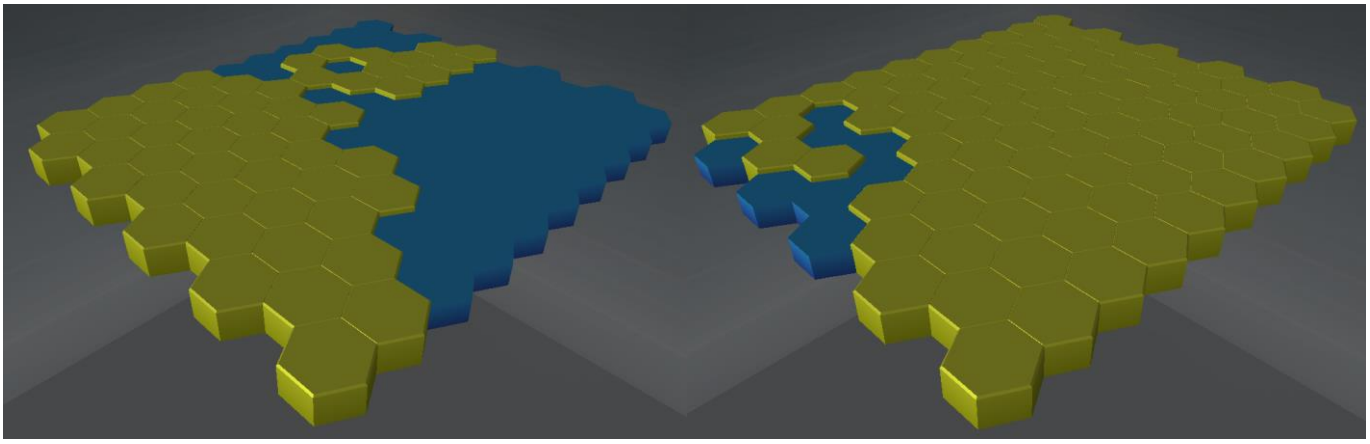
World Generation

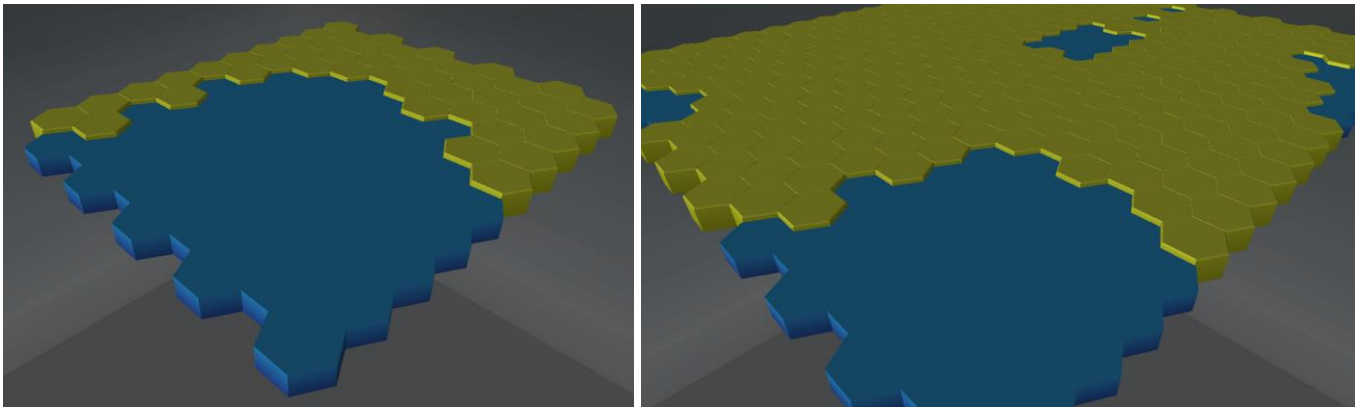
The world generation is based on a given size and seed. The seed determines the look and feel of the map such as the height map of all the tiles. With the combination of these two , even the same seed but a different size can provide a completely different map and feel to the game.

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 is the primary maths involved in calculating this height map, with a lot of other steps to improve the output of the noise generated. This creates a believable and natural looking height map.

```
float world_generator::generate_noise(float x, float y) {
    float scaled_x = x / static_cast<float>(size);
    float scaled_y = y / static_cast<float>(size);

    const uint_fast8_t control_points = (size / 4) + 1;

    uint_fast8_t grid_x = static_cast<int>(scaled_x * (control_points - 1));
    uint_fast8_t grid_y = static_cast<int>(scaled_y * (control_points - 1));

    // Calculate local coordinates within the cell
    float local_x = (scaled_x * (control_points - 1)) - grid_x;
    float local_y = (scaled_y * (control_points - 1)) - grid_y;

    // Generate stable random heights for grid points
    auto get_height = [this](uint_fast8_t x, uint_fast8_t y, uint_fast8_t control_points) -> float {
        std::uniform_real_distribution<float> dist(-0.5f, 0.5f);
        rng.seed(std::hash<std::string>{}(seed + std::to_string(x) + "," + std::to_string(y)));
        float base = dist(rng);

        // Reduce height variation at edges to prevent extreme terrain at borders
        float edge_factor = 1.0f;
        if (x == 0 || x == control_points - 1 || y == 0 || y == control_points - 1) {
            edge_factor = 0.7f;
        }

        return base * edge_factor;
    };

    // Get heights for the four corners of the current grid cell
    float h00 = get_height(grid_x, grid_y, control_points);
    float h10 = get_height(grid_x + 1, grid_y, control_points);
    float h01 = get_height(grid_x, grid_y + 1, control_points);
    float h11 = get_height(grid_x + 1, grid_y + 1, control_points);

    // Bilinear interpolation
    float x1 = h00 * (1 - local_x) + h10 * local_x;
    float x2 = h01 * (1 - local_x) + h11 * local_x;

    return x1 * (1 - local_y) + x2 * local_y;
}
```

As part of this, I did investigate 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.





(P1: “Terrain Generation” Size: 10, P2: “SEED TBD” Size: 10, P3: “Marcel” Size: 10, P4: “Marcel” Size: 20)

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 the variants, as well as implementing y-height the way I wished to. Since this is a complex aspect of the game, which would require balance, I believe with an extra week or two, it would be possible to create a very impressive world generator that would handle all these aspects. This is also fully expandable to fit future needs of the game as new features are added.

Audio Manager

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, but I see potential to rewrite the audio manager further to simplify things.

Unfortunately, due to limited time, I was not able to bind all 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.

Persistence Manager

I have attempted to create a persistence manager by storing the files as json – the schema I have created an example which can be seen below. After a bit of research, it seems to me that C++ unlike Java or C# doesn’t support serialization and deserialization in any standard modules, meaning I would need to use extra dependencies. Since these can be a hit or miss if not configured correctly to compile on other machines, and my lack of understanding of premake5 unlike CMake, I decided to pursue marks elsewhere and come back to it later if I had the time. I could have made a plaintext save file instead, but I think it would make maintaining and updating it very difficult in the future.

```
{
  "properties": {
    "version": "0.0.0",
    "timestamp": 123456
  },
  "gameConfig": {
    "seed": "1234",
    "size": 20,
    "difficulty": {
      "gather": 2,
      "enemy": 2
    }
  },
  "gameState": {
    "resources": {
      "gold": 50,
      "food": 50,
      "wood": 50,
      "stone": 50,
      "metal": 50
    },
    "tiles": [
      {
        "building": [
          {
            "coord": {
              "x": 0,
              "y": 0
            },
            "type": "foo",
            "color": "blue",
            "hp": 100,
            "owner": "foo"
          }
        ],
        "troop": []
      }
    ]
  }
}
```


With this example save file, the terrain would regenerate with the given seed and size. The gamestate would then load in the resources, any buildings and units that might exist ontop of the hex grid.

User Interface

As for User Interface, the only currently working interface is the track of resources at the top. This retrieves the data from the resource_manager and displays it as a text.

The idea was to have HUD elements for building at the bottom for the user to select with the mouse for placing, building, etc. In addition to that the user would be able to hover on buildings and units (or anything that extends the destroyable class trait) to see their health points.

Artificial Intelligence

As for AI, my idea was to use the A* Algorithm to trace a path for movement. Due to time constraints, I was unable to make any progress on this aspect. I have prepared certain parts of the hex header to contain references to its neighbours, but I didn't implement any part of the algorithm itself.

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).

Comparison with Requirements

Part 1

Intro Screen: Intro layer remains unchanged from M1.

Objects: A whole prefab system with many objects.

Skybox: Skybox remains unchanged from M1.

Audio: Parallelised background audio, no sound effects.

Part 2

Camera: Camera remains unchanged from M1.

Meshes: Mostly unchanged from M1.

Lighting: Mostly unchanged from M1.

FX: None

Part 3

Physics: None

NPC/AI: None

Gameplay: Terrain Generation, Resource Management and Building (via hex grid management due to missing UI)

TL;DR: What is Missing

Quite a few things are missing due to time constraints. That said UI and mouse interaction are the core two elements from making the game into a playable state with calling the other managers created.

- Saving
- Audio Bindings
- Tile Variety
- More Complex Terrain Generation
- Terrain Height
- Mouse Interaction
- Building UI

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.