Individual Project Proposal: The Thunder Dungeon

Thomas Young and Michael Yoshimura

Table of Contents

Project Overview

The project we propose is a game in QT and C++. The game will be a basic JRPG (Japanese RPG) that uses procedural generation.

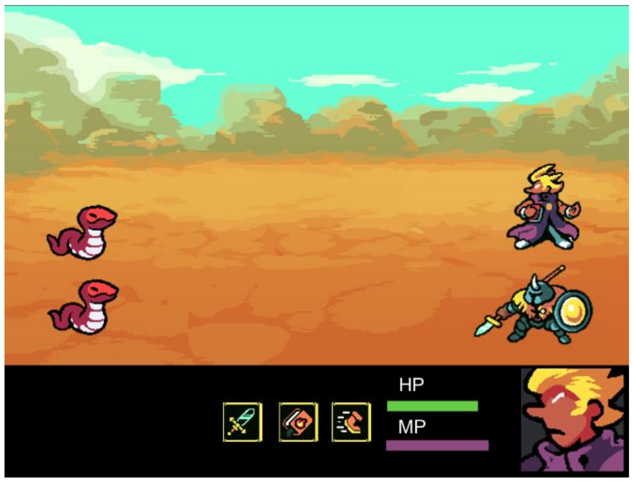
Main Features

The main mechanic of this game will be turn-based movement on a board with obstacles similar to the maze game in project 1. The secondary main mechanic will be a turn-based battle simulator in which the player uses an interface of buttons to battle with an enemy AI.

There will also be RPG-stats involved with this game, including equipment and skills to let the player make more interesting decisions in between the turn-based action.

Interface and Input

The interface will be simple interaction using the mouse to click on buttons and arrow keys to initiate movement.



An example of the type of battle simulator the game will have. (With one player)

Specific Gameplay

The player's main objective is to progress through the dungeon by defeating enemies and get stronger by acquiring better equipment. You will finally win the game after passing a set number of levels.

1. Items can be found and equipped if the player chooses
   a. When the player encounters a new item they will be shown the information about that item. At that moment, the player can choose whether to equip the new item. Whichever item they chose not to keep will be discarded.
   b. Equipped items will directly affect the player's stats and skills which are relevant to the battle portion of the game.
2. Enemies move around in the dungeon, and the player must decide whether to avoid or fight them.
   a. If an enemy moves into the player, or the player moves into the enemy, a turn-based battle will occur (the battle simulation).
   b. There will be an exit to every dungeon level, and when the exit is reached, the player enters a new, more difficult level. At the exit of every floor there is a boss/difficult enemy to fight.
   c. The game is beaten when the player reaches the exit on dungeon level 10.
3. In the battle simulation, the player faces one or more enemies with certain stats and skills.
   a. Player's will have skills that allow them to perform certain actions in battle.
   b. The enemy also has skills and will also battle each turn.
   c. If the player wins the battle (the enemy's health <= 0), the player will have the option to equip new the item(s). Then, the game goes back to the board.
   d. If the player loses (player's health <= 0), the game ends.

Additional Media

The graphics will be powered by QT, and be simple pixel art graphics. The sound and music will be minimal placeholder sounds, from a sound library.

Technologies

C++ (Familiar): this will be the object-oriented programming language to generate our game's core functionality.

Travis CI (Somewhat Familiar): this will be the continuous integration that we use in conjunction with Github and Catch2 to always test new changes.

Catch2 (Somewhat Familiar): this will be the unit testing framework we use to test our C++ code.

Github (Somewhat Familiar): this will be where we store our project. Git will be our distributed version control that will keep track of our changes and push code up to Github.

QT / QT Creator (Unfamiliar): this will be the game framework we will use. QT is fairly comprehensive, and will handle graphics, sound, and data reading and writing. We will also likely build on some of QT's other functions.

GIMP/Photoshop and Fruity Loops (Somewhat Familiar): Tools to create sound and art assets if needed

Essentials and Risks

Below is a list of elements that will be essential to the game, either in terms of mechanics or gameplay.

1. The dungeon will be generated via an algorithm. We both have some experience with this, so we don't expect this to fail, but just in case, a pre-generated world can be substituted.
2. Arrow key input and mouse input will be necessary as the player needs to interact with the game. Arrow key input will likely be easier, and if we struggle with mouse input, then we can simplify to just arrow keys and the enter button for selecting the interface.
3. 2D graphics will be essential for this game. We plan on having sprite graphics. In addition, we will need some interface as well. QT will be sufficient for all of our graphics needs. In the worst case, the game that we chose can be represented by ASCII graphics, but this is not ideal for a variety of reasons. Thus, we have already researched QT and implemented some a basic demo, so we are confident that QT will work for us.
4. The skills/enemies all will require some data to be stored in a JSON file, as their stats will be handcrafted. If this fails, these can be procedurally generated based on level. In fact, we plan to make our items entirely procedurally generated. The QT framework fully supports reading and writing JSON.
5. Storing the game state in order to leave and come back to a game will be important. This can be done by serializing the the game and storing it in a JSON file with QT.
6. Catch2 and continuous testing should not be too much of a problem, although we aren't sure how well it will work when we use a lot of QT's API.For testing simple things like entity stats and equipment the testing strategies we learn in class will work. If we encounter difficulties with testing the GUI, QT has a testing framework of its own, which we can try out.
7. Our project will not work without version control, or at least it will be very hard. We are confident that we can get Git and Github to work with our project.

External Resources

Since this is a game the main resources needed will involve either sound or graphics. Both of us have experience making and finding assets for games.

Simple 2D sprites

- We have access to free/cheap sprites for prototyping, so we aren't particularly worried about access to this resource.

Sound FX and music

- We also have access to a large sound library (~30GB) where we should be able to get most of our sound.
- In addition, we have access to music making software (fruity loops) to make simple tracks for the game.

Links to some of these resources are at the end of this proposal.

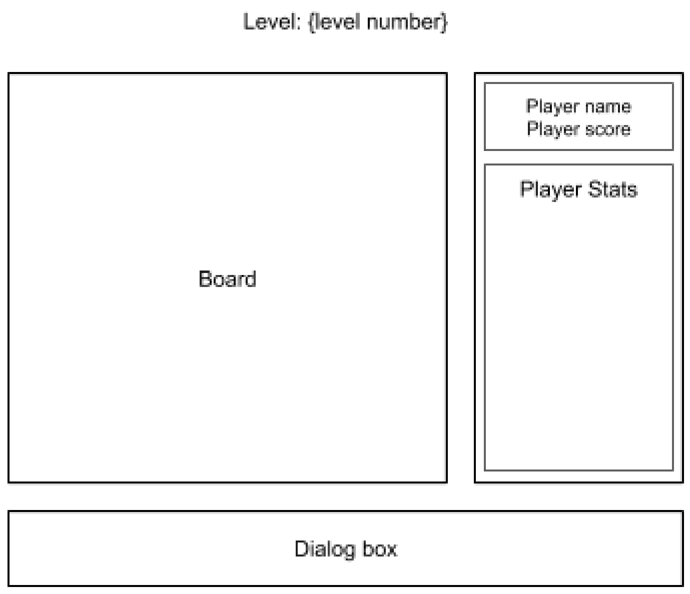
Architecture

Data model

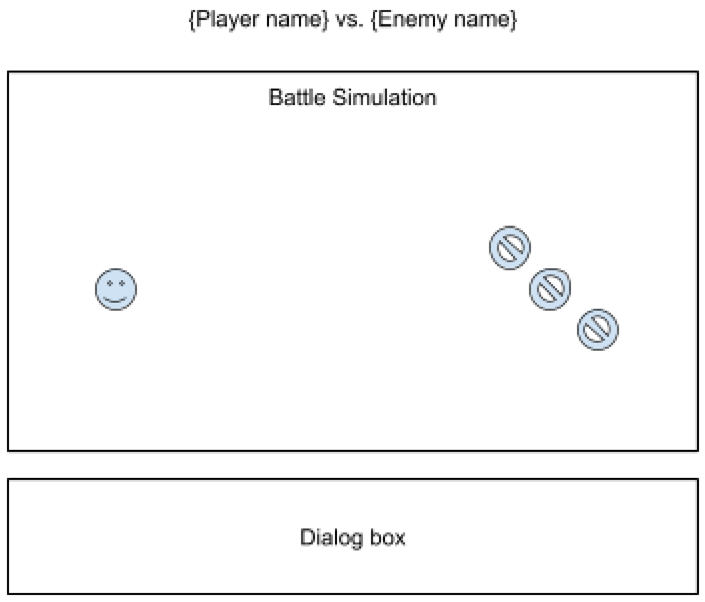
The only data we will be using it JSON files which are used for presets like skills and enemy stats and also for saving the state of the game when exiting. We will load those stats into different instances using the QT framework. The saving and loading of the game will be also done through JSON by serializing objects.

Interface and Graphics

We will have a rendering class which takes in data and renders everything, so that our UI and graphics are decoupled from the logic of the game. There will be three main parts to our GUI. There will be the board which shows the actual game. A sidebar with player information such as name, score, and stats. Finally, There will be a dialog box below all of this which will be in charge of getting user input if need, for example when a player needs to decide whether to keep their current item or discard it for the new item they picked up.



For the battle simulation we will use a very classic interface seen in many turn-based rpgs. This will include you player being on the left side of screen and the enemy on the right. Under the battleground there will be a UI to ask the player what they want to do: fight or escape. If they try to fight the dialog box will display the players skills.



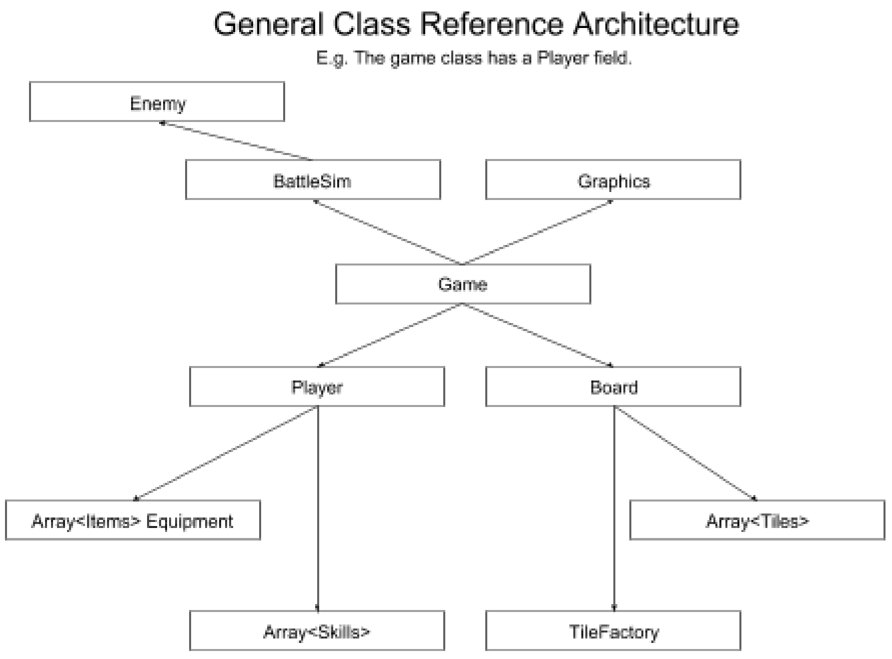
Technologies

QT will the be framework/game engine that we will work in. This is because QT provides a comprehensive amount of libraries and functionality that we can take advantage of. Our other technologies are for development: Git, Github, Travis CI, Catch2. These all work with the C++, which will be the logic. Graphics, sound, and JSON data serialization are all supported and documented in QT.

Class Hierarchy Tree (NOT an inheritance tree)

This is the general structure of our project's classes and data structures. Below, we have outlined this structure in a diagram and a bullet list.

- Game Class: The game's job is to pass all information needed around to all of the classes it has a reference to. As we are going to avoid singletons and passing information directly between classes at the same level, the game will pass all information between the Battle Sim, Graphics, Player, and Board. The game will also be responsible for saving and loading the data.
    a. QGraphicsScene, QGraphicsView: This class will accept information and render the game, including the GUI. The game will, for example, take information about the board and pass it to the Graphics class to render it with every update required.



    b. Board: The board handles all of the functionality associated with the dungeon part of the game, which includes the player, enemies, and the tiles. NOTE: The board is going to be different from the board in the maze game in a few ways.
        i. The board is going to handle the turns and movement for each of the tiles.
        ii. The board is also going to contain all of the references to different tiles (this will be explained further under the flyweight pattern that the tiles follow).
        iii. 3-dimensional array which stores the different rendering layers. Each rendering layer stores classes that all inherit the base Tile class.
            1. Environment (Floors and walls)
            2. Entities
            3. Items
    c. Player Class: Extends Entity class: Holds all of the statistical information about the player, which involves the stats and equipment. This is decoupled from the board and does **not** handle movement. It works closely with the battle simulation class.
        i. The equipment is an array of items: In charge of the equipment of the player

1. The item holds all the information of items, including the skills and modifiers, which alter the player's base stats.

ii. The individual stats in the Entity class extends the Stat class. The reason why we use a class to represent a stat instead of a float is to make it easily modified by items and skills.

  1. The Stat class contains a base value, and a list of modifiers to apply before returning the final modified value.

d. Battle Simulator: A class that handles the turn-based battle between an individual enemy using skills.

  i. Enemy: Extends Entity class. This will be an actual enemy instance (as opposed to the enemy tile used on the board) in which the player fights.

    1. Equipment and skills arrays

e. Start Menu: Simple intro and start button.

Inheritance Trees

We won't heavily rely on inheritance, but there are a few problems that are elegantly solved by them:

f. Entity: The base class which holds basic stats (like health and strength) as well as an array for equipment.

  i. Player: The player requires all the entity functionality, but since it will be controlled by the player, a few methods will be different.

  ii. Enemy: The enemy will contain some logic to choosing a skill to attack with, and thus must include some functionality slightly different to the player.

g. Tile: This base class will contain the intrinsic data for tiles, as well as implement some basic functions that will let tiles interact with one another. In the flyweight design pattern, this class is the class where all the tile instances point to.

  i. The floor and wall tiles will contain a sprite resource path.

  ii. The enemy tile class will contain the specific movement algorithms, the interaction when the player encounters it, and the type of the enemy (so the battle simulator can create an Enemy instance). **Keep in mind that the EnemyTile class and the Enemy that extends Entity are different: one is only for the board and movement, and one is only for stats and the battle simulator.**

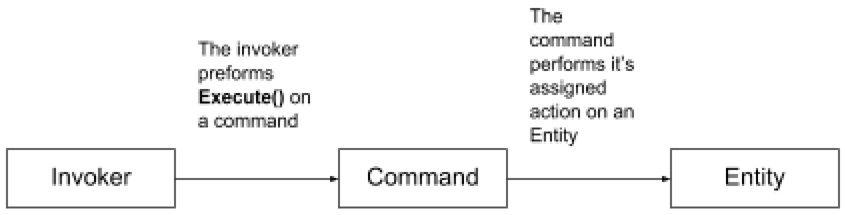
  iii. The player tile class will contain a specific movement method and an interaction method when the enemy encounters it. Again, this player tile class is almost completely separate from the player entity class.

  iv. Interactable tiles (doors, traps). If additional tiles are included, these can do certain things when a moving tile encounters it.

h. Command: For the command design pattern we will have a parent class called Command. The point of this object is to simply create a generic typing for commands and force all commands to have a function called execute.

  i. There will be subcommands which will each be defined to perform an action on an object. For example, we can have a command called "DepthFirstSearch". This command will be passed an enemy class for it to act upon.

ii. We will also have the Invoker. Although it is **not** a child of the Command class it is important in this workflow. The invoker's job is to be passed a command and to execute it. The following diagram is a visual representation of how this all lines up

iii. 
The invoker preforms **Execute()** on a command
The command performs it's assigned action on an Entity
Invoker
Command
Entity

Design Patterns

- **Factory:** The factory pattern works well when lots of instances of objects need to be created but are all different classes. Generating them from a unique id or enum makes things much cleaner with just having to deal with the base class type.
    a. We plan to use the factory pattern to instantiate the entities in the battle simulator. In the battle simulation, a unique id that maps to an Entity with specific stats will be given. The battle simulation will load the enemy's stats from a JSON file, and create the corresponding instance.
    b. The factory can also be useful in creating tiles. We will use a TileFactory in the Board class which will create tiles (floor, wall, enemy, player) based on a string that maps to different tiles. These tiles will be pointed to by pointers in the board arrays. It will make the architecture much nicer if they are all stored as the base Tile class, and thus the factory solves this problem for us.
- **Flyweight:** The flyweight pattern works best when many instances of objects are needed, but many of the fields of those objects remain constant for every object created. The tiles in our game can benefit from this.
    a. We plan to use a version of the flyweight pattern for the tiles in the game. The only data that needs to be stored per instance is the position of the tile; this is inherently stored via the position of the array. The array will contain pointers to the data that isn't instance specific, such as the sprite and movement type.
- **Command:** The command pattern works best when there are method calls you need to call on a variety of objects. The pattern helps to avoid direct calls to the class's method while keeping these calls organized and clean.
    a. We will be using the command design pattern in order to use a wide range of actions on multiple entities in an easy and organized way.
    b. A Specific use of this would be implementing multiple different algorithms that affect how the enemy moves around the map. This would allow us to dynamically change how the enemy will act as the game is going on in a straightforward way.
        i. An example of this is if we have two algorithms, one that makes the enemy search for the player and one that will follow the play. This way we can have the enemy search around the map for the player and once found it can follow. This way it creates a much more realistic opponent.

ii. Apart from being able to change the behavior of an enemy dynamically it also allows us to apply a behavior to any enemy very easily.

Detailed Plan

The overall plan is to have the basics of a procedurally generated JRPG set up and have some content that is playable. In addition, we hope to maintain good practices so that our code is easily extendable because a game like this usually requires lots of content (we won't be putting lots of content, because after all this is a programming project and we want to focus on the programming part).

Timeline

- **Homework 3: Board and Graphics**
    - a. Setup all development frameworks: Github, Travis CI, QT Creator, Catch2
    - b. The board is going to be the first working piece of the game. While the content of this portion is similar to that of the Maze game, the underlying structures are different in our implementation to make the dungeon more interactive.
        - i. Walls, floors, enemies.
        - ii. Generation: Rooms, hallways, and a guaranteed path from entrance to exit.
        - iii. Basic Enemy and player movement.
        - iv. The Game class, which will manage turns and rendering
        - v. Data persistence of the board (saving and loading the board)
        - vi. Unit tests for the board
    - c. After playing around with QT, it would be advantageous to integrate QT from the start, so we want to make this project use graphics as soon as possible.
        - i. The Game class will get the state of the board, and render a scene
    - d. Knowledge required to complete:
        - i. Although we know how QT will generally integrate, it is still the largest unknown, along with getting all development technologies to work together nicely.
        - ii. Data persistence may prove difficult. However, we have knowledge of JSON game serialization in Unity, so JSON serialization in QT will roughly translate.
        - iii. Dungeon generation algorithms: we will use a simpler generation algorithm that will get us rooms and hallways based on a grid.
- **Homework 4: Basic Battle Simulator:** Have a battle simulator that is functional, along with the interface.
    - a. First prototype of the battle simulation
        - i. The battle between the player and the enemy
        - ii. Basic Skills
            1. Data persistence of the skills

        iii. Basic Items
        iv. Stats and Modifiers
        v. Interface for the battle simulation
        vi. Unit tests for the battle simulation
    b. Knowledge required to complete:
        i. We don't know how to use QT specifically for GUI, but we should learn those skills in time for this homework 4.
        ii. We will be working in JSON reading and generating objects with unique stats, which we have some experience in Unity.

- **Checkpoint:** At this point, we won't be entirely sure how much progress we've made, though we hope we are finished with the basics of the game.
    a. Generate content/diversify the game
        i. Move beyond basic enemy movement
            1. Bosses
            2. Different movement algorithms
        ii. Procedurally generate items
        iii. Improve dungeon generation
            1. Fill in rooms of dungeon with objects of interest e.g. chests.
            2. Create different types of rooms to make the dungeon a bit more diverse (treasure rooms, trap rooms, etc.)
        iv. Make enemies choose their skills intelligently in the battle simulation.
        v. Unit tests for these new additions
    b. Knowledge required to complete:
        i. We will need to research movement algorithms and/or come up with our own
        ii. As we are taking *Intro to Artifical Intelligence* we know enough to make our enemy AI good enough to make the game fun.
- **Final Deadline:**
    a. Add the main menu, and other finishing features.
    b. Up until now, we've been trying to implement a ton of features and mechanics, so it will likely not look too great and it probably won't be balanced enough for fun gameplay. So, we will focus on refining and polishing the graphics, sound, and gameplay.
    c. Ideally, we are not adding any more features, but getting the game ready as a product.
    d. Preparing for project presentation.

Links To Resources

Github: https://github.com/thyo9470/csci3010-indavidual-project
Sound FX maker: https://www.bfxr.net/
Additional indie game resources: http://www.pixelprospector.com
QT JSON Game Serialization: http://doc.qt.io/qt-5/qtcore-serialization-savegame-example.html