Dithered - A top-down dungeon rpg game

By Jack Mao and Dieter Brehm

Project Overview

Our project is a top-down “roguelike” styled video game which we built using the Pygame python library. Inspired by games which ran exclusively in the terminal, this game utilizes a very particular graphic style implemented with low-resolution pygame sprites. The game world is explored by moving from level to level, fighting enemies, and collecting items. It features random level generation, collision detection with enemies and the map walls, characteristic modifying items, damage dealing enemies, and a unique art style.

Results

We created a hybrid between a pixel rpg-game and a roguelike game where you are the void locked inside a dungeon, and the only way you can go is down. Having nothing better to do, you decide to go down the dungeon to explore. But the dungeon is a dreary place, and the only thing that awaits you at the end is certain death. So, explore as long as possible, before the inevitable occurs.

This game uses minimalistic style of black and white pixel art to depict the dark atmosphere of the game. The gameplay is designed in such a way that you can only walk along the tile-based system of the game. There exists many different tiles, for instance, portal, the floor, wall, item, enemy, and hero tile. The player controls the hero tile and weaves around the floor tile, looking for the portal to go down while trying to survive.

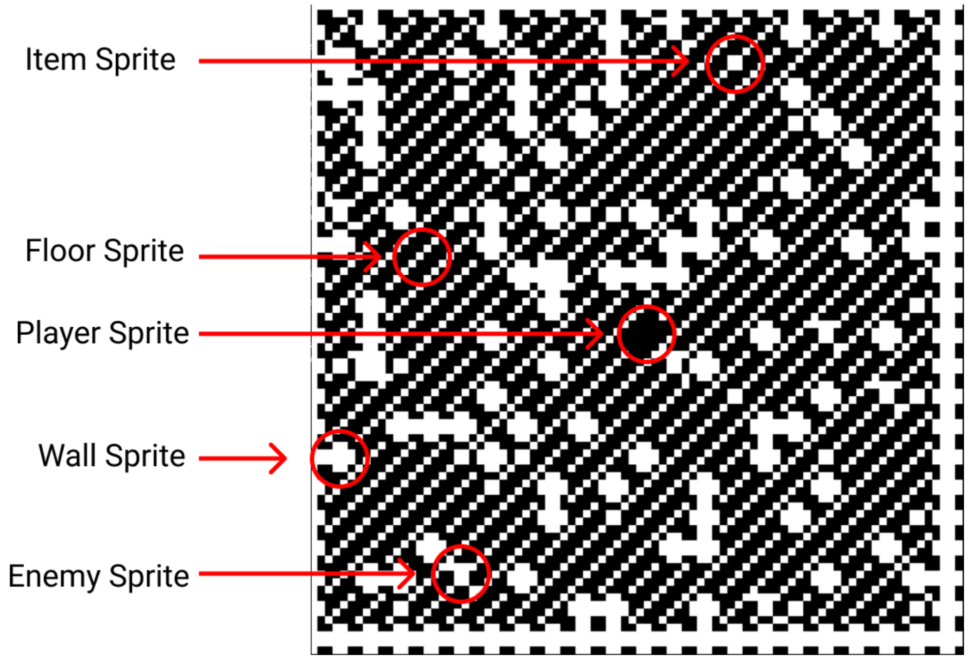
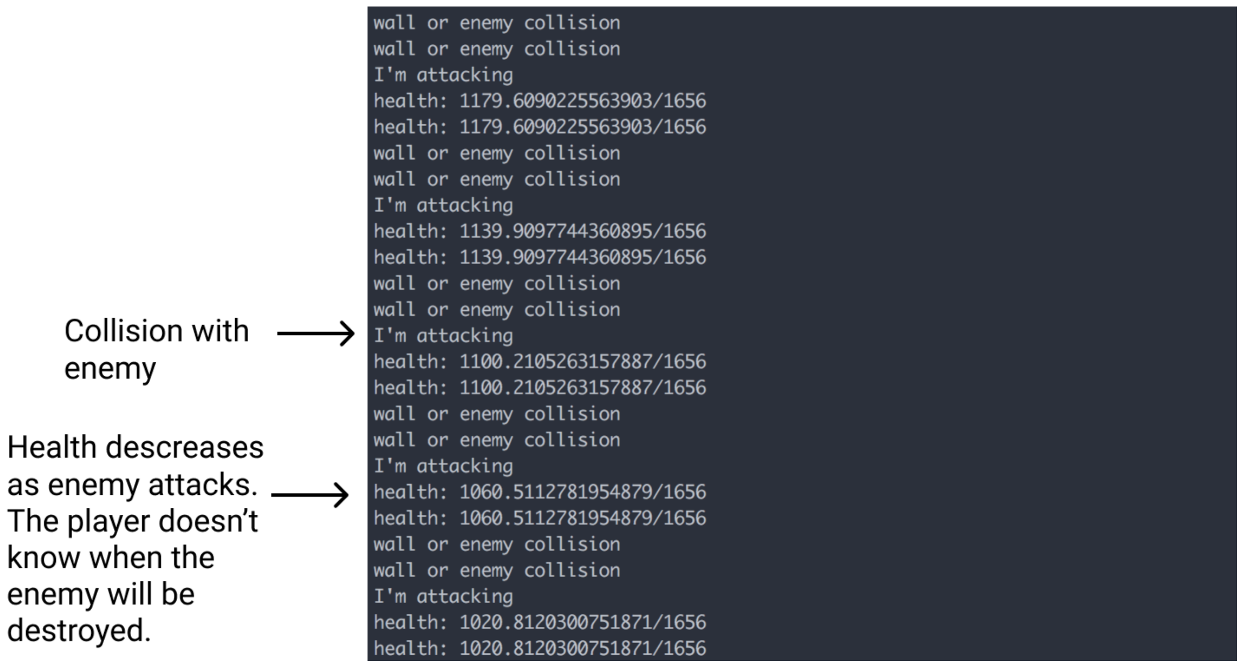


Figure 1: This is a diagram displaying the different types of tiles that exists within the game.


Collision with enemy
Health descreases as enemy attacks. The player doesn’t know when the enemy will be destroyed.

```
wall or enemy collision
wall or enemy collision
I'm attacking
health: 1179.6090225563903/1656
health: 1179.6090225563903/1656
wall or enemy collision
wall or enemy collision
I'm attacking
health: 1139.9097744360895/1656
health: 1139.9097744360895/1656
wall or enemy collision
wall or enemy collision
I'm attacking
health: 1100.2105263157887/1656
health: 1100.2105263157887/1656
wall or enemy collision
wall or enemy collision
I'm attacking
health: 1060.5112781954879/1656
health: 1060.5112781954879/1656
wall or enemy collision
wall or enemy collision
I'm attacking
health: 1020.8120300751871/1656
health: 1020.8120300751871/1656
```

Figure 2: The terminal updates information of the player’s stats during the gameplay.

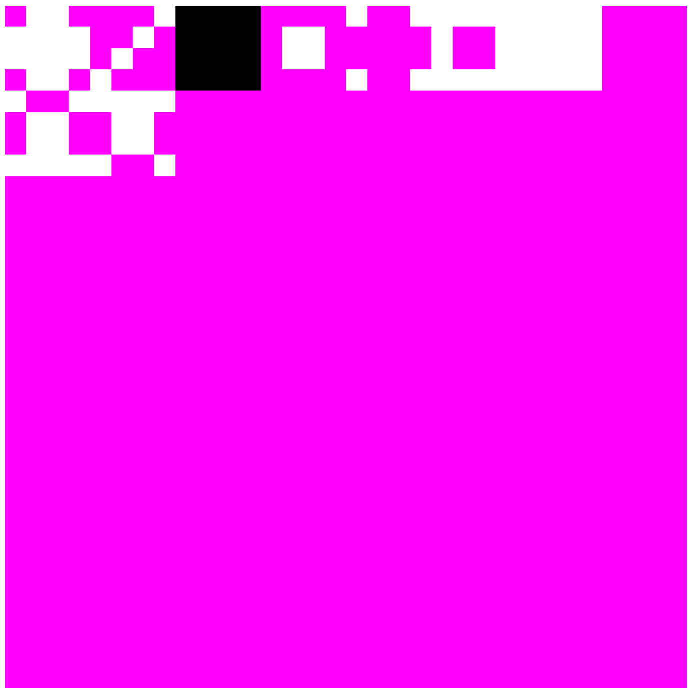


Figure 3: This is the spritesheet that we created to make the graphics.

Implementation

The logic of the game is predominantly controlled by the Characteristics class. This class stores all the information that a hero and enemy has, such as health and items. As of now, the only relevant data stored are current health, max health, armor, and speed. The HeroSprite class and EnemySprite class both have a characteristic as an attribute. One design decision which we had to make was about how the to handle the interactions between the enemy and the playing character(The “hero”). Because in our current implementation only the hero would initiate a battle, we created a handler inside the HeroSprite class which takes an EnemySprite as a parameter. Inside that handler, we interface between the character class of the hero and the enemy. This allows for a possible future implementation of multiple enemy types.

The view and controller elements of the game are controlled by the main RogueLike class and the Map class. The RogueLike class handles the event loop that keeps the tick of the game consistent and includes a number of utility methods. In the case of this project, this class also ties the project together, holding instances of the other classes as per the UML diagram.

The Map class handles the views of the game, namely the generation of sprites through random generation. This includes items, enemies, and the map elements such as the walls, the floors, and the next level portal. For each placed sprite item, we track the data and location about that specific item through a dictionary that contains all of the different categories of sprites which are organized using a pygame group. For instance, we have a “walls” group that is a list that holds all of the wall sprites on the map.

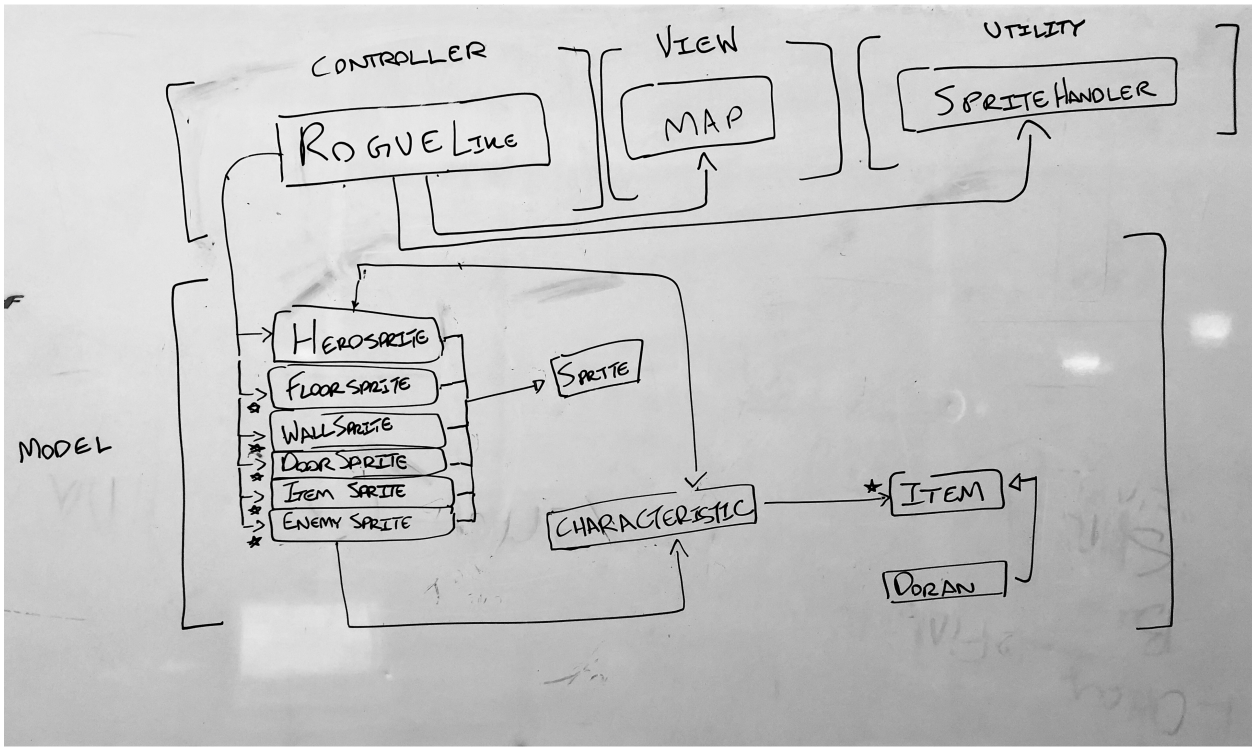


Figure 4: UML diagram of the video game program

Reflection

We created a MVP of our product. This is just a proof of concept right now, with only one type of item and one type of enemy. However, this was very well modularized. We can easily create more types of enemies and items. The project was appropriately scoped because we created the proof of concept, which is a sizeable amount of work over the course of two weeks. We divided work into two parts, the game logic and view/control. Jack worked on the game logic and Dieter worked on the game view/control. There were no particular issues when we were working together. We managed our work using multiple branches on a git repository, which allowed us to minimize the number of code conflicts between our computers.

As far as scope, we may have started with an overambitious scope. We had a number of stretch goals which we had hoped to reach but ended up not reaching. These include a fog of war and other lighting systems, along with animated enemies. These would be excellent next steps to take this game. Additionally, while developing the map generator, we realized that there is a lot of improvement to be made in validating a map generation. For instance, next steps could include more advanced generation that forms specific room shapes.