Dig Dug Design Manual

Introduction

This project is designed in a very intricate set of class relations and various uses of files and Java tools. To begin, we had to deal with a way to incorporate locations, movement, and images. So, right off the bat, we decided that we would use some sort of grid mapping system in order to accomplish this. We created a board with capability of holding very specific locations and points. Then, we needed to figure out a way to store static images and moving objects. Any type of object has something called a Vector2. This is essentially a location within the board. Any moving character, objects, items on a screen, etc, all store a Vector trait in order for our graphics interpreter to know where to place them on the screen.

Each type of object or character each has their own class to hold their specific functions and traits. They all have methods that are able to handle movement, shooting, pumping, dying, etc. In addition, there are methods to handle things such as transitions, sound, score, and other aspects of the game that are not necessarily placed on the board, but have an intricate role in the carrying out of this game.

We also followed a very strong MVC model.
All of our low level, game functionality classes are placed in the Model of our program. The view handles the screen images, and the control handles updating the view from the model. It is constantly checking states within the Model to make sure everything in the view is being output properly.

Finally, our program is all run through a main function.

User Stories

As a user I would like a menu in order to manipulate the game.
As a user I would like a main character in order to traverse the levels and gain score, shoot, number of lives, etc.
I would like to have a score tracker function in order to keep progress.
I would like to have a smooth graphics interface in order to play the game smoothly.
I would like to have a complex enemy system in order to complicate the gameplay and allow the game to progress.
I would like to have items on the board so that I can have collectibles.
I would like to have a sound system to represent the different states of the game.

The main menu is completed through a simple JFrame Panel and a png static image. There is a PNG image that represents the main menu image. We use some low level functions to load the PNG image into the JFrame, and display this image when we start our program

Next, we created a Driller.java class that handles the next user story. This class handles the main characters images, as well as the ability to move, keeping track of his direction, his state (dead, alive, drilling, not drilling, pumping, etc.) It holds all of these capabilities in order to carry out different tasks that the user may want to complete.

For score, all we did was assign specific point values to different events that happen within the game. As they occur, we add these scores to the total score of the game. There is also a ScoreKeeper.java class to handle this.

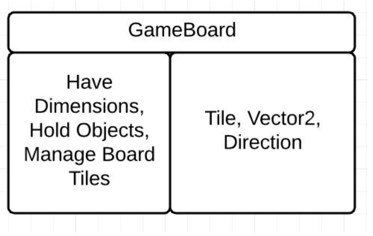
The graphics system was probably the most difficult to accomplish. We started by creating PNG images of every aspect of the game. Then, we created a JFrame to hold our board. This board has a very complex grid. In order to display our PNG images, we go through each individual pixel of a PNG image and load it into the grid. For objects, and fluid graphics, we stored different states of movement in a Hashmap and quickly cycle through them to display movement.

The items and enemies are implemented the same way as our Driller.

Finally, as we did with ScoreKeeping, we assign a function call to individual events. This all calls a Sound.java function that has threads set up for all different events.

OOP

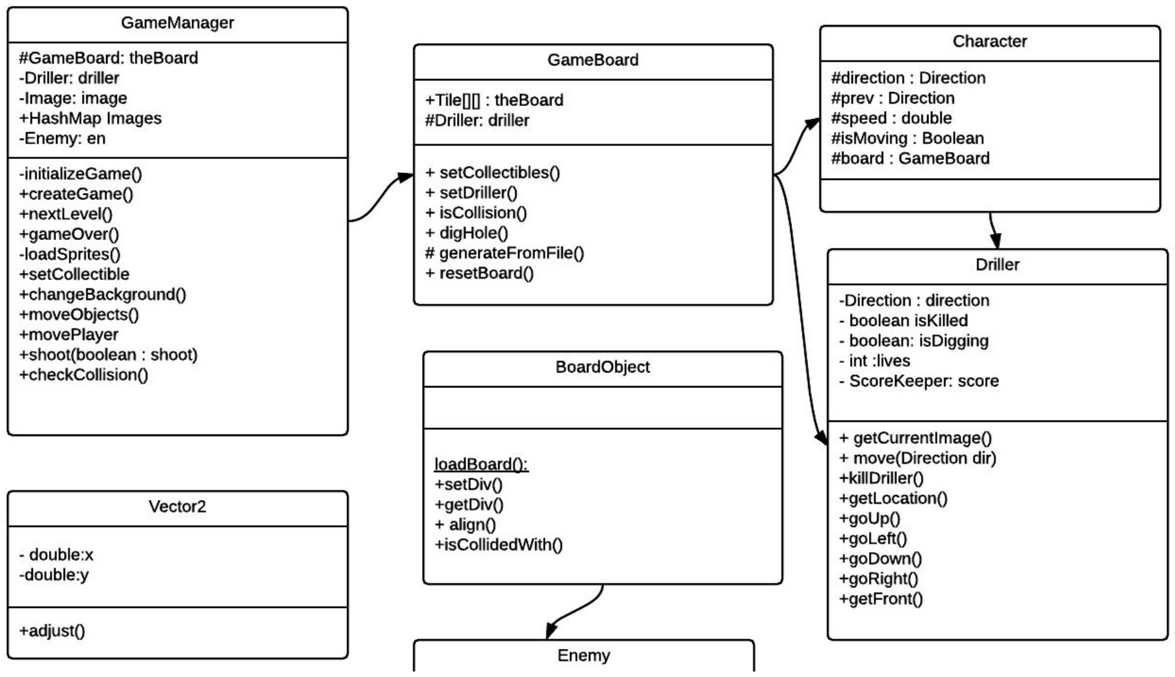
To begin we had to set up our board. You can see that this class relates closely with Tile, Vector, And direction. We needed a way to track the direction things are going, and it's actual location. The GameBoard class holds a grid structure, which is split up into tiles. Each tile also has individual blocks as well in order to hold pixels. Vector2 is a class that holds informations about location within the GameBoard structure.



The Board itself was also designed to hold objects, which are managed by the BoardObject Classes. All of these classes manage orientation, location, state, etc.

Next, we have a Character class, which inherits the BoardObject class. This holds direction, location, Vector2, and other uniform things that moving characters will have. Then we split this up into a class for each character (Driller, Enemy). The driller holds all of the relevant methods to handle the main character, while the enemy class deals with both the Pooka and Fygar classes.

Furthermore, the BoardObject class also extends to collectibles, and other graphics entities such as Holes. These are all static images that jut need to be present on the Board, and will need things traits such as Vector2 - and a location on the board.



As you can see in the Diagram, there is a large dependence on the GameBoard and BoardObject Classes.

Now, we also use a strong MCV model for handling the rendering of the state of the board - and also making sure the graphics are up to date. There is a timed loop within our controller that continually goes through a set of checks and event handlers, which will trigger changes within the Model, and also update the interpretation of the view.

Then we move on to class structures that handle extraneous tasks that need to be carried out in the program. This includes things like a ScoreTrackers and MenuHandlers. These functions take care of handing static graphics that need to be frequently updated on the screen. Although they are Objects on the board, they do not need to inherit the BoardObject class because they do not interact with any other object.

There is also a sound class, which is called and creates a thread loop for different events. All of the sound files are .wav files stored in the src package, and are easily accessed through file calls

All of these methods have are managed and handled by the DigDugMain Package. This initializes a controller which begins the program, and then begins the game loops.