Design

System Description.

Core Idea. We plan on implementing a 2D precision platformer video game similar to I Wanna Be The Boshy and Super Meat Boy.

Key Features.

• Have checkpoints in the game which allows the game to restart on a previous saved state on death.

• A GUI for the video game that accurately represents game state. • A boss(es) that challenge the player.

• Obstacles for which the player must avoid to stay alive.

• Multiple unique levels that challenge the player.

Narrative Description

We intend to build a 2D platformer which is mostly obstacle based. Unlike Mario which has enemy AI that move around the maps, most of the game will be avoiding obstacles such as spikes which are stationary. The game will also have set stages which do no move with the player. As the player character moves to the end of the stage, usually located on the right of the stage, the game will transition to another stage. So, it will not be a side-scroller. Because the obstacles are mostly stationary, the game will be more precision based in which the player character only has a tight space to avoid obstacles. This presents the main difficulty in the game as the player must be relatively precise in their movements. To keep the game fresh, there will also be other "gimmicks" that are stage specific. For example, we could reverse the player controls or put the stage upside-down to present more of a challenge.

For a game like this to be playable, we would definitely need to implement a GUI, so the player could see where the player character is and where the obstacles are. This GUI will display the state of the game and update automatically as the state advances. There will also be options to pause the game, restart the game, or to quit the game. The GUI will feature color and will be entirely 2D.

Checkpoints in the game will be implemented as certain flags in the game in which the player must move over to save. After this, if the player dies, the game restarts with the player located at the save flag. This isn't a pure save as the game won't remember the true state when the player moves past the flag. Only that the player is at the flag. This won't matter however as most of the obstacles will be stationary and thus does not change location relative to a given state.

The state will be similar to the game state that was implemented in A2. Of course, it will be much more complicated and keep track of different parameters. Different modules will be used to the different obstacle types such as a spike module. These modules will implement the size of the obstacle and the hitbox of the obstacle.

Finally, bosses will be different. They will move randomly and shoot projectiles at certain location. Of course, the boss will have a set attack pattern of which the player should learn through playthroughs. Because the boss will also change game state, there will be no checkpoints within the boss area. If the player dies during a boss fight, the player will revert to a checkpoint that will be outside the boss area and must access it again. The boss movement

pattern will be hardcoded and be cyclical. The projectiles that shoot out of the boss will be random or pseudo-random.

Module Design

State Module: This module will keep track of the current state of the game. This includes the position of all the entities within the state as well as the current level that the player is on. The state also keeps track of the various movement parameters surrounding player such as acceleration of the player model and the position of the model.

Physics Module: This module will include the main physics functions that pertain to entities. It will update the position of an entity based on various parameters such as velocity and acceleration. Essentially, the module will keep dictate how movement works in the game.

Main Module: This module will run the game and keep track of collisions and such. The module will initiate the game and call upon the other modules to run the game.

Level Module: This module will store the various level designs. The levels will be structures that place various entities in various positions. The level will also have a start position and exit position.

Entity Module: This module will store the various obstacles, projectiles, tiles, and everything physical the game contains.

Command Module: This module will interpret the player inputs and change the state of the game accordingly.

Data

The data that the game needs to maintain isn't much. Each level is self-contained which means that we only need to access one level at a time. Each level is hardcoded and at most contains a relatively small amount of data. Each tile is like one piece of data and a reasonable level would be bounded by around 600 tiles. Currently, all the data structures that will store the information of the game are lists. This is because we aren't storing large sets of data. A large twenty by twenty tile level only stores 400 pieces of data which a list can handle without much difficulty. If we do run into time complexity issues with our current state structure, we have considered storing data in structures such as a hash set.

Third-Party Libraries

We anticipate that we will use a third-party library for the graphics of the game. The game requires a nice GUI to accompany the platformer which we plan to implement in Java. This will be done with an ocaml to java library. We are all more familiar with graphics in Java so it would be easier to use this library to implement the graphics. Information about the library can be found at http://www.ocamljava.org/.

Testing

While there will be extensive testing of each module, the only real way to test a game is by playtesting. Before the GUI has been implemented, we will write black box test suites for each other. Glass box testing will be implemented by the person working on the module itself. After a basic GUI is implemented, we will test the program by playing the game and trying to break it within the game. Because we will split up the program by modules, bugs within a module will be fixed by the same person implementing the module in the first place. We also will send out the program to our friends to playtest and report any bugs that they find. By

playtesting, we can also modify the code to introduce any quality of life improvements to the game that the code would not show.