Project Overview:

We created a simplified clone of the game Super Smash Bros, with an emphasis on physics and multiplayer interaction. Like regular Smash, the goal is to eliminate every one of your opponent's stocks by knocking them off the stage. Our game however, leaves out many of the visual and mechanical complexities of Smash, and focuses mainly on character interaction.

Results:

We were able to create the game with successful physics and interactivity, such that at the end of the day we had somewhat of a playable game complete with collision detection and unique interactions between characters as the game progresses. The game possesses a basic physics engine, the ability to move, shield, attack, and the mechanic that increases knockback as damage is taken. In conjunction, these most essential components allow for two or more people to duke it out.

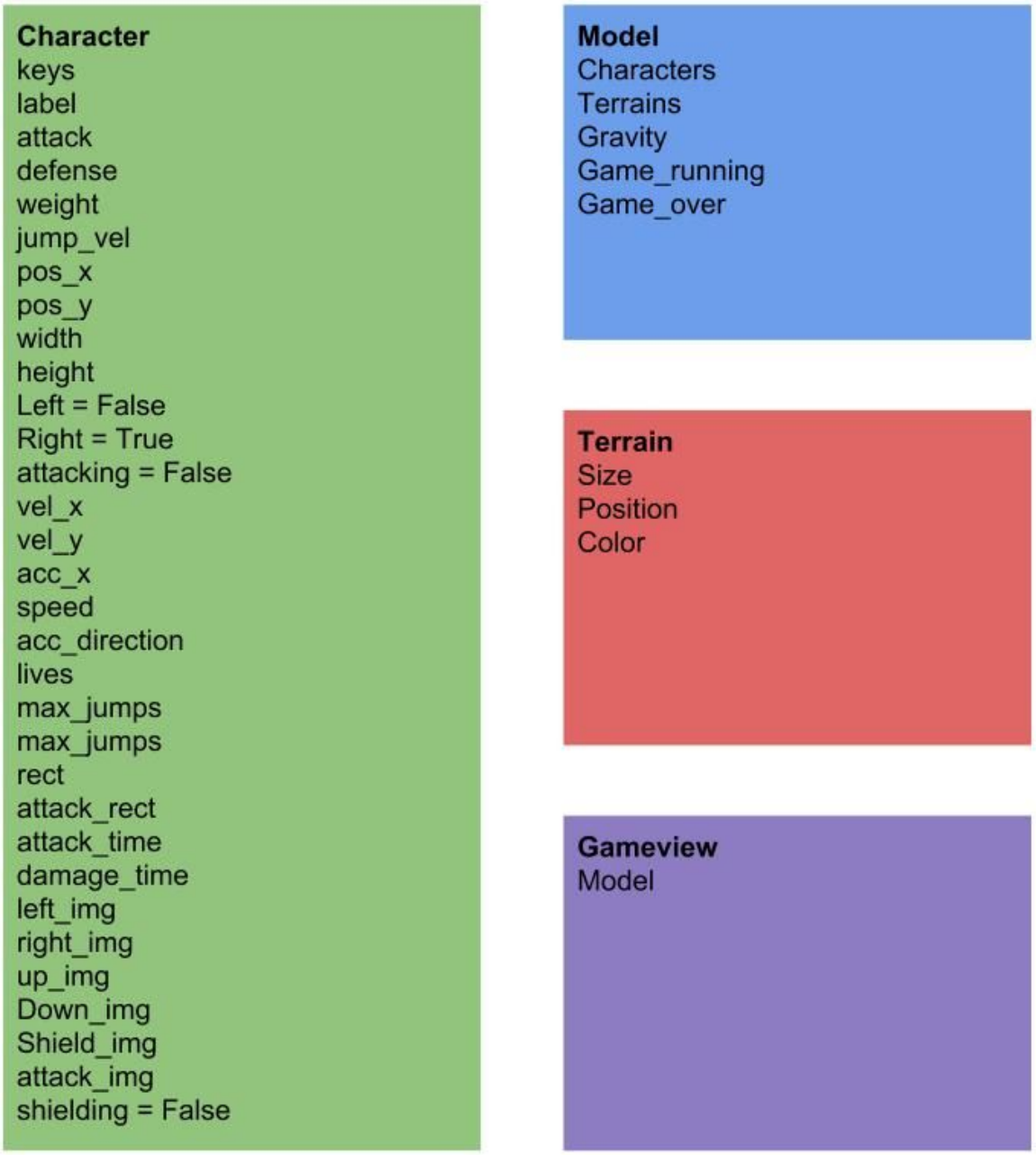
That being said, our game is lacking in a multitude of different ways. To name a few, motion is not as fluid as it should be, there are no dodging mechanics, there are no projectiles, there are not many sprites to indicate attacks and movement, and there are not many directional attacks. Because our game lacks many of the nuances of the real Smash, it lacks a any real depth in gameplay.

Implementation:

The main classes of this project were model, character, terrain, and GameView. The model holds all of the character and terrain objects, and contains functions for converting key inputs to updates to their states. This can trigger a shield or attack, or update x and y motion before updating total motion (which happens at every repeat of the main loop, but has no effect unless the x or y motion functions are called). Then there is the terrain class, which is mostly just a rectangular object that displays on the game screen and can be collided with.

The main object is character, which contains most of the physics for the character movement, as well as movements, sprite skins, lives, and more. The character class also contains multiple attributes, which are intrinsic and immutable properties of each instance of the character class, and multiple states, which change over the course of the game as a result of external stimuli. An example of an attribute of a character would be its speed, or jump height, both of which stay constant throughout the course of the game. An example of a state would be the number of lives the character has left, which changes as the game goes on.

In terms of the architecture of the program, the way in which specific methods and attributes were distributed across the different classes didn't really make that much sense. This was partially due to poor initial high level planning in the beginning of the project, and partially due to how the physics engine of the game was implemented. Some methods within the model class directly update specific states of instances of characters, whereas some methods within the model class simply call methods within the character class, which then update the state of the character. Ideally, our model class methods would either exclusively call other methods within characters, or exclusively update character states--not both. Because we used both ways of updating the character, it made the program as a whole a lot less organized and a lot less readable, which only compounded existing complications with git.



Reflection:

Our process was pretty efficient, with a couple of hiccups. We chose a divide and conquer approach, which mostly made sense given our respective skill levels and schedules. This was incredibly useful, except when there were problems with github or code that was not detected by the person responsible for pushing the previous step (which happened with both parties a couple of times). By the end of the project we really got the hang of collaborating over github and this was much less of a problem, so this problem actually was a pretty good learning experience.

The biggest changes that would be good to make for next time are figure out more concrete expectations for passing things off, and scoping the project more accurately given resources and free time. In addition, it would have been more helpful if we had spent more time at the very beginning deciding on high level details of the project in order to more effectively work individually, and to make the code's structure more organized, and thus more readable.