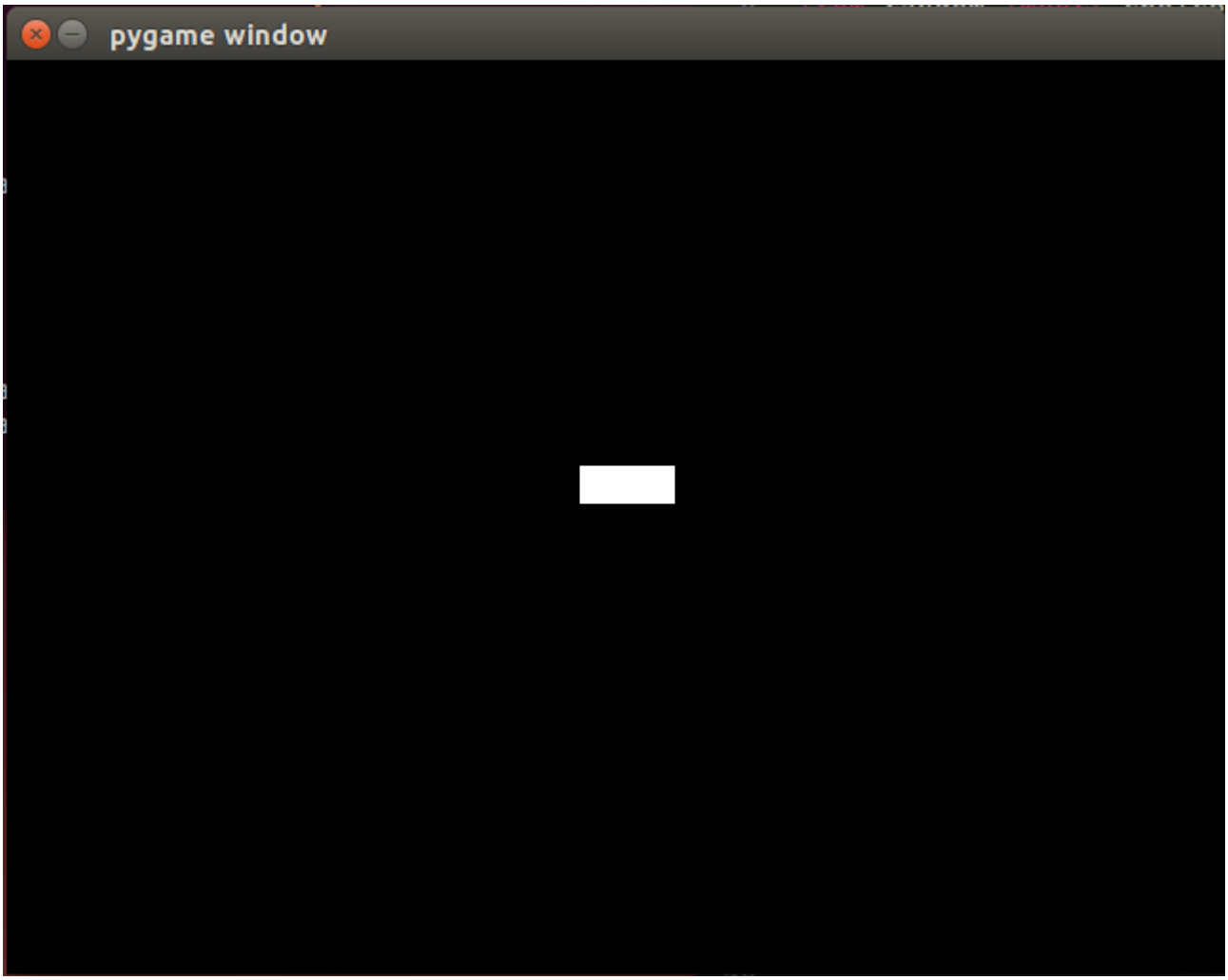
Interactive Programming Project
ENDER'S GAME
Maximillian Schommer
Claire Kincaid
March 10, 2016

Project Overview:

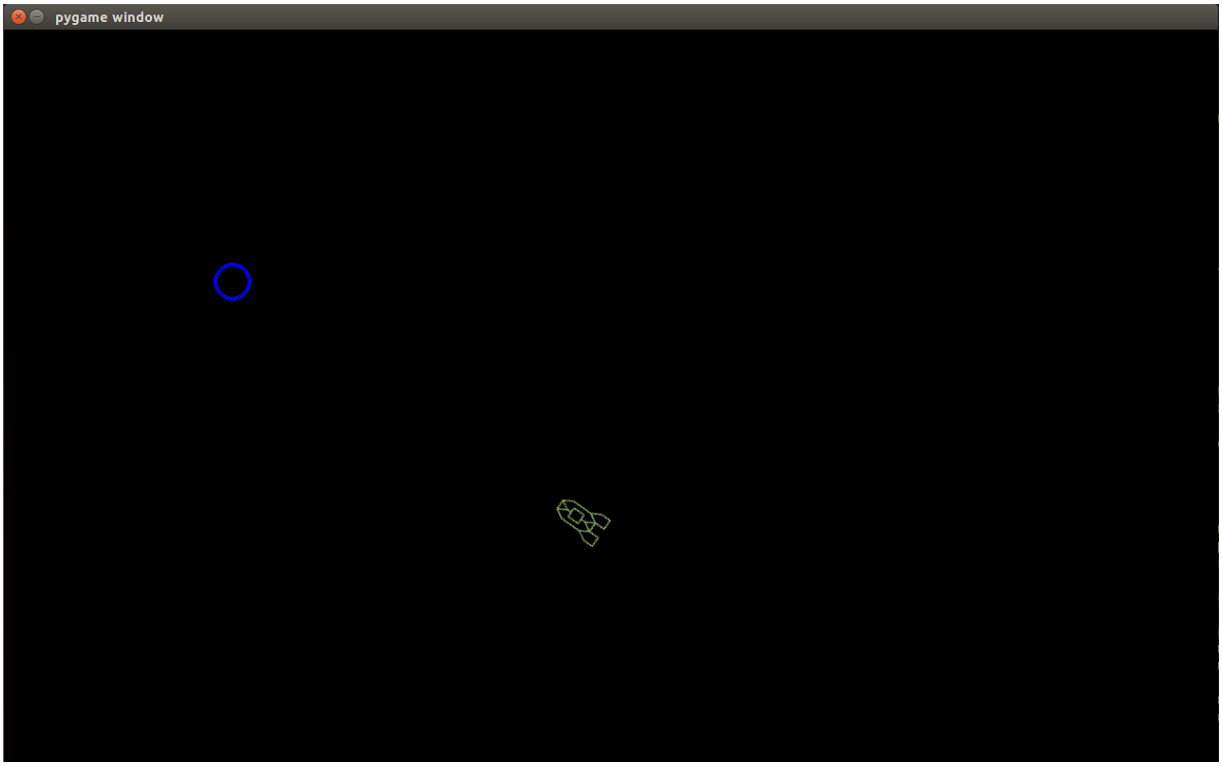
For the interactive programming project, we decided to create an arcade game using a combination of pygame and opencv. This arcade game is based off of the “simulations” from the science fiction novel-made-movie, Ender's Game. The goal for the arcade game was that two players, one controlling ship movement and one controlling missiles/lasers, would play at one console, controlling a fleet of ten ships to destroy a fleet of enemy ships. The User's fleet would be controled using computer vision, allowing the user to move colored balls in front of the camera to control movement. The missiles would be controlled by keyboard controls.

Results:

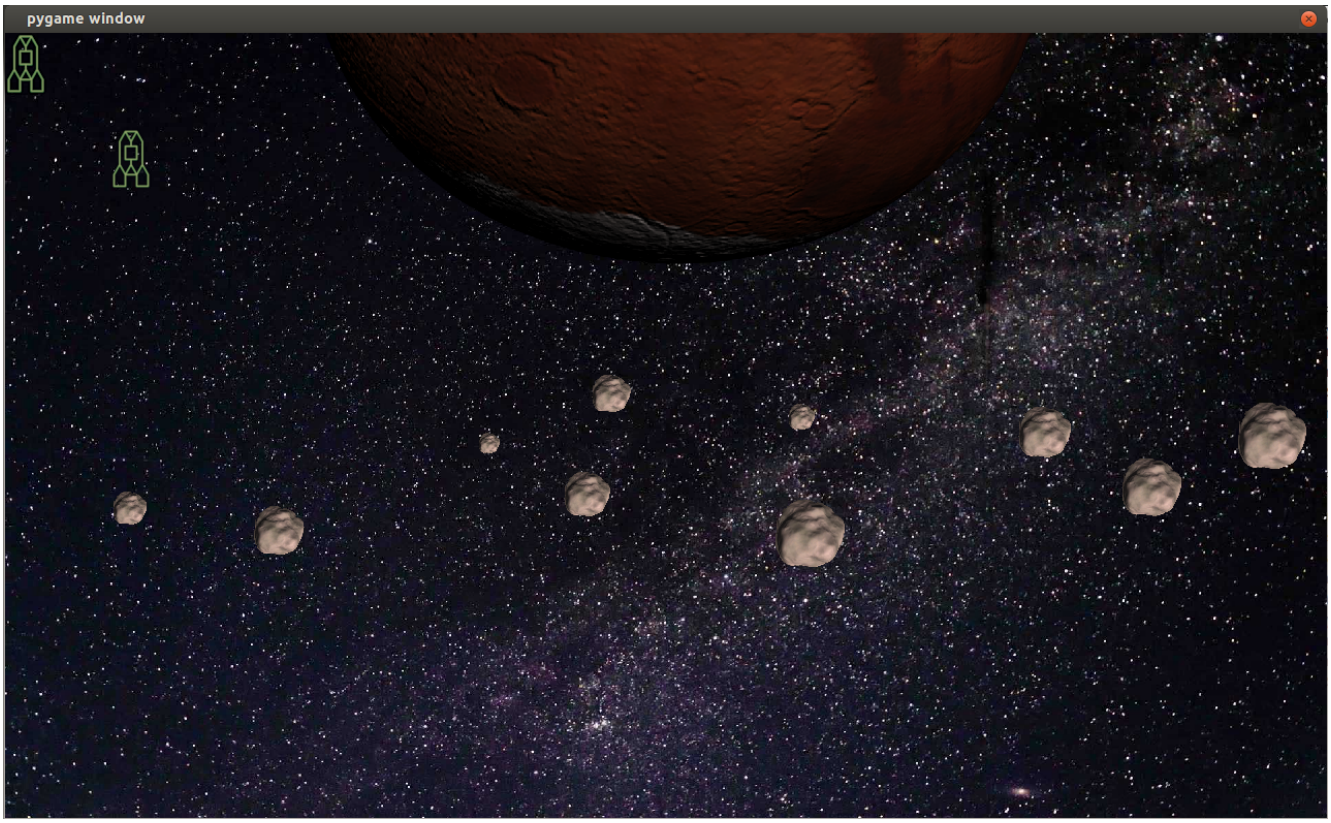
Thus far in this project we have accomplished two iterative drafts and are in the midst of debugging our third draft, with game play. Our first draft, which was created merely to understand the mechanics of pygame, is merely a brick that the user can control using the arrowkeys or their mouse. It is pictured below:



Our second iteration of Ender's game was created to understand basic game mechanics of user controlled ships and the implementation of open cv. It consists of a ship image which rotates and follows a blue dot that is created by computer vision tracking a yellow ball (or even sticky note) across the camera. It is pictured below:



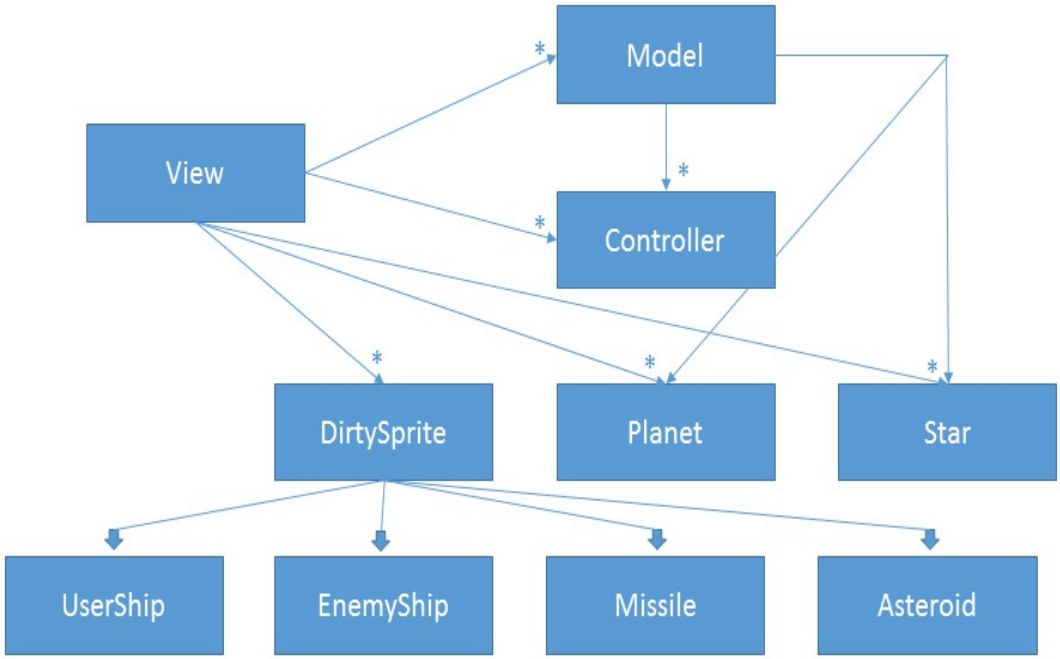
Our third iteration of Ender's game is a complex arcade game utilizing keyboard and open cv controllers. True to the nature of the project, we heavily utilize classes and object oriented programming. We have several sprites and experimented with both sprite and dirty sprite classes. At the moment, our third iteration code is in the process of debugging, due to some method and inheritance issues. However, we do have ships utilizing our Computer Vision controller to move around the screen. Our current gamestate is pictured below:



Implementation:

Our game consists of a view class, a model class, a controller class, six in-game object classes, and a game loop. Our game view visualizes the game in a pygame window and then draws the game state to the screen. It takes in as a parameter the model. Our model class stores the state of the game-- the positions of each of the user and enemy ships, the arsenals (number of missiles) of each user ship, the number and position of asteroids (sprites that absorb missiles), and the position of the home planet on the game screen. It takes in as a parameter our controller class, which uses color tracking in Open CV to control the user's ships, which will accelerate towards the dot tracked by computer vision. Our controller also uses keyboard controls, assigning a different key on the keyboard for each user ship. When these keys are pressed, it calls “Shoot” a method within the usership class that shoots missiles. We then have our six in game object classes: Star, Planet, Asteroid, UserShip, EnemyShip, and Missile. Star and Planet are merely background objects that draw to the screen to create the game environment. Asteroid is a class which inherits all of the methods of the pygame.sprite.DirtySprite class, allowing it to constantly update. Asteroids exist for the purpose of absorbing wayward missiles and serving as an obstacle between the two fleets. UserShip, EnemyShip, and Missile also all inherit the class pygame.sprite.DirtySprite, allowing them to update with game state as defined by collision. These classes take in a name, x and y position, angle, and velocity. Usership also has an optional parameter 'arsenal'

which has a default value of 10, which defines the number of missiles each ship has. When usership's method “shoot” is called, it launches a missile, decreasing the arsenal by 1. Our game loop consists of definition of view, screen, model, and basic collisions. Since most of our sprites are rectangular, we used basic collision mechanics for this iteration. A UML diagram of our game mechanics is displayed below:



When designing our object classes, we knew that to sense collisions we needed to have them inherit the qualitites of a pygame sprite. However, we had to choose between using a sprite, which updates only when the entire game state updates, or a dirty sprite, which updates on its own independent of the game state. We chose dirty sprite to allow each of our sprites to update, which should in theory allow our game to run faster. We then had to debug all of the dirty sprite params, and are currently still working on utilizing dirty sprite to its full potential.

On our To Do list for continued project programming, we have extensive debugging, in addition to the full usage of the dirty sprite parameters. We need to work on mechanics and graphics for our missile class, and we also need to implement movement and shooting for the enemy fleet (we're leaning towards a basic pattern programming or very simple AI level-by-level). We need to implement levels and music of some kind, and would like to implement our optional “laser” class, which ideally would create an instantaneous white line towards the ship we're aiming at, which would destroy them. For this to work, we will also need to extensively

edit and advance pygame's built in collision detection programs.

Reflection:

From a process point of view, our early iterative programming went well. However, around this past weekend, we somehow lost sight of iterative programming and just began writing code without testing it. This was a terrible idea. We are now in the process of debugging that code and developing our game mechanics further. Additionally, our object were programmed well, though we would like to further be able to utilize the abilities of pygame's dirty sprite class. Rather than unit testing, which we thought would be less applicable for an arcade game, we iteratively compiled our code in python step by step to make sure it was doing what it was meant to do. It was difficult for us to think up doctests that would serve the same purpose as observing game mechanics in the pygame window. We both wish we'd had more time and more experience with pygame/opencv before we began the project, as we feel that would have allowed us to make decisions that were more scalable to the time slot of this project.

We feel that team processes were implemented well. We split the work up based on subject: Schommer studied opencv and Claire studied pygame. We then switched subjects so that we had a better understanding of eachother's subjects as we moved forward in the game. Our early iteration code was more of a divide and conquer affair, while our third iteration is much more collaborative, pair coding. Mostly our issues were derived from a lack of time on either member's part which occasionally made working together difficult. Next time, we would schedule team meetings more effectively and have a better idea of what needs to be done when.

We plan to continue this project after spring break as our “Extend one of your mini project” assignment. For that assignment we hope to have our completed third iteration up and running, ideally with laser class implemented.