MOTHERLOAD:

Cabbage Rendition

By Willem Thorbecke and Sung Park

Project Overview:

In this project we made a 2D mining game called Motherload: Cabbage Rendition. It is a spiritual predecessor (defined by its increase in glitchiness and decrease in playability) to the beloved Miniclip game Motherload. In this game the object is to mine underground for minerals on Mars and return to the surface to refuel before your fuel runs out. The minerals you mine can be sold and then a fuel tank size upgrade can be bought. The game ends if the player runs out of fuel. The goal of this game (in this simplified iteration) is to end up with the largest fuel tank possible as a byproduct of collecting and selling the most minerals.

Results:

The game world with all the necessary components of the game - buildings, ground blocks, and minerals, are generated at the start of the game. As seen in Figure 1, the mining vehicle, represented by a white square located at the top center of the screen, is placed on the top of the ground. Beneath the vehicle is the underground mine, consisting mostly of brown "soil" tiles and occasional empty pockets and minerals.

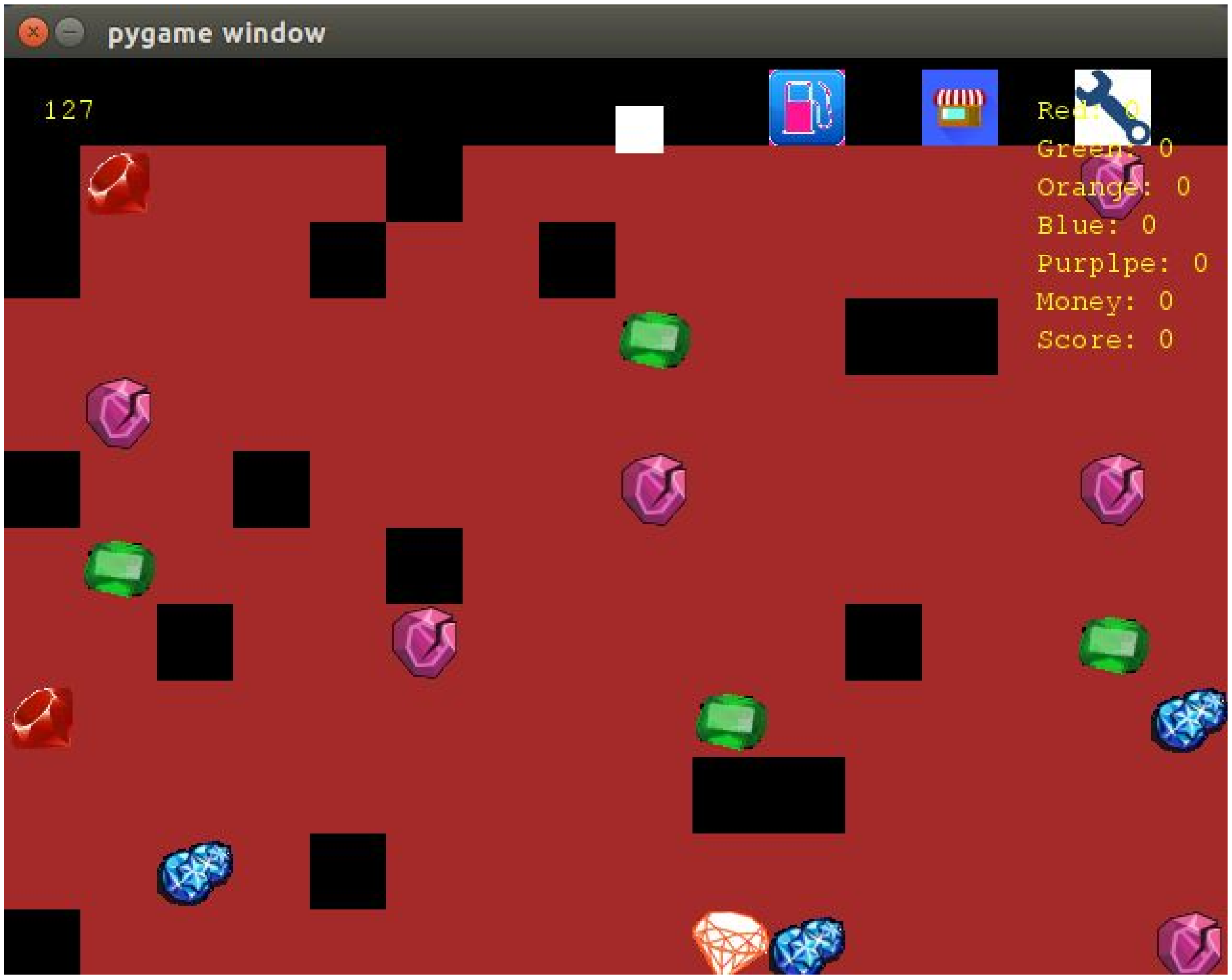


Figure 1 - The start of the game. All the support buildings can be seen on the right of the vehicle.

There are three support buildings for the vehicle in this game - a fuel station (Gas station logo), a store (store clipart), and a workshop (spanner). When player visits one of the support buildings and a certain condition is satisfied for the specific building that the player is visiting, the special effect of the support building is applied. A player refills the vehicle's fuel tank to max level by visiting the fuel station. Visiting the store will allow the player to sell the minerals that the player collected for 100 credits each. Visiting the workshop will increase the max fuel capacity of the vehicle by 500 units in exchange for 500 credits.

There are displays for the fuel level and other statistics on the top left and top right corner of the screen. The number of the top left represents a fuel level, and the fuel level is decremented over time. The display panel on the top right represents the number of each minerals that player currently possesses, the amount of money currently possessed, and the score, which is the cumulative sum of the value of the minerals that the player has collected so far.

As seen in Figure 2, the minerals are randomly scattered around the mine.

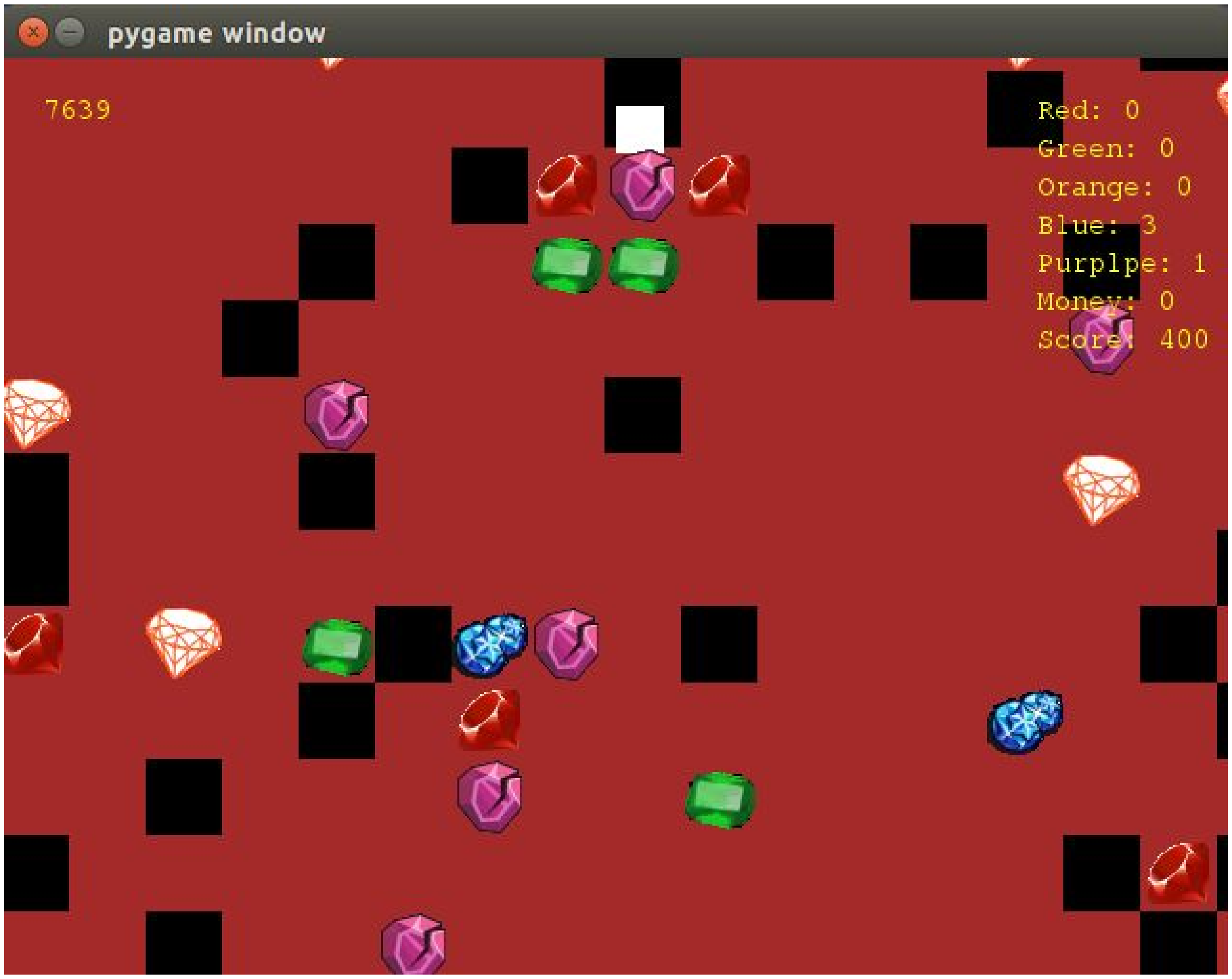


Figure 2 - Minerals are randomly scattered around the mine and there are occasional empty pockets.

The player cannot mine upward so the only way for the player to move back to aboveground is by navigating back through the tunnel that the mining vehicle had already dug. Therefore, the player has to strategically choose a way that it will navigate through the mine, so that the player can collect enough minerals while it can easily go back up to refuel before running out of fuel.

If the player runs out of the fuel, the game over message is displayed as seen in Figure 3.

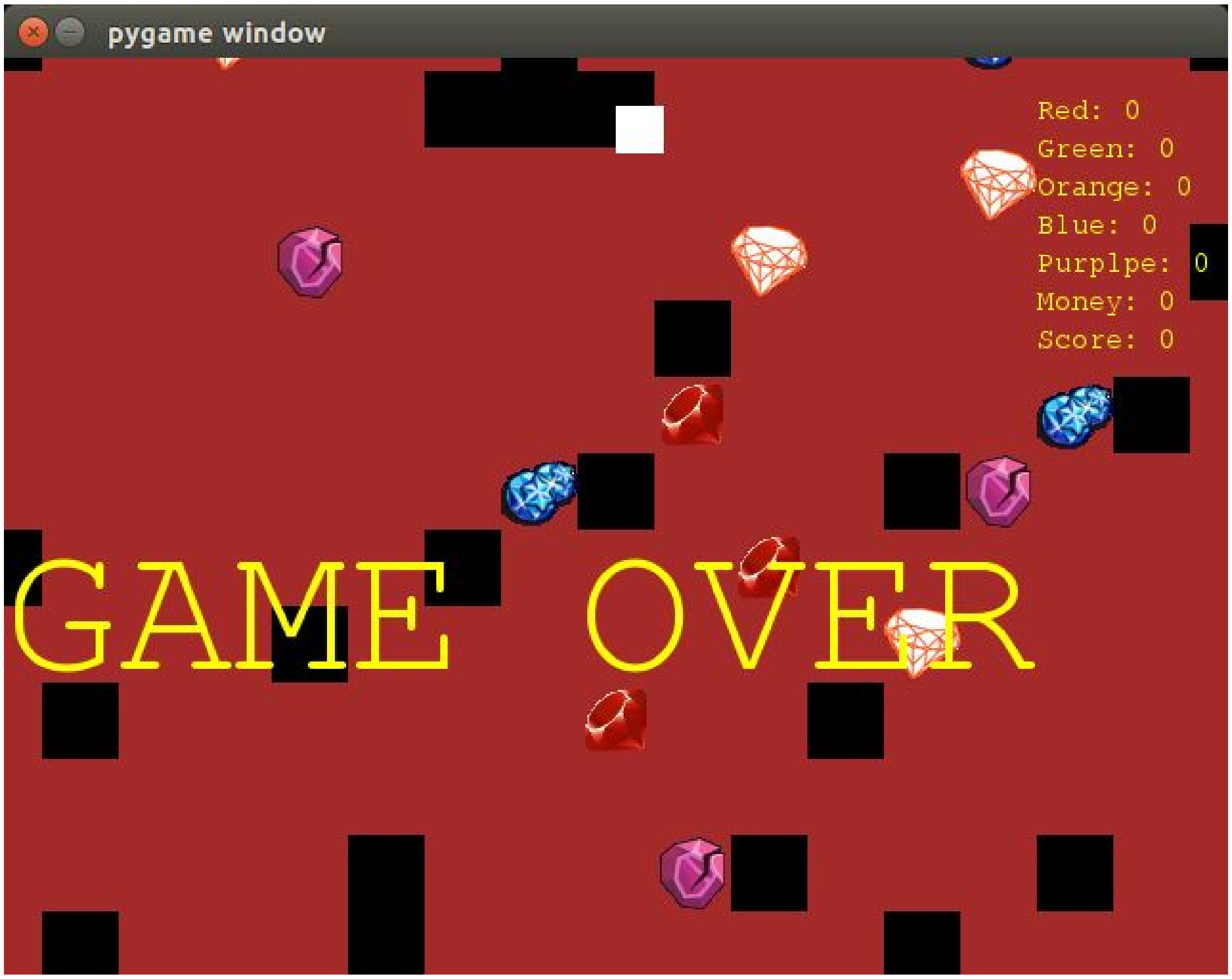


Figure 3 - Game over message

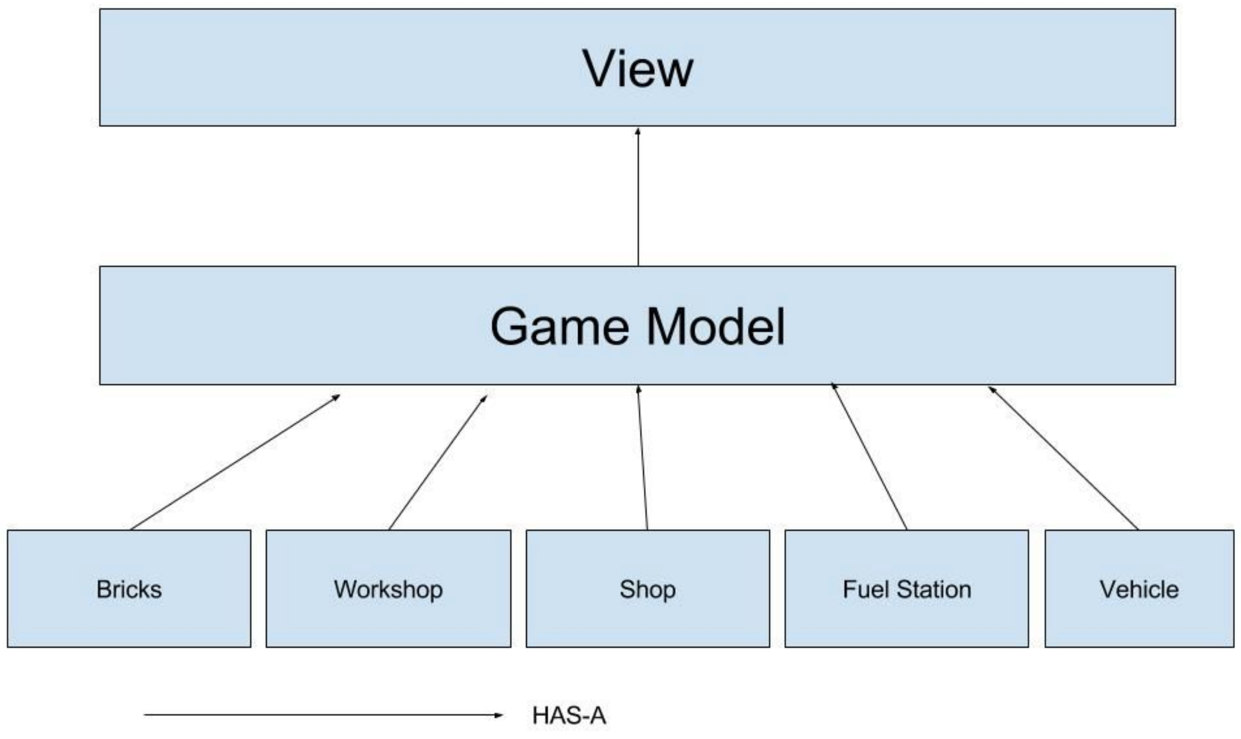
After displaying the message for a brief moment, the game terminates.

Implementation:

We separated our project into two files, one was responsible for both running the game and drawing everything on screen (motherload.py) and the other was responsible for containing all the classes that stored and kept track of the models in our game (game_model.py).

game_model contained the Brick class which defined each block in our game. A 2D array of brick objects is created at the start of the game and stored in the BrickModel class. Additionally there is a Vehicle class, Shop class, and Workshop class which all represent the vehicle, shop, and workshop in the game and are all stored in the BrickModel class. The BrickModel class has a function in it called world_enlarger that randomly generates a bigger world when the user reaches the bottom of our predefined map. This extra world gets added to the 2D array. A problem with this implementation is that at any given time the draw function displays every object stored in the BrickModel class. This is obviously a problem when the world gets too large. What we could of done was use a temporary array that stored only the parts of the world near the vehicle and displayed those objects, but due to time constraints, we didn't get to implementing this feature despite our belief that the logic behind our would be implementation was solid.

The other file, motherload.py contained a class called GameViewer which was responsible for looping through the objects in our game and checking for collision detection and then drawing the appropriate state of the game. The collision detection data procured from the Draw class was then passed along to a while loop outside of the GameViewer class where we took care of user input of keyboard controls with respect to the collision detection data. The movements from the keyboard controls in said while loop were then passed back to the GameViewer class to be displayed. Alternatively collision detection could of been its own class along with keyboard controls. Although this would've have been a much cleaner implementation it was simpler for us to place the keyboard controls and collision detection where we did (e.g., we were looping through our array of objects in the GameViewer class anyway, so why not check for collision detection too). If we hadn't been pressed on time we would've made these appropriate changes.



Reflection:

Due to a hefty pivot halfway through this project, we ended up with a lot of work to do in a short amount of time. Based on the amount of time we had to complete this project, the scope of our project was far too large. This resulted in our final product feeling half baked, poorly coded, and not as fun as we originally hoped.

In terms of lessons learned we found that it’s always better to understand your program’s logic as much as possible before coding. Yes, this seems like it should be obvious, but we weren’t so quick to pick up on this, especially when it came to collision detection

Collision detection took up the bulk of our time in our game’s development and in retrospect we really should’ve sought help from NINJAs or instructors. If we had we would’ve started using Sprites immediately. Additionally, we would’ve implemented more hitboxes -- something we only discovered we could do after the we completed our game.