Welcome to Interactive Maze

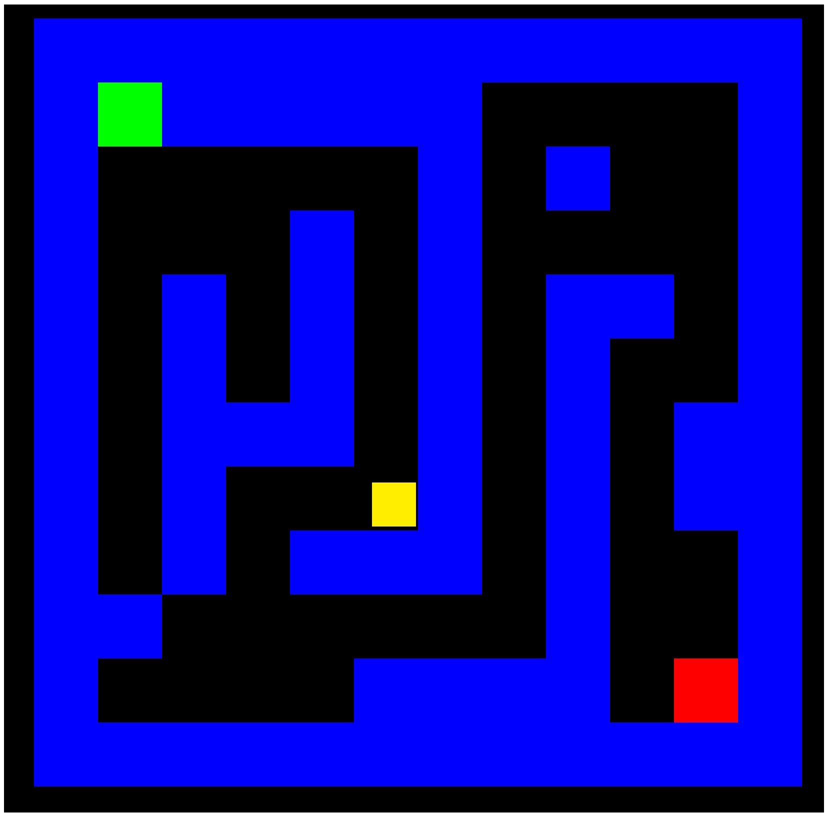
This project was made by Anisha Nakagawa (@anishan), Mackenzie Frackleton (@frackleton), and Jaime Grau Pirozzi in March, 2015 for a Software Design course at Olin College of Engineering.

Project Overview

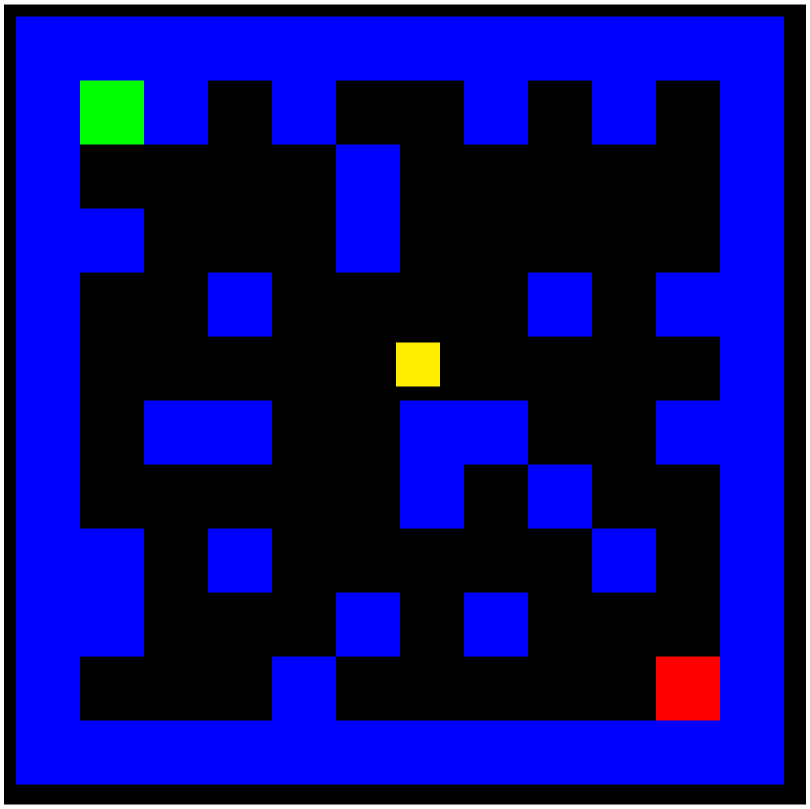
We used the pygame library to generate an interactive maze that moves around the "ball" (a fixed point) through input from the arrow keys on a keyboard. The maze uses collision detection from the Rect class built into pygame to keep consistent maze walls around the fixed point, until collision with a goal rectangle, which terminates the program. The maze randomly generates each time using a matrix [

Results

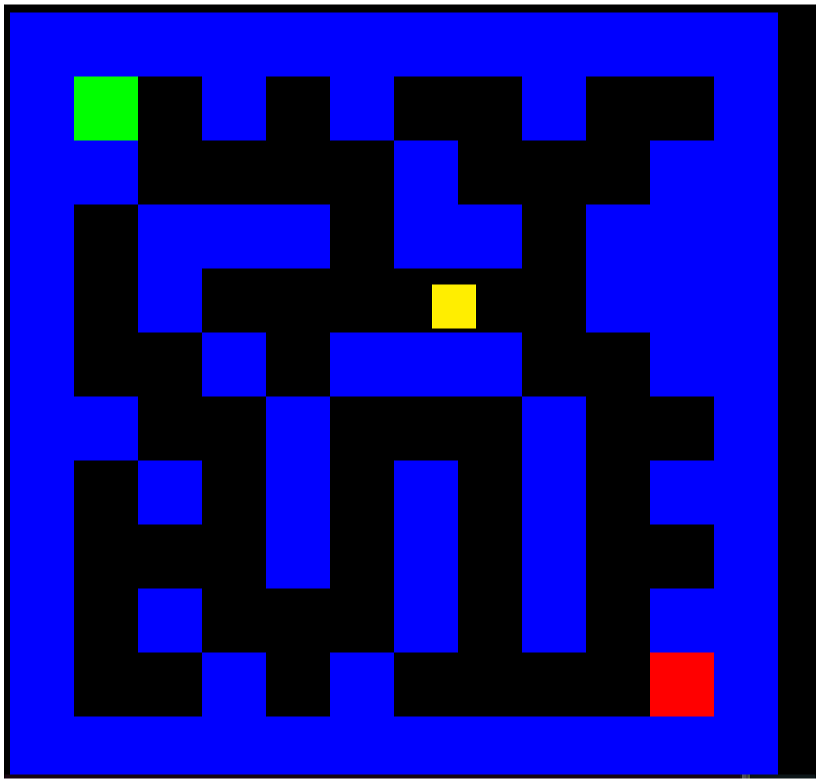
Hard Coded:



Random 1:



Random 2:



We generated three final products: one hard-coded maze and two randomly generated maze versions. The hard-coded maze final product was most useful in development of collision detection, but only differs in aesthetic from the randomly generated copies; all have the same functionality. The first randomly generated version uses a simple algorithm we generated from top to bottom that produces a solvable maze the majority of the time, with random blocks interspersed across the screen. The second randomly generating maze always creates a solvable maze with a more conventional “maze” layout, characterized most by the long, continuous walls and multiple dead-end paths. This second iteration was generated using open-source code not generated by any team member.

The maze functions primarily off collision-detection between rectangles. A fixed point in the middle of the screen collides with, but never intersects, the blue walls of the maze. This holds true especially for randomly generated walls. Collision with a red rectangle, the goal/endpoint, ends the game and prints the text “You Win!”

Users can use arrow keys to translate the maze across the screen, up or down, to move around the fixed point in the center of the screen. This results from a controller which takes the arrow keys as input and interprets arrow keys as a change in x or y velocity. Position is then updated to represent velocity added to previous position.

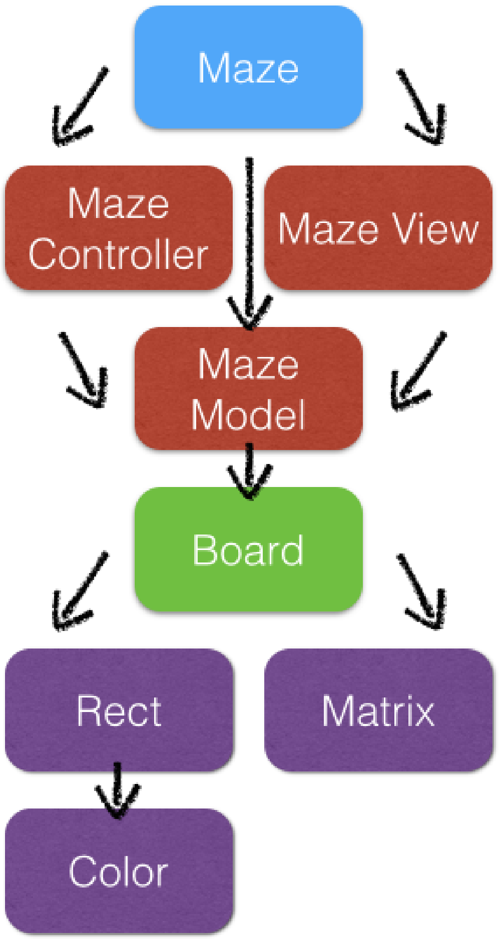
Implementation

The UML class diagram included below outlines the relationship between the different classes in the program. The main class is the Maze() class, which contains objects from the MazeView(), MazeController(), and MazeModel() classes, and continually updates each of those classes while the game is not over. The MazeModel class stores the information about the contents and position of the maze, and is updated continuously. The MazeView class draws the visible game screen by using the information from MazeModel, which is passed in as an attribute. The MazeController class also has a MazeModel object, and the MazeController updates the model based on keyboard input. The MazeModel class has a Board object, which contains all the information about building the maze board. The Board class has a Matrix object, which contains information about the structure of the maze. The Board class represents the maze as a list of Rects, which is a type of pygame object that can be drawn to a screen. The Rects all have Color attributes.

The structure of the maze is in the Matrix class, stored in a two dimensional array with 1s and 0s, where the value at each index corresponds to a free or occupied space in the maze grid. Each value is indexed by the row and column of its location. In each maze, there is also one starting point indicated by a ‘2,’ and the end indicated by a ‘3.’ In the board class, this matrix is translated into a list of Rect objects, which are the pygame built in objects for drawing rectangles. The coordinates of the individual Rects are determined by the relative location of the squares given by index positions in the matrix, and the location of the overall board (stored as an attribute in the Board class). Each Rect has a color attribute based on whether it is free or occupied space. The board of the maze moves around in the screen, and the position of the board is given by the absolute position of the top left pixel. This means that the board can move by as little as one pixel in any direction. We could have instead made the entire background a grid and have the board move by a discrete block each time, but the structure that we have allows for more fluid movement.

We are including three files in our submission for this project, where the differences between the files is how the maze is generated. In the maze_hard_coded.py file, the maze is generated from a matrix that was hard-coded into the program. This file shows the same maze each time. The maze_random1.py file has a somewhat randomly generating maze that we programmed. This algorithm builds the maze by randomly combining ‘blocks’ of the maze from a pre-defined set of blocks. Each block is a two-by-two grid, where one of the positions is an occupied ‘wall.’ This will almost always be solvable with multiple solutions. While this version does not produce a perfect random maze, it was a good exercise in coding to have a different maze for each run. The third file, maze_random2.py, uses open source code that randomly generates a maze with only one correct path to the end. We included all of these versions to highlight the different ways that the maze can be generated, and each has its own benefits.

UML Class Diagram:



Reflection

From a process point of view, it worked well to build the code starting with a basic scaffolding. We began with a barebones class structure, added in additional classes we realized would be necessary, then outlined functions that each class would require. This started without any actual workable code, and allowed us to organize in our larger team of 3. We could have improved streamlining the code near the beginning, as this scaffolding left us more prone to outlining unnecessary classes. We also could definitely improved working in parallel; the three of us started out working on the same issues at the same time. This led to confusion and miscommunications more often than not. Over time, the team did correct and each member diverted attention to a separate component of code.

As time progressed, we found the project to be well scoped and continued to add new components. We did realize rotating the maze around a fixed point would be more complicated than we first thought, but this aspect of complication was dropped and replaced with random generation. We often found that planning ahead was often ineffective due to our unfamiliarity with the pygame. It would have been more helpful to set periodic deliverables for ourselves from the beginning, instead of setting these goals later on in the project. To unit test, we would isolate specific components of the program and assess the output. This eventually evolved into testing the controls at each stage. Saving each iteration was definitely crucial to debugging. Going forward, we will use this experience to share the workload of coding assignments more evenly, and work in a more efficient, segmented manner.