Thought Process: Avoidance Ball

Introduction

I decided to take another look at a different game from Battleship – focused on using PyGame, a python module with established ways to create external windows and basic 2D graphics. Further, I wanted to try making a game on my own – something simple to play yet challenging to code.

The idea for the game is simple: the user controls a blue ball using a keyboard that moves the ball through a grid while red and green squares move from side to side. The score increases by 1 when the ball hits a green square; it decreases by 1 when the ball hits a red square. Further, after hitting 3 red squares, the game ends.

Sounds simple, right? As is often the case, it proved to be far more complex and far more difficult to debug while illustrating some of the limitations of data structures and types along the way.

This time, there is no computer solving, but many of the same elements – such as calculating the random path for the squares to follow for every frame – remain, especially when it comes to planning.

This project is just as complicated as the previous two, although I did have to go back and make many efforts to try and consolidate the code the best I could. Most of what follows won't make a lot of sense without an understanding of the code, so I'd recommend at the very minimum quickly skimming over the code – and if you have python and PyGame, try playing it[1]!

I have split this document into a few sections which mirror my own experience planning, designing, and writing the project. First, I will talk about PyGame in general and the overall logic of this game, then the board class, movement of the ball, the colorSquare class, the functions, and finally the program flow.

PyGame and the Game Logic

First, a quick disclaimer: what follows is my understanding of PyGame. I only focused on things that I thought would be useful in making the game, which is why some perfectly valid things are not here. The PyGame website has much more thorough documentation for a much wider variety of applications.

Every game, or more generally video, has something called a "frame". The movement of these frames very quickly, one after the other, is processed by our eyes as continuous motion. For movies, generally 24 fps is used, while for many games, either 30 or 60 fps is standard, with some users getting even higher frame rates for specific games considering the technical specifications and network speeds of their device.

PyGame creates an external window (treated as an object[2]) outside of the terminal that is given a name, width and height when initialized. Then, there is generally a main loop that consists of all of the

[1] All of my project files are on my Github and can be found here: https://github.com/aniruthn/Projects
[2] This program is object-oriented; as such, some prior background on object-oriented programming is helpful. The previous program "Battleship" uses this. An object is an item that has "instances" with each of those instances

various activities being done when playing the game – such as checking if the blue ball has come into contact with a square, whether the square is done along its path, movement of the ball, etc. – before filling the window with black (erasing the last frame), adding certain things to the window at specified locations using the drawing and "blit" methods of the window to do so. Finally, the display is updated, such that all these changes appear at once – creating the "frame".

Every complete iteration of this main loop is a frame, and the continuous iterations create the frames for the user to use. At the very end of the loop, the frame rate is restricted to 60 frames per second using the tick method on the clock object.

Something important to note is that the way Pygame denotes coordinate points is a bit peculiar. It uses a x and y coordinate plane, although the upper left corner is (0,0) and the y value increases going downward while the x value increases going to the right.

The very broad game logic has already been mentioned in the introduction, but it's worth mentioning again. The user controls a blue ball's position in a grid using the "WASD" keys (common in many video games due to ease of movement with the left hand). The objective is to move the ball such that it meets as many green squares as possible (adding 1 point to the score each time), while avoiding red squares (subtracting 1 point upon hitting the ball). Once 3 red squares are hit, the game is over.

There are more subtasks to ensure that all of this happens. For one, the window would have to create a grid for the ball to move, store the location of that ball at all times, allow for movement of that ball, generate the squares, animate their movement accurately, determine interactions between the ball and a square at every frame, and adjust the score and obstacleCount variables as needed.

The board Class

The user interacts with a ball that moves on a grid – the purpose of having the board class. It does not interact with the squares or affect their movement in anyway, but it does always store the position of the ball and it contains a method for drawing the board to the screen.

The board object is first initialized with a variety of variables that are then used to create attributes for the object. It takes in the rows and columns, creating a 2D array that has a value of 0 in every position. It also adds the color and thickness as attributes (to be used when drawing the board).

Then, there is a method to get the ball's location which iterates through every value in the array to find where the ball is. In the 2D array, there is a 1 at a certain intersection of the rows and columns which is the location of the ball. Once it finds the 1, it then returns the location of where the ball is being drawn, by giving the x and y of the coordinates as well as the radius of the ball.

Finally, there is another method to draw the board. It draws directly to the window, drawing a rectangle at every position where there is a grid, spacing out as needed (there are 200 pixel margins on the sides of the grid and each row and column is 100 pixels wide). And, if it detects that the value at that position is a 1, it draws the ball, using the center and the radius.

having both attributes and methods. For example, if I have an object of a person, I could create instances of multiple people, while each one of those people have different attributes – like hair color or height.

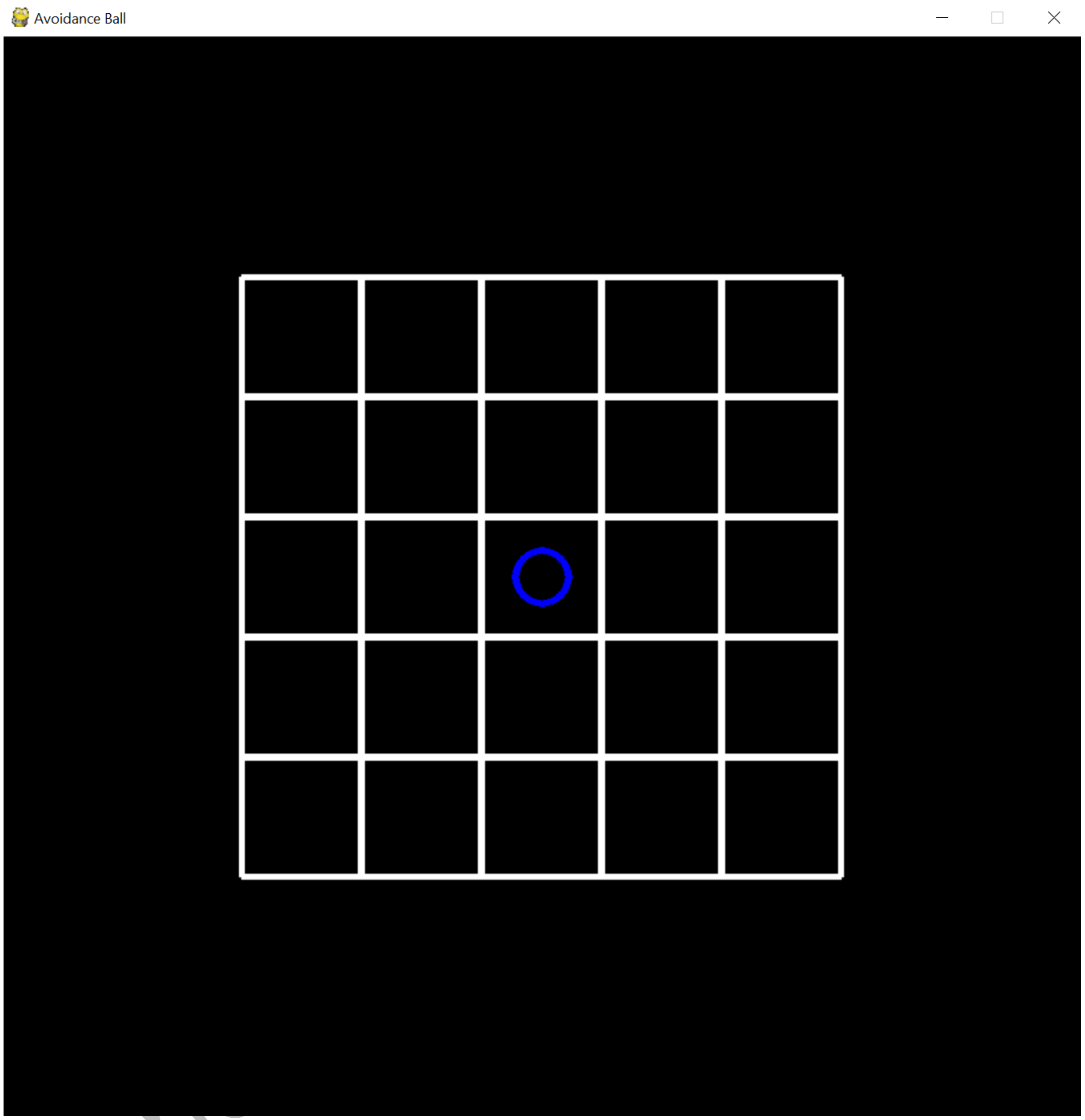

Figure 3-1

The board and the blue ball are drawn to the window.

An important thing to note is the syntax of PyGame, for it affects the math later. To draw a rectangle, PyGame requires the window, the color, and a tuple. This tuple has the upper left point of the rectangle's coordinates, followed by the width and height. There is an optional value at the end for thickness – if omitted, it fills in the square, as is the case with the green and red squares. However, for the grid to have an effect, the thickness is required. To draw a circle, it requires the window, the color, the center of the ball coordinates, the radius, and then the optional thickness value.

Moving the Ball

The movement of the ball is relatively simple as PyGame makes it easy to register key input. Effectively, when a certain key is pressed, that triggers a conditional statement that then moves the ball. A list called “keys” is created in line 191 that takes all of the keys being pressed, and then those keys are then accessed in 4 conditional statements (one for each key) to determine how the ball moves.

The modulus operator is used to cycle the ball around. It adds one to move to the right and down (since the array indices increase to the right and down), but to move to the left and up, it first adds the number of rows or columns and then subtracts one. This is to prevent a negative number (since the arrays are zero-indexed) which could hamper the modulus operator. It has no effect on the value however, since the modulus is only looking at the remainder (for example, 3 % 2 is 1, and 5 % 2 is also 1). This is illustrated as well in the graphic below.

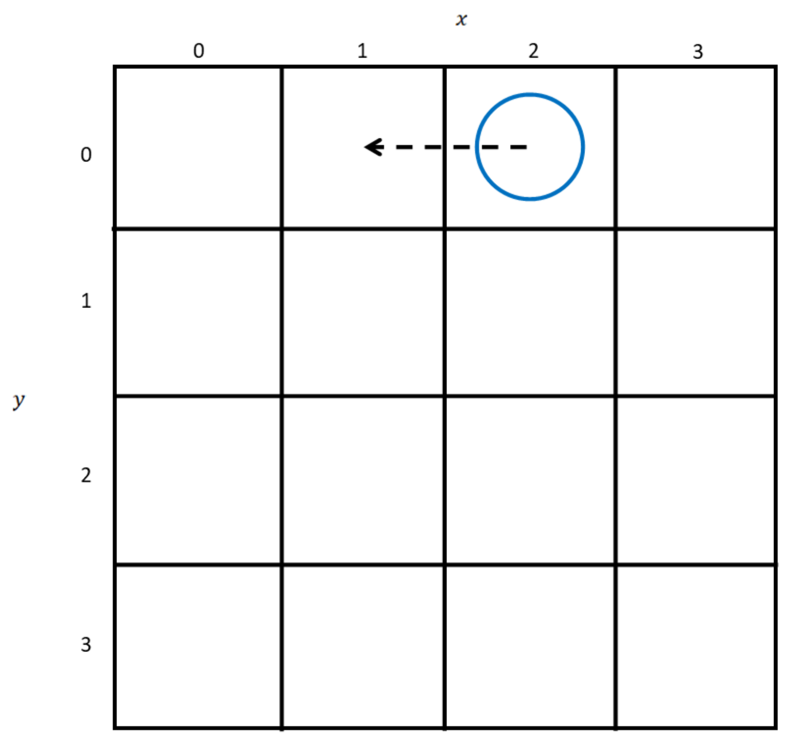

Figure 4-1

In Figure 4-1, the user presses the key “A”, which means that the ball must move to the left in the grid. In this case, only the x value would be changed, first the placeholder value, and then the grid itself. While 1 could be subtracted in this case, consider what would happen if the ball were at (0,0). It would then become (-1,0), so to prevent that from happening, it adds the number of columns so that it becomes (2,0). In this case, the modulus operator is used as well. The ball is currently at (2,0), and since there are 3 columns, it becomes ((2 – 1 + 3) % 3, 0), or (4 % 3, 0) which is (1, 0). It may seem a bit convoluted, but it ensures that the edge case when the ball has one of the coordinates be (0,0) is accounted for to allow the ball to quickly swing to the other side of the grid.

When the keys are used to move the ball, three things happen. First, either the y value or the x value variable gets updated, second, the moveLoop is set to 1, and third, the previous position of the ball is reset.

There is a y value and a x value variable to act as placeholder values to avoid any issues with changing the grid[3]. Instead, they are adjusted and then switched back to the location of the ball after the conditionals regarding the ball's movement. This is where the modulus operator is used.

The moveLoop variable is used to require 5 frames for each move. Effectively, for the ball to move, the moveLoop must be at 0, which is where it is at initially. Whenever the ball registers movement, the moveLoop is then set to 1, which is incremented for every iteration of the entire loop (every frame) until it gets higher than 5, where it is then reset and allows movement once more.

The last thing that happens in the conditional statements is that the previous position of the ball is reset by accessing the original index of the ball and setting that value to 0. This removes the ball from the grid before it is put in again by updating the board's grid with the new location, setting the new location to have a value of 1 (so that when the grid is drawn, it draws the ball at that location).

Up to this point, there is now a grid and a ball being drawn on that grid that the user can move using the keys. Now, the movement of the green and red squares as well as the actual game logic of the collisions between them begins.

The colorSquare Class

The movement of the squares across the board requires many considerations. For one, it must have some sort of a path to follow that is randomly generated – easily solved by providing a list of 4 values that can be used to access the start and end points. Then, it has to figure out the individual movement for every frame to create the animation of the square moving while tracking the progress so that when it's done, the square is replaced with a new one with a different path across the screen. Additionally, it must check for collisions with a circle which requires additional math.

The purpose of having a colorSquare class is to set all this information as attributes for each square. By setting each square as a separate object, it allows specific information to be used for each object while re-using the code performing operations on every object, simplifying the code significantly. The class does not do all those tasks – in fact, it merely calculates some of the values that will be used later by various functions and statements which accomplish that list of tasks.

When initializing the square, it takes in three parameters – the color, the path, and the velocity. The velocity is how many total pixels along the path given the square should move, which will then be split up into component speeds[4]. The color is then set as an attribute to be used when drawing the square, and the path is used to generate the component speeds, the initial start position, and the expected frames.

[3] The inclusion of these placeholder variables is most likely not mandatory but simplifies the coding process and offers only marginal efficiency improvements upon removal.

[4] Technically, these are component vectors since they have direction given by the sign of the value (whether it is positive or negative) but I found it easier to type speedX instead of velocityX so I just kept it.

Most of the math of this function is around generating the component speeds. Initially, the first two values in the path are set to be the square's x and y coordinates which will be used when drawing the square (it is the upper left corner that is being drawn). Then, the change in the Y and the change in the X is determined by finding the difference in the coordinates. The slope is then found before the exact angle to be used is derived by using arctan, with an edge case for when change is 0 to prevent an error. Finally, the theta is then used in the components, before attaching signs. The signs have to be manually added at the end because of the way that the graph is set up – moving to the right and down is positive in PyGame, so arctangent cannot be used on its own to determine the signs of the values as it would be able to regularly. Finally, the frames are determined by taking the total change in the Y and dividing by the speed of Y (it would be the same number of frames if the change and speed of X or the total path was used).

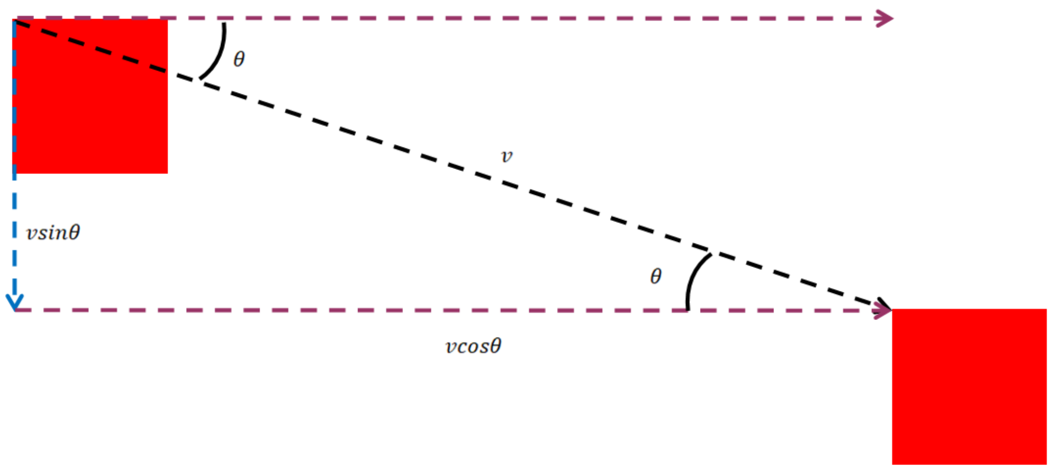

Figure 5-1

Figure 5-1 helps illustrate what is happening from a physics perspective. The red square is going from the upper left to the lower right, so it is an increase in the x and a decrease in the y coordinates along the way. As such, it calculates the θ as needed through the arctangent of the change in y over the change in x and then it distributes the total velocity along the path by multiplying with trigonometric functions as necessary[5] before giving the vertical speed a negative value and the horizontal speed a positive one so the square moves in the right direction. This procedure, while a bit tenuous, ensures that everything is drawn as needed on every frame.

The colorSquare class also has a method to draw the object using the color and the exact position of the square and it operates in the same syntax as drawing all other rectangles.

[5] This requires a conceptual understanding of basic trigonometry. It may be helpful to draw out the diagram and try running some calculations with a sample path – that is what I did to ensure that the math conceptually made sense.

Functions

This section will focus on the three methods at the very beginning of the avoidanceBall program – where everything is imported and used to create the game. These functions are what perform most of the calculations and generation that is used for the color squares.

The first method is the randomPath method. Effectively, it determines where the square will be moving either horizontally or vertically, and then it determines which direction along that path it will be moving. If the square is moving horizontally, it randomly generates the y coordinates, and if the square is moving vertically, it randomly generates the x coordinates.

This procedure ensures that a square will not be generated in the middle of the screen and then removed, standardizing motion by requiring that it move across the entire screen no matter what. The values used are meant to reflect that all coordinate points only apply to the upper left corner of the square. Therefore, the starting positions are shifted to be -50 and the ending positions are exactly on the screen's height and width to allow the squares to fade in to the screen and fade out (the width and height of the square is 50 pixels, so having the starting be -50 ensures the right and bottom edges are out of the screen at the beginning of the path).

The second method is the how the squares are generated. It calls the previous method to generate a random path to assign to the square, uses a set velocity, and then transfers the color parameter to the object it returns. This method merely simplifies the code a bit in the actual program flow.

The third method determines collision between the blue ball and any one square. The math here is a bit intense in terms of the logic and cases considered, but I have done my best to try and explain it below – it is interesting how it works as a solution. There is also a handy graphic at the end of the explanation – Figure 6-1.

Before going through the algorithm in terms of pseudocode, it is worth considering why this even poses a challenge at all. Consider the case of having two squares (or rectangles) and the task of having to determine if they intersect[6]. Fairly simple, right? For them to intersect, at least one corner of one box will be inside the other square and vice versa. Each box has clearly defined lines, so this procedure requires a few checks and inequalities but is not too complicated conceptually.

Now consider a circle. Circles have no lines (technically, they have an infinite number of lines, but that proves useless in this case). To determine if a circle intersects with a square, it is a bit more difficult – since the distance between the circle's center must be considered. Yet, it would be highly inefficient and slow to check all 4 points to determine the distance to the center every frame for every square, comparing it to the radius. To make matters worse, this may not even work in some cases. If the box is directly below the circle, the distance from the entire edge to the center of the circle may be less than the distance from the center to either of the points that define the edge.

The algorithm itself considers 3 edge cases and then checks all 4 edges. It relies on one postulate: if a circle and a box overlap, the distance from the circle's center and at least one of the edges of the square must be less than or equal to the circle's radius.

[6] Intersect, when used here, is inclusive of both touching and overlapping. The function is called overlap, but the usage of less than *or equal to* the radius includes this (somewhat of a misnomer).

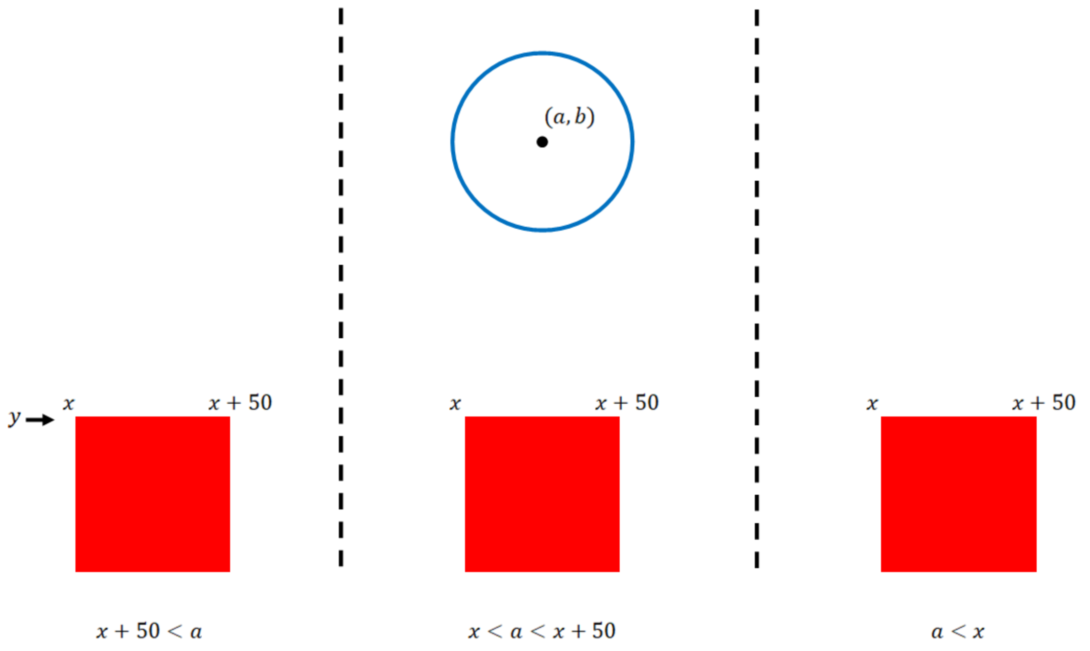

Figure 6-1

Figure 6-1 illustrates this principle, displaying the 3 edge cases for the top edge (a similar procedure is used for the other edges, so I will explain the algorithm solely in the context of the top edge). If the square is in the leftmost position, then only the upper right corner needs to be checks in relation to the circle's center at (a,b). If it is somewhere in the middle, then it checks the distance between b and y (the vertical height from the edge to the center of the circle). And if it is to the far right, then it checks only the upper left corner. In all three cases, only one calculation is done to determine the shortest distance from the square's top edge to the center of the circle.

The same procedure happens for the other edges. For the left edge, the x value remains constant (for the top edge, the y value remains constant, which is why it has an arrow to denote it remains consistent through all three edge cases). Then, it would be checked with both y and y + 50 to determine the position relative to b and taking the distance as needed.

Program Flow

The program itself utilizes these various classes, methods, and functions to run. The actual program is designed to be flexible, accounting for different numbers of rows and columns. This utilizes user input to determine the rows and columns as well as the level of difficulty that the user prefers to adjust the number of red and green squares as a result. It then creates a few of the initial objects, such as empty list for the squares, initialized counters, calculating screen width and height with 200 pixel margins on each side and 100 pixel width rows and columns, creating the font, a clock object, and more. The main thing to note is that when the game board is created, the ball is given an initial position in the upper left, for it is guaranteed to always exist as a position (for the game to be playable with a grid in the first place).

Based on the difficulty, it then calculates the number of red and green squares, appending each to the colorSquares list which will be used to access and draw every square, every frame. With this, the loading screen appears for 2 seconds before the actual main loop begins.

The main while loop runs continually, with every iteration serving as one frame of the game. It begins with an exit method so clicking the X in the upper right corner closes the window by ending the loop. Then, the ball movement statements are utilized – as previously discussed. It takes in key input, then fills the window with black (clearing the previous frame) before drawing the gameboard grid and the ball.

The next large section of the code is another while loop that iterates through every object in the colorSquares list to determine if there is a collision for that square, if that square is done in its path, draws it, and then adjusts the position by the speed for the next frame.

One important thing to note before delving into the specifics of this part is why a while loop and not a for-each loop is used. When there is a collision or an overlap, then the square is immediately replaced. This procedure is not recommended with a for-each loop, since a for-each loop is best utilized for simply accessing the contents of a list, not modifying the list itself.

The frame counter is relatively simple, as for each square, the frame count is increased by 1. It then checks that with the expected number of frames from the object's initialization. If they are equal, it means the square is at the end of its path, and so it goes to that index in the colorSquares list to generate a new square with the same color. While positions could be used, sometimes the position is off by a very, very marginal amount to a low order of magnitude which prevented it from detecting that it was at the end – so the frame count method works a bit better as it is guaranteed to equal the number of frames, since it is an integer number.

A similar process happens when checking for collision. By utilizing the overlap function and then determining the color of the square, it adjusts the score and obstacle count as needed before replacing the square (if it wasn't replaced, then the square would move through the ball, messing up the score and obstacle counters to represent the pixels of contact, not the number of squares in contact). Then, the square is drawn, and the square's position is incremented by the speed as the process repeats for the next square in the list.

The final section of the code adds text to track the score and the obstacles hit before updating the display window to synthesize everything into one frame. The order in which these are done ensures that the score and obstacles hit is always at the very top above the squares and that the squares are above both the grid and the ball. It also checks the obstacle count to determine if the game needs to be ended, in which case it simply tells the user that the game has ended for 2.5 seconds and then ending the loop. The final statement limits the frames per second to 60 using the clock object initialized before the main loop. Finally, whenever the run loop is broken, the game ends.

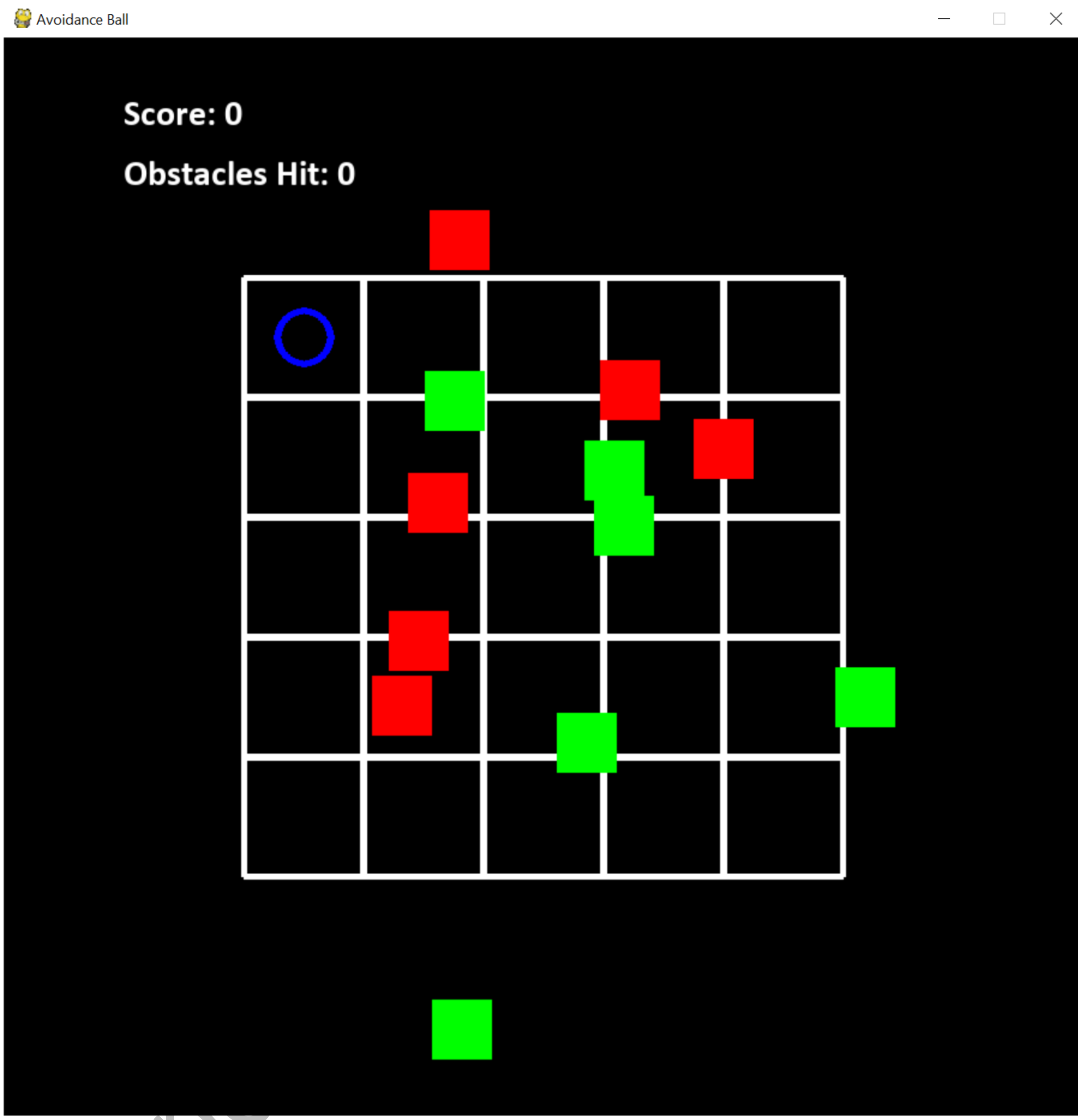

Figure 7-1

The result for the user to interact with the game

Figure 7-1 shows how the gameplay looks for the user. The squares are randomly generated and moving, and the user must move the blue ball to try and hit as many green squares as possible to increase the score while avoiding the red ones. In this case, the game just started, which is why the score and obstacles hit both have a value of 0 with the ball being in the default position in the upper left (assigned after the board’s initialization).

Areas for Improvement

As with all programs, and especially with games, there are still ways that this could be further refined.

1. The 70 lines of code in the overlap function could be optimized and shortened to a much greater extent.
2. After writing the full file, I realized that I switched the rows and columns. The reason for this was that in testing I had set them both to be 3, which led me to mix them up quite a few times. While it does respond to user input correctly, as mentioned in the program flow (specifically on lines 118 and 119), it is a bit confusing sometimes. It was not worth going back and changing in this case, but for future developments, this is something to consider and plan of time before starting the project.
3. Most of the games that I play or that I have seen have more in-depth and detailed loading screens. One idea that I had was to create an interactive menu using the arrow keys to select the level of difficulty, but I ended up deciding to keep the selection in the terminal instead because it simply wasn’t worth it.

For now, as usual, I will leave these to another time.