Programming a Classic Arcade Game with Ruby

First Draft

Andrew Wheeler

September 13, 2015

Contents

Foreward

This enrichment activity will be providing an introduction to video game programming using the Ruby language. It is written with grade 9 and 10 students in mind, although this enrichment activity would be appropriate for any students who have some experience with the Ruby language. Specifically, they should be comfortable with loops, classes, and functions. This project is split into multiple chapters, each of which focuses on a specific component needed in the creation of the classic arcade game Breakout. The first section of each chapter provides a description of the objectives for the given chapter, which will generally be a specific component of the Breakout game, along with explanations of the problem-solving and programming strategies needed to accomplish these objectives. Activities are then provided that are aimed at reinforcing the concepts discussed and provide the student with the required knowledge reinforcement to accomplish the given objectives. The final section of each chapter will then require the student to actually implement the required game component into their Breakout game and it will also include teacher notes and a student checklist.

1 Introduction

In order to start to build Breakout we will obviously need to have an understanding of its general game play. Breakout's game window's set up is shown below. We have coloured rectangles in the upper half of the window, which are referred to as bricks and are in fixed positions. The white rectangle at the bottom of the screen is the paddle and the small white square is the ball. The paddle is in a fixed vertical position but is able to move horizontally with user input from the left and right arrow keys, while the ball moves linearly throughout the window bouncing off the walls, bricks and paddle. One major key component to the game play is that when the ball bounces off a brick, that brick must disappear. This game play sequence continues until one of two things occurs: either the ball gets past the player's paddle and hits the bottom of the screen or all of the bricks have been cleared. There are also other game play aspects that are key to the users gaming experience such as points, lives and increasing difficulty level. However, we are going to wait until we get our basic game sequence working before we start to worry about adding these important pieces. To view a simple version of the Breakout game running you can view the following video.

Figure 1.1: A screenshot of the basic layout of the game components in Breakout.

This would also be a good opportunity to discuss the Gosu library that we will be using. Gosu is a 2D game development framework for the Ruby programming language. It provides us with some basic building blocks that are needed to build video games. These include a window with a main game loop, the ability to add basic 2D graphics and text, and it gives us basic methods for receiving input from the users keyboard or mouse. Thanks to Gosu and the building blocks that it provides, we are able to focus our energy on adding our game components and logic. The specifics of how to implement Gosu will be covered in each individual section.

2 Game Window

In terms of the user experience, the game window is the computer window where our game will actually be played. Gosu provides us with a main window class that is really the foundation for our entire game. This Gosu Window class manages the creation (drawing) of all the core components as well as providing the game loop, which allows your game to be continually updated with new information. However, we want the ability to edit and add to the Gosu Window class to make it specific to our game and this is where the power of inheritance comes into play. We will create our own `GameWindow` class, which inherits from the Gosu Window class. Our `GameWindow` class will then have access to the `initialize()`, `update()` and `draw()` methods found in the Gosu Window class. It is the combination of these methods that make up the main game loop, which is described in the figure below and is quite important to understand as it forms the foundation of our game.

2.1 Game Loop

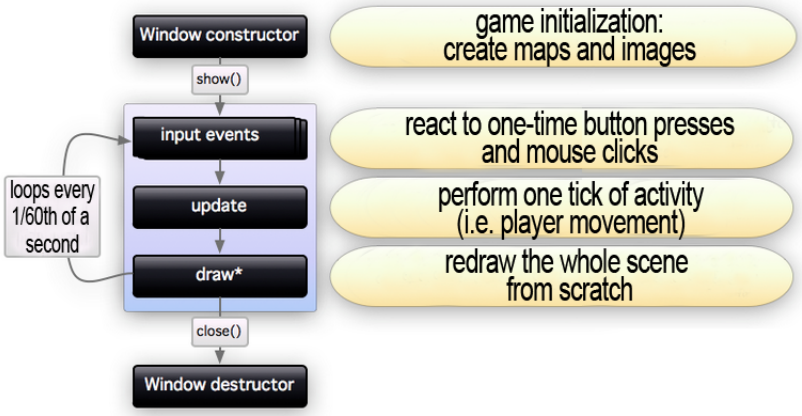

Figure 2.1: Gosu's game loop.

Once a new instance of the `GameWindow` is created it first runs the `initialize()` method, which creates the window along with any required images. In terms of our game, we will want to have the paddle, ball and all of the bricks created in our `initialize()` method. Now once the `show()` method is called on our newly created `GameWindow` instance it will start the game loop which goes through the following steps at a rate of 60 times per second;

i Check for any input events from the user, such as pressing any buttons or using the mouse.

ii Update the game components. Did the user click the left arrow key? If so, move the

paddle left a set number of units. Did the ball hit a wall? If it did, then make it change direction, otherwise the ball should continue in the same direction.

iii We now need to redraw the whole scene with our updated information. For example, if the ball hits the brick then we need to make sure that that brick is not drawn and that the ball is drawn in its new location as it would have bounced off the brick.

2.2 Setting Up The Game Window

We will first look at the `initialize()` method, which is responsible for constructing the game window with all the required game components. It will not actually draw the components onto the screen, as the `draw()` method is responsible for that, but it simply takes in all of the information for the components and stores them in this instance of the game window. If we look at Figure 2.2 below we can see the basic code for getting the game window set up. Remember that our `GameWindow` class inherits from Gosu's Window class, and therefore the super (line 5) is passing its arguments into Gosu's Window class `initialize()` method which takes the following arguments of width, height, and fullscreen. The width and height are clearly the dimensions of the window, while fullscreen is a boolean value that determines whether to create a full-screen window. Additionally we set a caption which you will see creates a title for the window.

```
1  require 'gosu'
2  include Gosu
3
4  class GameWindow < Gosu::Window
5    def initialize
6      super(800,600,false)
7      self.caption = "Setting Up Game Window"
8    end
9
10   def update
11   end
12
13   def draw
14   end
15 end
16
17 window = {\tt GameWindow}.new
18 window.show
```

Figure 2.2: The basic code required to get the game window setup and running.

2.2.1 Activities

1. Type the code in Figure 2.2 into your editor, save the file and run the program. If you do not see a black square window pop on your screen then carefully look over your code and ensure you have not mistyped anything. Make sure that you save this code as we will be appending to it as we move through the project. [Solution]

2. Create a game window that is 300 x 500 pixels and not full-screen. Does it matter which order you input the arguments? Explain. [Solution]

3. Set the parameter for full screen equal to true for the game window in question 2. What happens? Does the game window get set to the pixel dimensions of your screen? [Solution]

2.3 The Game Loop: Draw

We are going to look at the `draw()` method in the `GameWindow` class, which remember is part of the game loop. To experiment with the `draw()` method we need to actually draw something. This is where Gosu comes in handy again, as it has built in methods for drawing simple images. In this case we will use the `from_text()` method from Gosu's Image class. This is a class method and takes in the parameters of window, text, font_name, and font_height. We can then use this method to create an image made up of text that we can appropriately assign to the variable @text.

```
4  def initialize
5      super(300,300,false)
6      self.caption = "Setting Up Game Window"
7      @text = Gosu::Image.from_text(self, 'Testing the Draw Method!',
8          Gosu.default_font_name, 40)
9  end
```

Figure 2.3: The `initialize()` method in the game window that is constructing an image from text.

If we add this code to our `initialize()` method in our `GameWindow` class, as shown in Figure 2.3, we are constructing the @text image. Now if we run this new code that has the @text image, what do we get? You might expect to see a line of text on the screen that says "Testing the Draw Method!" but that is not the case. It only gives us the same black window we had before. We need to remember that the `initialize()` method only constructs the @text image and does not actually draw it onto the screen. Our instance of the `GameWindow` now contains the information required to draw the @text image, we just need to tell it do so and let it know where exactly on the screen we want it to be drawn. This is where the game loops `draw()` method comes into play.

```
14 def draw
15     @text.draw(10,10,0)
16 end
```

Figure 2.4: The `draw()` method in the game window, which is used to draw images to the screen.

Now if we update our `draw()` method, as shown in Figure 2.4, and run our program again we will see that the @text image has been drawn on the screen. As you may have guessed, the parameters for the instance method on the Gosu Image object are its (x,y,z) coordinates. But if Gosu is for building 2D games, why is there a z-value?

2.3.1 Activities

1. Using our "Testing the Draw Method!" image, experiment with drawing at various coordinates in our game window. You may have noticed that only the x and y values control the location of the image and the z value seems to do nothing.
   a) We will get to explaining the z-value soon, but for now can you draw a coordinate axes for the game window? [Solution]
   b) Where is the origin (0,0) on the game window? [Solution]
2. We have seen how the `from_text()` method works. Now you are going to learn how to include an image as a PNG file.
   a) Select a PNG file of your choice and draw it anywhere on our window. [Solution]
   b) Now use the `draw()` methods x_factor and y_factor parameters to scale the image. Try scaling it to be both larger and smaller. [Solution]
   c) Scale your PNG image to take up about half of the window. Now try to place the image in the center of the window by passing in coordinates for the window center, which should be (400, 300) unless you have changed the window size. Does the image appear to be in the centre of the window? Explain why not. [Solution]
3. Draw both our "Testing the Draw Method!" image and your PNG onto the window and use the same coordinates.
   a) Which image is on top? Now in your `draw()` method, flip the order in which you drew the two images. What happened now? Explain. [Solution]
   b) Now change the value of the z-value for the image that is currently on the bottom to a number greater than the z-value for the image on the top. What happens? Can you explain how the z-value works? [Solution]

2.4 The Game Loop: Update

We now know how to draw components onto the screen, but we do not know how to actually make them interact with each other or with the user. This is where the `update()` method comes into play. Before we get ahead of ourselves, we will just show that the game loop is in fact running. Update your `initialize()` and `update()` methods as shown in Figure 2.5 and before running the program, try to figure out what will happen (hint: be sure to watch the terminal window and not just the game window).

Now that we know that the game loop is in fact working we can start to use it to draw and interact with images in our game window. We will create a program (Figure 2.6) that will output a text image that shows the current coordinates of the image and using the keypad, allows the user to move the image, with the coordinates of the image updating in real-time.

```
4 class GameWindow < Gosu::Window
5   def initialize
6     super(800,600,false)
```

```
7       self.caption = "Testing Game Loop"
8       @counter = 0
9     end
10
11    def update
12      @counter += 1
13      puts @counter
14    end
15  end
```

Figure 2.5: This program that shows that the update loop is working and how quickly it runs.

If we break down the program in Figure 2.6 we can see many similarities to earlier examples. We are again initializing a game window, but this time we are setting @x and @y instance variables that will set the initial values for the image to be drawn. Now once the `show()` method is called the game loop begins and our `update()` method checks to see if any of the arrow keys have been pressed by using Gosu's `button_down?()` method. Pressing any of the buttons results in the expected change in our (x,y) coordinates for the image. After the `update()` method is complete the `draw()` method is run, which uses Gosu's `from_text()` method to create our @message image, which will show the x and y coordinates (remember these coordinates will have been updated if any arrow buttons were pressed). Once the image is constructed it is then drawn on the screen calling the `draw()` instance method on @message. That is one cycle of the game loop, which is then run again and again at a rate of 60x per second. This continues until the window is closed or the program is ended. This a good frame rate for our purposes, but it is possible to change update interval. Along with the width, height, and fullscreen parameters in the `initialize()` method for the game window, we can also pass in the `update_interval` parameter. By default this is set to 16.666666 milliseconds, which represents the interval between update calls.

```
1   require 'gosu'
2   include Gosu
3
4   class GameWindow < Gosu::Window
5
6     def initialize
7       super(800,600,false)
8       self.caption = "Game Loop"
9       @x = 10
10      @y = 10
11    end
12
13    def update
14      @x -= 1 if button_down?(Gosu::KbLeft)
15      @x += 1 if button_down?(Gosu::KbRight)
16      @y -= 1 if button_down?(Gosu::KbUp)
17      @y += 1 if button_down?(Gosu::KbDown)
18    end
```

```
19    def draw
20      @message = Gosu::Image.from_text(
21          self, "x: #{@x}, y: #{@y}", Gosu.default_font_name, 20)
22      @message.draw(@x, @y, 0)
23    end
24  end
25
26  window = GameWindow.new
27  window.show
```

Figure 2.6: A demonstration of how to combine the `update()` method and `draw()` method to move an image around the screen through user input.

2.4.1 Activities

1. Experiment with changing the increment used when the arrow keys are pressed. How does this impact the movement of the image? [Solution]

2. In earlier examples we had constructed the images in the `initialize()` method for the `GameWindow`. Try moving our construction of the @message image (line 25) into the `initialize()` method. What happens? Explain why the @message image needs to be constructed in the `draw()` method in order for the program to function as we planned. [Solution]

3. a) We are currently writing the real-time coordinates for the image to the screen. Add the code required to also write to the screen the number of times that the `draw()` method has been called. The image text will now look something like this "x: #@x, y: #@y, draws: #@draws". [Solution]

   b) What happens to the number of draws when you are not moving the image with the arrow keys? Why might this not be optimal? (Hint - needs_redraw?) [Solution]

   c) Implement the `needs_redraw()` method to prevent the @message image from being drawn when it is not necessary. [Solution]

2.5 Breakout - Create the Game Window

Brief Summary for Teacher

Students should now have the foundations and understanding to build the game window for their Breakout game. They should be able to create windows of varying sizes and understand the grid system used in the window, specifically that the origin is in the top left corner. Possibly the most important aspect for the students to fully understand is the game loop, as it forms the critical component in the games creation. Students should be able to verbally explain several iterations of the game loop, as it is implemented in Activity 2.4.1 Question 3.

Student Checklist

- ✓ Create a game window of appropriate dimension. Something around 800 x 600 pixels should work well.
- ✓ The window should not be full-screen.
- ✓ Include a caption of your choice.
- ✓ Make sure you require 'gosu' and include Gosu.
- ✓ Remember to actually invoke a new instance of the game window (`GameWindow.new`) and to call the `show()` method.
- ✓ Ensure that you have a good understanding of the game loop. Specifically, understand how the `update()` and `draw()` methods work together to create the basic foundation for our game.

Note: Please refer to the final Breakout game code in the appendix for the solution. Students would be provided with the incremental solution but it was not included here as it would result in excessive repeating of code.

3 The Paddle

We have the window set up where the game will be played and now we need to start creating our game logic and components. Our first task will be to create our paddle which is fixed vertically but moves horizontally across the lower portion of the screen based on the user's input.

3.1 Drawing the Paddle

In the previous chapter we saw how to create an image from text and how to load a PNG to use as our image. We are also able to use Gosu's Image class to draw images, specifically quadrilaterals, by passing it a collection of coordinates. Also remember that classes can be viewed as a type of factory that has the blueprints for creating specific objects in our game. We've created a `GameWindow` class that is responsible for constructing our game window and now we need to create a class that is responsible for our user's paddle. Our `initialize()` method for the `Paddle` class is going to need to know which game window to draw itself in, so we need to pass this as a parameter in our `initialize()` method. We'll also need a `draw()` method that invokes our `draw_quad()` method. At first glance, the `draw_quad()` method appears complicated with its many parameters, but it is actually quite simple. You are passing in the x and y coordinates for each of the 4 vertices, along with a single z-value.

```
1  class Paddle
2    def initialize(window)
3      @window = window
4    end
5
6    def draw
7      @window.draw_quad(
8          50, 50, Color::WHITE,
9          200, 50, Color::WHITE,
10         50, 100, Color::WHITE,
11         200, 100, Color::WHITE)
12   end
13 end
```

Figure 3.1: The `draw_quad()` method will be very useful as we start to build our game components.

3.1.1 Activities

1. On its own this `Paddle` class will not draw anything to your game window, as they are not connected in anyway to each other. Update your game window class to draw

the quadrilateral in Figure 3.1 onto the screen. [Solution]

2. Experiment with drawing the following types of quadrilaterals to the screen; a parallelogram, trapezoid and a kite. [Solution]

3. Draw a square and a rectangle. For each one calculate the central coordinate. This will be useful when we try to move our quadrilaterals around the screen. [Solution]

4. Draw a star with six vertices (Hint - `draw_triangle`). [Solution]

3.2 Moving the Paddle

We have the skills to draw the paddle to the screen but now we need to get it to move on user input. Something very similar was done when we learned about the game loop and the `update()` method, where the text image was moved around the screen using the arrow keys. Except in that case, we were moving an image that had a single set of coordinates and now we are working with a quadrilateral's four vertices. When we create a new paddle object using the `Paddle` class we are now going to have to also pass to it the (x,y) coordinates along with the window object. However, what point will this (x,y) coordinate represent? It could represent one of the four vertices and we could add and subtract from those coordinates to find the other 3 vertices. The problem with this is that it might make things slightly more confusing when we need to start doing calculations to see if the paddle has come into contact with other game components. For this project, I would suggest setting the x and y coordinates for the paddle to represent its center.

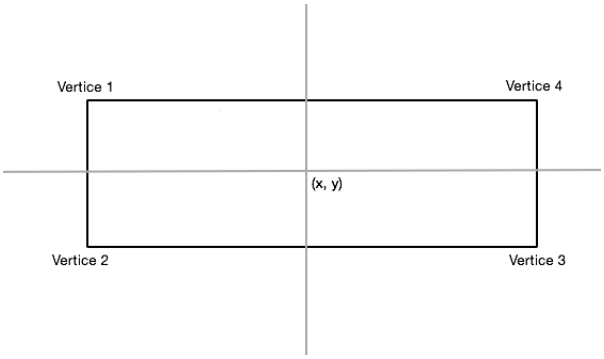

Figure 3.2: This figure shows the center of the paddle, which will be used to update its location, as well as its 4 vertices.

3.2.1 Activities

1. a) Write a program that allows the user to move a rectangle around the game window but instead of using the arrow keys use the W, A, S, and D keys. [Solution]

   b) Now add a second rectangle with a different colour to your program from question 1 and make this rectangle move around the screen using the arrow keys. [Solution]

c) What happens when the two rectangles overlap? Use the z-values to see if you can control which rectangle is shown on the top. [Solution]

3.3 Breakout - Create Your Paddle

Brief Summary for Teacher

If the students had a good grasp on the creation of the `GameWindow` class from Chapter 2 than they should experience success with creating their moving paddle. Although the addition of a new `Paddle` class does add some extra complexity that some students may initially struggle with. Struggling students likely find Activity 3.2.1 Question 1 difficult, as it requires the creation of two movable rectangles. Continuing to review the game loop and how the `update()` and `draw()` methods work together with `initialize()` method will help them to work through their difficulties.

Student Checklist

- ✓ Create a paddle of appropriate size. I would suggest around 50px in length and 10px in height but this is of course your personal choice.
- ✓ The paddle should be fixed vertically but move horizontally on key input from the user.
- ✓ Set the fixed vertical position to something that is close to the bottom of the screen (maybe 30-50px).
- ✓ Ensure that the paddle moves at an appropriate speed on user input. Too fast and it can be hard to control and jumpy. Too slow and it can be frustrating and difficult to get to the ball in time.
- ✓ The paddle should not be able to go off screen. In earlier examples we let this happen as there was no check being done to see if the paddle was at the end of the screen. I would suggest creating instance methods named something along the lines of move_left and move_right, which will help you to do a check and only subtract or add to the x position if appropriate.

Note: Please refer to the final Breakout game code in the appendix for the solution. Students would be provided with the incremental solution but it was not included here as it would result in excessive repeating of code.

4 The Ball

4.1 A Moving Ball

We want to start by just adding the ball to the game window and making it bounce around the window, completely ignoring the paddle. Once we have that working we can add the logic to make the ball also bounce off the paddle. Unlike the paddle, the ball needs to move on its own, independent of the user. The paddle waited for the user to direct its movements, while the ball should be updating its position in every iteration of the game loop. When the ball is constructed in its `initialize()` method we need to not only pass in its initial coordinates, but we also need to give it a horizontal and vertical velocity. For example, if the horizontal velocity, vx, is set to 5 and the vertical velocity, vy, is 3 then the ball will move 5 units to the right and 3 units up every time through the game loop.

4.1.1 Activities

1. Using your program from Chapter 3 with the game window and paddle classes, add a ball that begins in the bottom right of the window and moves towards the upper left corner. We will actually be using a square ball for our game, which does sound a little strange. I'd suggest making your ball a 5x5 pixel square. [Solution]
2. Once you have accomplished the above task, experiment with moving the ball in different directions and at different speeds. [Solution]

4.2 Bounce off the Walls

As you have likely noticed, our ball is currently moving off the screen. We need to find a way for the ball to check to see if it has reached one of the windows edges and bounce off in the opposite direction if it has. Of course we want the ball to follow the physical laws where the angle of incidence is equal to the angle of reflection (Figure 4.1). This may seem tricky but it is actually quite simple. Look at the figure below and think about how the horizontal and vertical velocity components have changed after the bounce, compared to if there was no wall. You can assume that the velocity of the ball does not change after the collision with the wall.

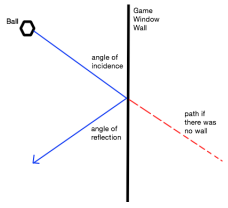

Figure 4.1: This figure shows the ball's path during a bounce off the wall.

4.2.1 Activities

1. We now need to stop the ball from moving through the edges of the game window. The ball should bounce around the game window but still pass directly through our paddle if they come into contact.

   a) Start by creating a method that checks to see if the ball has collided with either of the vertical walls. Remember that the x-value of the left wall will always be 0, while the right wall will have an x-value equal to the width of our game window. Therefore if the x-value of the ball is less than 0 or greater than the width of the game window then we want to change the horizontal direction of the ball.

   b) Now create a method that checks for the ball colliding with the top and bottom walls. You should be able to implement a strategy similar to the one applied in (a).

   c) Update the methods created in (a) and (b) to ensure that only the edge of the ball appears to contact the edge of the wall. Hint - remember that the coordinates for our ball represents its centre and not its edge.

   [Solution]

4.3 Bounce off the Paddle

Our ball is moving around our game window beautifully now, except it is still ignoring our paddle. This becomes a little trickier than bouncing off the walls, due to the fact that our paddle is moving and not stationary. We need a way to determine if a collision has occurred between the ball and the paddle. As you can imagine, collision detection is a very common aspect of game development and there are various strategies for implementing it. Luckily for us, we are dealing with simple rectangular shapes so we will have a relatively easy time implementing the bounding box detection method. Clearly, for every possible positioning case for the ball and paddle, they must either be intersecting or not intersecting (we will consider touching to be intersecting). We want to know if our ball and paddle are intersecting; however, in this case it is actually simpler for us to create a method that checks if they are not intersecting. We know that if this method returns false that our ball and paddle must have collided. So now, how do we go about checking to see if our two objects are not intersecting? Well, what must be true if the x-value of the right edge or our ball is less than the x-value of the left edge of our paddle? If this does not make sense, try drawing it out to help with you visualization. As you will likely see, if the above statement is true then our ball must not be intersecting with the paddle because our ball is on the left side of the paddle. You can use a similar check to see if the ball is to the right, above or below the paddle and if any of these cases is true then they must not be intersecting. It is also worth pointing out here that we will be considering the collision to be frictionless, although there is possible game extensions that incorporate a friction component.

4.3.1 Activities

1. Use the bounding box method to detect collisions between our ball and the paddle. I would suggest implementing the bounding box method within an instance method for

the ball, which takes in the paddle as a parameter and changes the vertical direction of the ball if a collision has occurred.[Solution]

2. We have ignored the fact that the ball could hit the side of the paddle. If the ball did hit the side of the paddle this would not help the player, as it would result in the ball changing its horizontal direction but not its vertical. But we want our game to look as realistic as possible and currently if the ball contacts the side of the paddle it will pass through. Update your method that checks for a collision between the ball and paddle to account for side collisions. As a hint, we will now need to run a check after a collision is detected to determine if it has contacted the side of the paddle. [Solution]

4.4 Breakout - Add the Ball

Brief Summary for Teacher

Students should now have a ball that bounces appropriately off the walls of the window and the paddle. Students may initially struggle with determining how to make their ball bounce with the angle of incidence equalling the angle of reflection; however, if they draw the ball bouncing with its respective x and y velocity vectors they should be able to reach the correct solution. Adequate testing should be done to ensure that the ball bounces correctly no matter where it hits the paddle, as depending on the values used there might be some areas with glitches.

Student Checklist

- ✓ It is suggested that the ball is a 10x10 pixel square.
- ✓ The ball should move freely throughout the whole window and bounce off the walls and paddle.
- ✓ Make sure that the speed of the ball is appropriate for the game. Something around 5px per frame should be a good start, but you are encouraged to experiment and test different ball speeds.

Note: Please refer to the final Breakout game code in the appendix for the solution. Students would be provided with the incremental solution but it was not included here as it would result in excessive repeating of code.

5 The Bricks

5.1 The Wall of Bricks

At this point we are experts at creating rectangles on the screen at any location that we want, so you may think that creating the bricks will be easy. The challenge is not creating a brick but creating all of the bricks in the correction arrangement. The Breakout game received its name from the idea of breaking out of the wall of bricks that you are faced with when the game starts. Once you have accomplished this you are able to get the ball bouncing between the top of the window and the remaining bricks, racking up multiple brick hits with minimal effort. This means that it is important that the bricks make up a wall with a space at the top. Refer to Figure 1.1 to get a reminder of the general idea for the bricks configuration.

You may be thinking that you will be able to create the bricks in the same fashion as you did the paddle and the ball. The issue with this is that you will have to create at least 40 bricks, assuming you have 4 rows of 10 bricks, and each of these will have unique coordinates. That would make for many lines of repetitive code, which we really do not want to do. We will discuss some options for using the Ruby language to do the majority of this tedious work, but first we need to figure out the dimensions and arrangement of our bricks. You might think that you could just arbitrarily choose a width and length for your bricks but the issue with this is that you might be left with a gap between your bricks and the wall. This would not be good because the ball would then have a path to sneak its way up to the top of the window before the user has actually broken through. The key to determine the correct brick size is in knowing the width of the window, the number of bricks and the size of the spaces you want between each brick. In Figure 5.1 we can see an example where the width of the window is 200px and the spacing between the six bricks is 2px. As there will be a total of 7 spaces, each measuring 2px, we are left with 186px for the bricks and therefore each brick would need to be 31px.

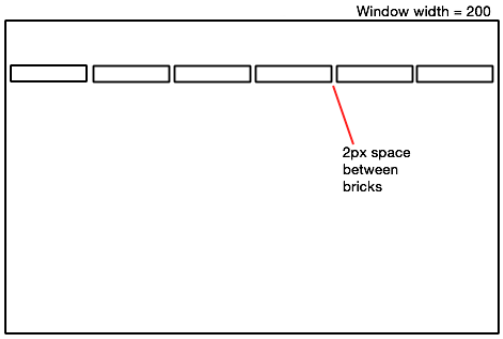

Figure 5.1: It is important that the row of bricks contains no spaces large enough for the ball to pass through.

Now that we know how many columns of bricks we will need in our wall we can start to focus on constructing them. Our goal is to create an array that contains all of the constructed brick objects with their correct coordinates. With this array, we can do a quick and easy `.each` iteration, moving through the array and drawing each of the bricks onto the screen. As an example, we will walk through creating an array that contains the six brick objects that would be needed to create the row of bricks seen in Figure 5.1. We can see that the distance between the midpoint of each brick is going to be the same, so we should be able to find a pattern for each of the brick's midpoints.

Bricks		Midpoint x-value
1	(2+31)-17.5	15.5
2	2(2+31)-17.5	48.5
3	3(2+31)-17.5	81.5
4	4(2+31)-17.5	114.5
5	5(2+31)-17.5	147.5
6	6(2+31)-17.5	180.5

Looking at the table we can start to see a pattern that we can apply to find the x-value for the midpoint of our bricks, no matter how many bricks we have. It is also important to note that the y-value is not important in this example, as we are only building one row the y-value will be the same for all these bricks. Now using this pattern and the `.each` iteration function in Ruby, we are able to create an array that contains all of the brick objects in this row (Figure 5.2). As mentioned earlier, once we have the array containing the brick objects it is only a matter of moving through the array and calling the `draw()` method on each object in the array to have the bricks drawn to the screen.

```
@bricks = []
(1..6).each do |column|
    @bricks << Brick.new(self, 33*col-17.5, 40)
end
```

Figure 5.2: Demonstrating the use of the `.each` method to create a row of bricks with uniform spacing.

5.1.1 Activities

1. a) Assume that we want the space between our bricks to always be 2px. With this in mind, create a formula for the width of each brick given that the window width is w and the number of bricks is b.[Solution]

   b) Using the example code in Figure 5.2 and your formula from part (a), create a row of bricks, with spacing of 2px between each, in your Breakout program from Chapter 4. Remember to leave a sufficient space at the top of the game window.[Solution]

2. The next step is to draw multiple rows of bricks to the screen to build up your wall of bricks. The decision as to how many rows to have is up to you, but I would suggest at least 4 or 5. Hint - you might need to use a `.each` iterator function within a `.each` iterator function. [Solution]

5.2 Removing Bricks

Adding our wall of bricks has really helped our game to actually start to take shape and resemble the original Breakout, although our ball is still ignoring the bricks and passing right through them. We now need to make the ball not only bounce off the bricks, but also cause the brick that was hit to disappear. The task of making the ball bounce off the bricks should not be anything too difficult, as it is very much what we did to have the ball bounce off the paddle. Once we have a method that checks to see if a brick is touching the ball, we simply need to run through the array of bricks and apply this method to each of them. If we find a brick that is touching the ball, we want to remove that brick from the array. This way, when the array is drawn again in the `draw()` method it will not draw this specific brick.

5.2.1 Activities

1. Now that we have our wall of bricks you need to add the code to remove any bricks that the ball has collided with.
   a) First lets focus on getting the ball to bounce off our bricks. Remember that we want the ball to bounce off not only the bottoms of the bricks but also the sides and tops of the bricks. As a hint, look back at your code for the collision between the paddle and the ball where we used the bounding box detection method.
   b) With our method for checking for brick collisions we can now start to remove bricks. We need to iterate through our array of bricks and check each one for any collisions. If a collision is detected we can then remove that brick from the array, thereby preventing it from being drawn. [Solution]

5.3 Breakout - Adding the Bricks

Brief Summary for Teacher

The challenge with creating the wall of bricks will be to not only calculate the appropriate number and dimensions of the bricks, but to also construct the correct algorithm that will create the array that contains all the brick objects. At this point in the creation of the game, it may make sense for students to start to change the dimensions of their window, paddle and bricks to improve gameplay.

Student Checklist

- ✓ The bricks are uniformly spaced and there are no gaps large enough for the ball to pass through.
- ✓ There is a gap beween the top row of bricks and the top of the window.
- ✓ The bricks are sized appropriately. I'd suggest playing the game a little to get a sense of whether you should make your bricks larger or smaller.
- ✓ The bricks are removed when the ball collides with them.

Note: Please refer to the final Breakout game code in the appendix for the solution.

6 Final Touches

6.1 Losing the Game

You may have noticed the one big glaring issue with our current game, there is no way to lose. The ball just continues to bounce around all the window edges and the user never loses. We need to fix this and make the game end when the ball gets past the paddle. To help with this and clean our code up a little, I would suggest pulling out some of the code from our `update()` method into its own separate method. The `update()` method can be viewed as the code required to run the game and currently it is always being run no matter what, which results in the game always running so long as the program is running. For this reason, it makes sense to pull the code from the `update()` method and put it into its own method that we can call `run_game()`.

```
def update
  run_game
end

def run_game
  @paddle.move_left if button_down?(KbLeft)
  @paddle.move_right if button_down?(KbRight)
  @ball.update
  @ball.change_horizontal_direction if (@ball.x >= 795 || @ball.x <= 5)
  @ball.change_vertical_direction if (@ball.y >= 595 || @ball.y <= 5)
  @ball.change_vertical_direction if @ball.hit?(@paddle)
  @bricks.each do |brick|
    if @ball.hit?(brick)
      @bricks.delete(brick)
      @ball.change_vertical_direction
    end
  end
end
```

Figure 6.1: We can pull all our logic that is used to run the game into its own method, thereby simplifying our `update()` method.

We could also start creating some separate methods for all of the different tasks being performed by the run_game method, but we will leave that for later. Now our `update()` method is much easier to understand; whenever the `update()` method is called the game is run. We need to add some logic to this so that the game is only run if the ball is above the paddle. It would be useful to make a variable such as `game_running` that is set to a boolean. Our `update()` method could then check to see if game_running is set to true before running the `run_game` method.

6.1.1 Activities

1. a) First, add the boolean variable to our `GameWindow` class and set it to initialize to true. Also, add a check to our `update()` method that only runs the game if `@game_running` is true. This should ensure that the game immediately starts running when the program starts.

   b) Create a `game_over?()` method that takes in the ball as a parameter and sets the `@game_running` variable to false if the ball has made it past the paddle. Test your game to ensure that your logic works accordingly.
   [Solution]

6.2 Starting the Game and Playing Again

Another issue with our game is that the game starts immediately after the program starts running. Also, once the player has lost, there is no way to play again without closing and restarting the program. Both of these problems can be solved using the `game_running` variable that we just created to check when the game is lost. It should be possible for us to leverage this variable and set it to false if the user has not requested that the game start. When the user takes the required action we can set the `game_running` variable to true and invoke the `run_game` method from our `GameWindow` update.

Restarting the game after the player loses is quite similar, except for one problem. If we allow the user to simply start playing again by clicking the 'play again' key then the ball will just continue moving from the point below the paddle. Therefore, we also need to reset the balls position as well as set the `game_running` variable back to true.

6.2.1 Activities

1. a) Make the game start when the user has pressed a specific key, such as the spacebar. Remember that we will now want our `@game_running` variable to initialize to false and we will change its value on the user input.

   b) We now want to be able to restart the game after we have lost. Create a `play_again?` method that resets the `@game_running` variable to true if it is false and the space bar has been pressed.

   c) You will have likely noticed that our game can now restart but our ball just continues moving from its current location. We need to create a method that resets the balls position and this needs to be invoked after setting the `@game_running` variable to true in our `play_again?` method.

   d) As an extension you could also add a message that appears on the screen that prompts the user to press the specified key to start or replay the game.
   [Solution]

6.3 Breakout - Finishing Your Game

Brief Summary for Teacher

Students will have the basic components and game play functioning now and it is only a matter of adding some final touches. For a simple solution, I suggest implementing a method that checks if the user has requested the game to start and/or replay as well as a method for checking if the ball has made it past the paddle. These methods can be used as filters to only allow the `run_game()` method to be called when appropriate. However, this is the point in the project where students may start to experience some scalability issues. If students wish to continue to add extensions and new game play components to their game then they will realize that they may need to rework some of their earlier game logic in order to proceed. This is an excellent opportunity to discuss the importance of writing clean and scalable code.

Student Checklist

- ✓ The game loop should stop when the ball passes below the paddle.
- ✓ When the program starts, the game loop should wait for user input before starting.
- ✓ When the user loses they should be able to restart by pressing a specific key and the ball should reset its location.

Note: Please refer to the final Breakout game code in the appendix for the solution. Students would be provided with the incremental solution but it was not included here as it would result in excessive repeating of code.

7 Extension

We have now created our nice little Breakout game. Although, I am sure that you have noticed that there are certain aspects missing that would improve the game play. Here are some suggested extensions that will improve your game even further. These are of course just suggestions and you are encouraged to use your imagination to improve the game in any way you see fit.

- It would be great to be able to keep score. Points could be awarded for each brick that is cleared, possibly awarding more points for bricks that are in the higher rows. It would also make sense to colour the bricks different depending on the number of points they are worth.
- Currently you can keep on continuing the game after losing and the bricks are not reset. It would make sense if the player gets a specific number of lives and once they have lost that many times, the bricks (and score) reset.
- We currently hard code all our important values (game window width and length, ball size, paddle size, brick size) into the program directly. It would be fantastic if we set these to global variables. This would allow you to change these values in one spot in the program without having to go through and look for every line of code where it is called. This would be very useful for testing out different dimensions for your window and game components.
- It would be pretty neat if the game got more challenging the longer the user was playing for. Maybe you could make the ball's speed get faster and faster as the game progresses or the wall of bricks slowly move down towards the paddle (or both!).
- Another fun idea would be to add a power-up component to the game. There could be a couple of randomly selected bricks that when cleared released a power-up and if the paddle touches the power-up before it reaches the bottom of the screen the user gains some sort of ability. Some possible ideas for power-ups include;
  - increasing or decreasing the size of the ball and/or paddle
  - gaining an extra life
  - making the paddle 'sticky' so that it catches the ball and the user can release the ball on key input
  - adding multiple balls

8 Solutions

8.1 Chapter 2 Solutions

8.1.1 Section 2.2.1

1. If the program is not working read the error carefully and see if you can problem solve your way to the solution. Ask a peer or teacher if you continue to experience difficulties.

2. The order does matter, as the the width is the first parameter and the height is the second parameter.

```
require 'gosu'
include Gosu

class GameWindow < Gosu::Window
  def initialize
    super(300,500,false)
  end
end

window = GameWindow.new
window.show
```

3. The game window is not set to the dimensions of the screen, but instead it creates a game window that is a maximized fullscreen window.

8.1.2 Section 2.3.1

1. a) The coordinate axes;

b) The origin is in the top left corner of the game window.

2. a) Drawing a png of choice anywhere one screen:

```
class GameWindow < Gosu::Window
  def initialize
    super(800,600,false)
    self.caption = "Setting Up Game Window"
    @png = Gosu::Image.new(self, "brick.png", true)
  end

  def draw
    @png.draw(400,300,0)
  end
end
```

   b) Scaling the png to be larger and smaller:

```
@png.draw(400,300,0, 4, 4)  #scales to be 4x larger
@png.draw(400,300,0, 0.5, 0.5)  #scales to be half the size
```

   c) The item does not appear in the centre because the top left corner of the image is placed at the coordinate passed to the method.

3. a) The image that is drawn first appears on the bottom while the image drawn second appears on top.

   b) The image with the greater z-value appears on top of the image with the smaller z-value.

8.1.3 Section 2.4.1

1. The image is moved farther with each button click as the increment is increased. It has an appearance of moving faster, unless the increment is much higher as it then makes the image appear to jump between different locations on the window.

2. If the image is constructed in the `initialize()` method then the image is only constructed once, when the game window is invoked. At that time, it is passed the initial x and y values and these will not be updated as the image moves because the image is not being constructed in the `draw()` method.

3. a) Part c includes the code for writing the number of draws to the screen.

   b) The number of draws continues to increase even when the image is not moving. The computer is having to do processing work when it isn't necessary. It is minor in this case because what it is drawing is a minimal object, but the same principal could be applied to more complicated images or processing tasks.

   c) Implementing the `needs_redraw?` method:

```
class GameWindow < Gosu::Window
  def initialize
    super(800,600,false)
    self.caption = "Game Loop"
    @x = 10
    @y = 10
    @draw = 0
    @buttons_down = 0
  end
```

```
10    def update
11      @x -= 1 if button_down?(Gosu::KbLeft)
12      @x += 1 if button_down?(Gosu::KbRight)
13      @y -= 1 if button_down?(Gosu::KbUp)
14      @y += 1 if button_down?(Gosu::KbDown)
15    end
16    def button_down(id)   #adding to both Gosu's button_down and button_up methods
17       @buttons_down += 1
18    end
19    def button_up(id)
20      @buttons_down -= 1
21    end
22    def needs_redraw?     #if @draw is 0 then there have been not updates
23      @draw == 0 || @buttons_down > 0  #if @buttons_down > 0 then a key was pressed
24    end
25    def draw
26      @draw += 1      #increment @draws each time the draw method is invoked
27      @message = Gosu::Image.from_text(self, "x: #{@x}, y: #{@y}, draw: #{@draw}",
28          Gosu.default_font_name, 20)
29      @message.draw(@x, @y, 0)
30    end
31  end
```

8.2 Chapter 3 Solutions

8.2.1 Section 3.1.1

1. To draw the quad to the screen we need to update our `initialize()` method:

   ```
   @paddle = Paddle.new(self)  #self indicates the game window itself
   ```

   Add the `draw()` method that we discussed earlier to our `GameWindow` class:

   ```
   def draw
     @paddle.draw
   end
   ```

2. Parallelogram

   ```
   @window.draw_quad(200,325,@colour,375,375,@colour,375, 125, @colour,200,75,@colour)
   ```

   Trapezoid

   ```
   @window.draw_quad(175,165,@colour,275,165,@colour,125,265, @colour,390,265,@colour)
   ```

   Kite

   ```
   @window.draw_quad(60,130,@colour,120,50, @colour,180,130,@colour,120,200,@colour)
   ```

3. The center of the square and rectangle can be found by calculating the midpoint for the length and width.

4. Star

   ```
   @window.draw_triangle(300,100,@colour,500,400,@colour,100,400,@colour)
   @window.draw_triangle(100,200,@colour,500,200,@colour,300,500,@colour)
   ```

8.2.2 Section 3.2.1

1. a) Refer to solution 1b.

b) Controlling two separate paddles simultaneously:

```
require 'gosu'
include Gosu

class GameWindow < Gosu::Window
  def initialize
    super(800,600,false)
    self.caption = "Breakout"
    @paddle1 = Paddle.new(self, 200, 200, Color::WHITE)
    @paddle2 = Paddle.new(self, 600, 200, Color::RED)
  end

  def update
    @paddle1.x -= 1 if button_down?(Gosu::KbLeft)
    @paddle1.x += 1 if button_down?(Gosu::KbRight)
    @paddle1.y -= 1 if button_down?(Gosu::KbUp)
    @paddle1.y += 1 if button_down?(Gosu::KbDown)
    @paddle2.x -= 1 if button_down?(Gosu::KbA)
    @paddle2.x += 1 if button_down?(Gosu::KbD)
    @paddle2.y -= 1 if button_down?(Gosu::KbW)
    @paddle2.y += 1 if button_down?(Gosu::KbS)
  end

  def draw
    @paddle1.draw
    @paddle2.draw
  end
end

class Paddle
  attr_accessor :x, :y
  def initialize(window, x, y, colour)
    @window = window
    @x = x
    @y = y
    @colour = colour
  end

  def draw
    @window.draw_quad(
      @x-75,@y-25,@colour,@x-75,@y+25,@colour,
      @x+75,@y+25,@colour,@x+75,@y-25,@colour)
  end
end

window = GameWindow.new
window.show
```

8.3 Chapter 4 Solutions

8.3.1 Section 4.1.1

1. The code for the Ball and `GameWindow` classes are given below, while the Paddle class would remain unchanged.

```
class GameWindow < Gosu::Window
  def initialize
    super(800, 600, false)
    self.caption = 'Breakout'
    @paddle = Paddle.new(self, 400, 580)
    @ball = Ball.new(self, 750, 550, -3, -2)
  end
  def update
    @paddle.move_left if button_down?(KbLeft)
    @paddle.move_right if button_down?(KbRight)
    @ball.update
  end
  def draw
    @paddle.draw
    @ball.draw
  end
end

class Ball
  attr_accessor :x, :y
  def initialize(window, x, y, vx, vy)
    @window = window
    @x = x
    @y = y
    @vx = vx
    @vy = vy
  end

  def update
    @x += @vx
    @y += @vy
  end

  def draw
    @window.draw_quad(
      @x-5,@y-5,Color::WHITE,@x+5,@y-5,Color::WHITE,
      @x+5,@y+5,Color::WHITE,@x-5,@y+5, Color::WHITE)
  end
end
```

8.3.2 Section 4.1.2

1. We will need to update our program from 4.1.1. The Ball class needs the following two methods:

```
def change_vertical_direction
  @vy *= -1
end
def change_horizontal_direction
  @vx *= -1
end
```

a)
```
@ball.change_horizontal_direction if (@ball.x >= 800 || @ball.x <= 0)
```

b)
```
@ball.change_vertical_direction if (@ball.y >= 600 || @ball.x <= 0)
```

c)
```
@ball.change_horizontal_direction if (@ball.x >= 795 || @ball.x <= 5)
@ball.change_vertical_direction if (@ball.y >= 595 || @ball.y <= 5)
```

8.3.3 Section 4.1.3

1. Adding the following method to the Ball class will give it the ability to check to see if it has collided with the paddle:

```
def paddle_hit?(paddle)
    if (@x+5 >= (paddle.x-25) && (@x-5 <= paddle.x+25) &&
        (@y + 5 >= paddle.y - 5) && @y - 5 <= paddle.y + 5)
        self.change_vertical_direction
    end
end
```

We must also invoke the `paddle_hit?` method in our game loop `update` method.

```
    @ball.hit?(@paddle)
```

2. To check for ball collisions on the side of the paddle we must add some checks to determine the balls location after a collision has occurred.

```
  def paddle_hit?(paddle)
    if (@x+5 >= (paddle.x-25) && (@x-5 <= paddle.x+25) &&
        (@y + 5 >= paddle.y - 5) && @y - 5 <= paddle.y + 5)
      if (@x < (paddle.x + 25) && (@x > (paddle.x - 25)))
        self.change_vertical_direction
      elsif (@x <= paddle.x - 25)
        if self.vx < 0
          self.change_vertical_direction
        else
          self.change_horizontal_direction
        end
      elsif (@x <= paddle.x + 25)
        if self.vx > 0
          self.change_vertical_direction
        else
          self.change_horizontal_direction
        end
      end
    end
  end
```

8.4 Chapter 5 Solutions

8.4.1 Section 5.1.1

1. a) The formula for the width of the bricks is:
$BW = \frac{w-2b-2}{b}$
where w is the window width and b is the number of bricks.

   b) You will need to add a Brick class:

```
class Brick
  attr_reader :x, :y

  def initialize(window, x, y)
    @window = window
    @x = x
    @y = y
  end

  def draw
    @window.draw_quad(
      @x-25,@y-5, Color::RED,
      @x+25,@y-5, Color::RED,
      @x+25,@y+5, Color::RED,
      @x-25,@y+5, Color::RED,
      )
  end
end
```

   You also need to construct an array of brick objects in the `GameWindow`'s `initialize()` method. The values used below are based on a window with a width of 800px:

```
    @bricks = []
    (1..15).each do |column|
      @bricks << Brick.new(self, 53.125*column-25, 50)
    end
```

   Lastly you will need to make sure that you iterate through the array of brick objects and draw them to the screen in the `draw()` method of your window:

```
    @bricks.each do |brick|
      brick.draw
    end
```

2. To get the multiple rows of bricks you will need to update the each iterator function in the `GameWindow` `initialize()` method. We need to add a second `.each` method to iterate through each column of bricks and make multiple rows:

```
    @bricks = []
    (1..15).each do |column|
      (1..6).each do |row|
        @bricks << Brick.new(self, 53.125*column - 25, 13*row + 50)
      end
    end
```

8.4.2 Section 5.2.1

1. a) Refer to b.

   b) Instead of creating a new method to check to see if the ball has hit the paddle, I have reused the method `paddle_hit?()` that we created to check to see if the ball hit the paddle and renamed it `obj_hit?()` and passed it the following parameters; the object, the objects width, and the objects height. Remember you will need to update the `paddle_hit?()` method to `obj_hit?()`.

```
def obj_hit?(obj, objwidth, objheight)
  if (@x+5 >= (obj.x-(objwidth/2)) && (@x-5 <= obj.x+(objwidth/2)) &&
      (@y + 5 >= obj.y - (objheight/2)) && @y - 5 <= obj.y + (objheight/2))
    if ((@x < (obj.x + (objwidth/2))) &&
      (@x > (obj.x - (objwidth/2))))
      self.change_vertical_direction
    elsif (@x <= obj.x - (objwidth/2))
      if self.vx < 0
        self.change_vertical_direction
      else
        self.change_horizontal_direction
      end
    elsif (@x >= obj.x + (objwidth/2))
      if self.vx > 0
        self.change_vertical_direction
      else
        self.change_horizontal_direction
      end
    end
  end
end
```

   My `update()` method in the `GameWindow` now uses the `hit?` method to check to see if the ball has hit any of the bricks in the array, and if it has, it deletes that brick and changes the vertical direction of the ball.

```
  @bricks.each do |brick|
    brickwidth = self.brick_width(800,15)
    if @ball.obj_hit?(brick, brickwidth, 20)
      @bricks.delete(brick)
    end
  end
```

8.5 Chapter 6 Solutions

8.5.1 Section 6.1.1

1. a) Refer to b.

   b) Add the boolean variable to the game window:

```
  @game_running = true
```

   Now we need to create a `game_over?` method and invoke it in our `update()` method:

```
def game_over?(ball)
  @game_running = false if ball.y > 590
end

def update
  game_over?(@ball)
  if @game_running
    run_game
  end
end
```

8.5.2 Section 6.1.2

1. a) Refer to c.

   b) Refer to c.

   c) We need to change our @game_running variable to false, so that when the program starts the game is not running. We can then create a method for checking if the game has started, if the user has requested to play again, and a method to reset the balls location. These are then used within the `update()` method to check to see if the `run_game` method needs to be invoked. The ball's vertical velocity has also been reset to be negative so that the user has time to move the paddle from its losing location.

```
def game_start?
  @game_running = true if button_down?(KbSpace)
end
def play_again?
  if (@game_running == false && button_down?(KbSpace))
    @game_running = true
    reset_position
  end
end
def reset_position
  @ball.x = 400
  @ball.y = 300
  @ball.vy *= -1
end
def update
  game_start?
  game_over?(@ball)
  play_again?
  if @game_running
    run_game
  end
end
```

   d) Refer to the final program included in the appendix.

Appendix

The code snippets in this project were created using the minted package for Latex.

Final Breakout Game

A video of the code for the Breakout game running is included here.

```
require 'gosu'
include Gosu

class GameWindow < Gosu::Window
  def initialize
    super(800, 600, false)
    self.caption = 'Breakout'
    @font = Gosu::Font.new(self, Gosu::default_font_name, 20)
    @paddle = Paddle.new(self, 400, 580)
    @ball = Ball.new(self, 300, 400, -2.5, -4)
    @bricks = []
    (1..15).each do |column|
      (1..6).each do |row|
        @bricks << Brick.new(
        self, (2+(brick_width(800,15)))*column - (brick_width(800,15)/2), 23*row + 50)
      end
    end
    @game_running = false
    @first_game = true
  end

  def game_over?(ball)
    @game_running = false if ball.y > 590
  end

  def play_again?
    if (@game_running == false && button_down?(KbSpace))
      @game_running = true
      reset_position
    end
  end
```

```
32    def game_start?
33      if button_down?(KbSpace)
34        @game_running = true
35        @first_game = false
36      end
37    end
38
39  def reset_position
40      @ball.x = 400
41      @ball.y = 300
42      @ball.vy *= -1
43    end
44
45    def brick_width(w, b)
46      return (w - 2*b - 2)/b
47    end
48
49    def update
50      game_start?
51      game_over?(@ball)
52      play_again?
53      if @game_running
54        run_game
55      end
56    end
57
58    def run_game
59      @paddle.move_left if button_down?(KbLeft)
60      @paddle.move_right if button_down?(KbRight)
61      @ball.update
62      @ball.change_horizontal_direction if (@ball.x >= 795 || @ball.x <= 5)
63      @ball.change_vertical_direction if (@ball.y >= 595 || @ball.y <= 5)
64      @ball.obj_hit?(@paddle,50,10)
65      @bricks.each do |brick|
66        if @ball.obj_hit?(brick, self.brick_width(800,15), 20)
67          @bricks.delete(brick)
68        end
69      end
70    end
```

```
71    def draw
72      @paddle.draw
73      @ball.draw
74      @bricks.each do |brick|
75        brick.draw
76      end
77      if @first_game == true
78        @font.draw("Welcome to Breakout! Press Spacebar to start game.",
79        175, 350, 3.0, 1.0, 1.0, 0xffffffff)
80      end
81      if (@game_running == false && @first_game == false)
82        @font.draw("Game Over! Press Spacebar to restart.",
83        225, 350, 3.0, 1.0, 1.0, 0xffffffff)
84      end
85    end
86  end
87
88  class Ball
89    attr_accessor :x, :y, :vy, :vx
90
91    def initialize(window, x, y, vx, vy)
92      @window = window
93      @x = x
94      @y = y
95      @vx = vx
96      @vy = vy
97    end
98
99    def obj_hit?(obj, objwidth, objheight)
100     if (@x+5 >= (obj.x-(objwidth/2)) && (@x-5 <= obj.x+(objwidth/2)) &&
101         (@y + 5 >= obj.y - (objheight/2)) && @y - 5 <= obj.y + (objheight/2))
102       if ((@x < (obj.x + (objwidth/2))) && (@x > (obj.x - (objwidth/2))))
103         self.change_vertical_direction
104       elsif (@x <= obj.x - (objwidth/2))
105         if self.vx < 0
106           self.change_vertical_direction
107         else
108           self.change_horizontal_direction
109         end
110       elsif (@x >= obj.x + (objwidth/2))
111         if self.vx > 0
112           self.change_vertical_direction
113         else
114           self.change_horizontal_direction
115         end
116       end
117     end
118   end
```

```
119   def update
120     @x += @vx
121     @y += @vy
122   end
123
124   def change_vertical_direction
125     @vy *= -1
126   end
127
128   def change_horizontal_direction
129     @vx *= -1
130   end
131
132   def draw
133     @window.draw_quad(
134       @x-5,@y-5, Color::WHITE,
135       @x+5,@y-5, Color::WHITE,
136       @x+5,@y+5, Color::WHITE,
137       @x-5,@y+5, Color::WHITE,
138       )
139   end
140 end
141
142 class Paddle
143   attr_accessor :x, :y
144
145   def initialize(window, x, y)
146     @window = window
147     @x = x
148     @y = y
149   end
150
151   def move_left
152     @x -= 5
153     @x = 25 if @x < 25
154   end
155
156   def move_right
157     @x += 5
158     @x = 775 if @x > 775
159   end
160
161   def draw
162     @window.draw_quad(
163       @x-25,@y-5, Color::WHITE,
164       @x+25,@y-5, Color::WHITE,
165       @x+25,@y+5, Color::WHITE,
166       @x-25,@y+5, Color::WHITE,
167       )
168   end
169 end
```

```
170 class Brick
171   attr_reader :x, :y
172
173   def initialize(window, x, y)
174     @window = window
175     @x = x
176     @y = y
177   end
178
179   def draw
180     @window.draw_quad(
181       @x-(@window.brick_width(800,15)/2),@y-10, Color::RED,
182       @x+(@window.brick_width(800,15)/2),@y-10, Color::RED,
183       @x+(@window.brick_width(800,15)/2),@y+10, Color::RED,
184       @x-(@window.brick_width(800,15)/2),@y+10, Color::RED,
185       )
186   end
187 end
188
189 window = GameWindow.new
190 window.show
```

Bibliography

1. Collingbourne, H., *The Little Book Of Ruby*, Dark Neon Ltd., New York, 2008.
   In this resource, basic foundational concepts of the Ruby programming language are covered. It will act as a good resource for supplementing the students prior knowledge.

2. Gillette, J., *why's (poignant) Guide to Ruby*, Creative Commons Distribution License, 2007.
   This source provides a good breadth of information on the Ruby programming language, from the basic concepts of strings and arrays to more complicated topics like as classes and inheritance.

3. Radocchia, S. (2013, June 18). Look Ma, I Built a Game! [Web log post]. Retrieved from http://blog.flatironschool.com/look-ma-i-built-a-game/
   This source provides an introduction to building games with Ruby and the Gosu library. It also links to the source code for a version of the Breakout game.

4. Varaneckas T., *Developing Games with Ruby*, Leanpub, Vancouver, 2014.
   This source is a detailed and thorough look at building games with the Ruby language.

5. Gosu's Documentation (rdoc). http://www.libgosu.org/rdoc/.
   This source is the complete documentation for the Gosu library.

6. Gosu's Ruby Tutorial. https://github.com/jlnr/gosu/wiki/Ruby-Tutorial.
   This source gives a detailed tutorial on various aspects of using Gosu with the Ruby language, including setting up the game window, drawing images and moving objects.

7. Ruby Monk. http://rubymonk.com/learning/books/1-ruby-primer.
   This source is a collection of free interactive tutorials focused on Ruby. It will act as a good resource for students to review and reinforce the concepts learned.

8. CodeAcademy - Ruby Track. http://www.codecademy.com/tracks/ruby.
   This source focuses mainly on syntax and basic concepts of Ruby.