Keyboard driven movement using Ruby and Gosu.

A complete tutorial.

Written by Terry Lay - 1742434

User driven manipulation of objects using ruby and GOSU

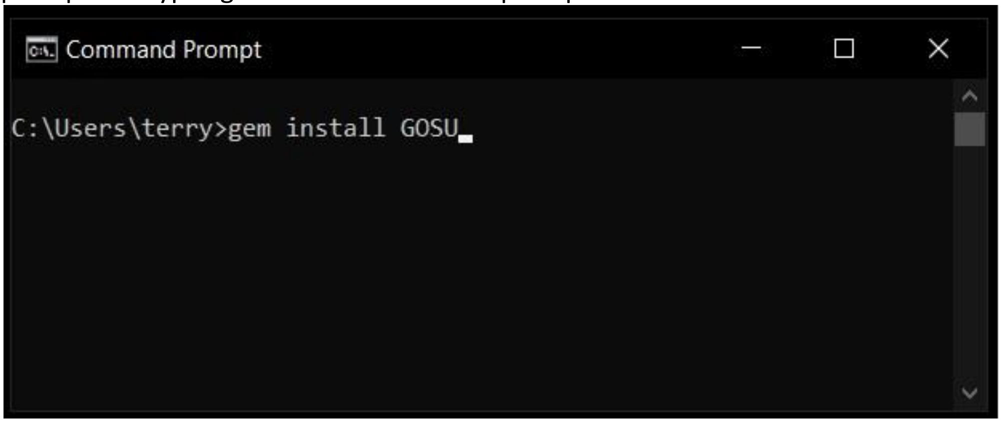
This tutorial requires an up to date install of Ruby and an integrated development environment (IDE) of your choice. Please note. This tutorial is for windows users. Whist code functionality will be the same of Mac OS the install process will differ slightly. Examples shown are being written in Visual Studio Code. However your preferred IDE will work just fine.

What is a gem?

Ruby using the gem GOSU offers a variety of powerful tools to enable user interactivity within programs. Here we look at a common method used to enable keyboard input of rendered objects within a game window.

1. *Installation of the gem GOSU*

Ruby has the ability to install additional functionality through the addition of GEMS. These gems are installed through the command prompt (CMD). In order to install the GOSU gem open the command prompt and type "gem install GOSU" This prompts the download and installation of our gem.



```
C:\Users\terry>gem install GOSU_
```

Figure 1. (Left) Command prompt command to install the ruby gem GOSU.

2. Create your ruby file.

First open up your IDE. For this example we will call our file "tutorial.rb" and save the file on your desktop. This will make it easier to find, debug and run our file in later steps. Please note the case of the text used as this is an important detail. Failure to do so will result in an unfunctional program.

Once done your developer environment should look like this. Ensure that after each step or before the program is run that the file is saved.

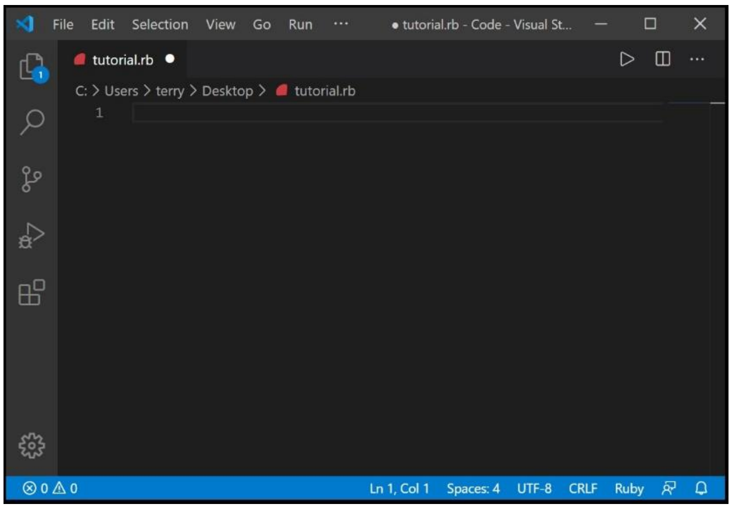

File Edit Selection View Go Run ···
● tutorial.rb - Code - Visual St...
tutorial.rb ●
C: > Users > terry > Desktop > tutorial.rb
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Ruby

Figure 2. (Above) A blank IDE environment set up and ready to go.

3. CREATION OF GAME CONSTATNS AND THE REQUIRE

The next step involves telling Ruby to include "require" our GOSU gem. This can be done by entering "*require 'gosu'*" on the first line of your project. Next we want to set up our constants and game window.

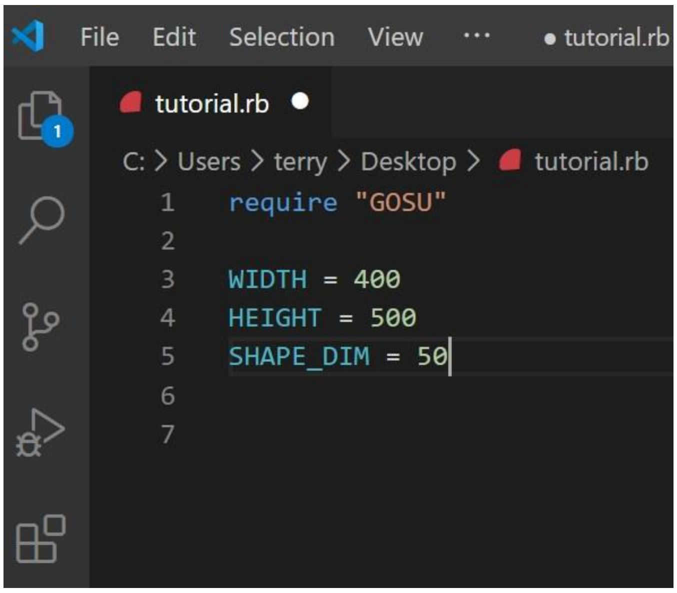
First we should include some constants. Constants are used to represent global values in your program that remain unchanged throughout its operation. For this exercise we will include:

WIDTH = 500

HEIGHT = 500

SHAPE_DIM = 50

Please not the case of these lines. The upper case tells Ruby that these lines are global constants.



```ruby
require "GOSU"

WIDTH = 400
HEIGHT = 500
SHAPE_DIM = 50
```

Figure 3. (Above) Showing the require as well as global constants.

4. CREATION OF OUR GAME WINDOW

The next step involves setting up the template we will use for our code as well as setting up the game window that will tell Ruby to display the result of our code to the user.

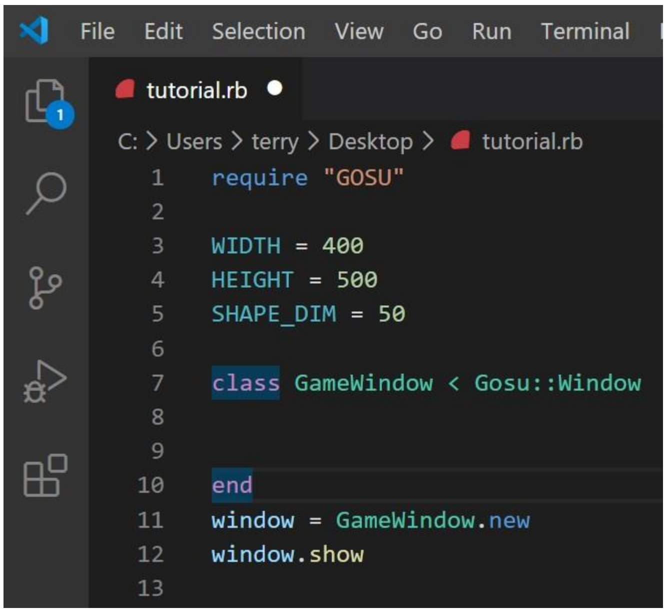
Include the following code in your file:

class GameWindow < Gosu:::Window

end

window = GameWindow.new

window.show



```ruby
require "GOSU"

WIDTH = 400
HEIGHT = 500
SHAPE_DIM = 50

class GameWindow < Gosu::Window


end
window = GameWindow.new
window.show

```

Figure 4. (Above) Displaying our GOSU window class

This code defines the container we will use to write the majority of our program and provides a foundation for us to initialize core components. This window also provides timing functionality. A feature we will look at shortly. The "*window = GameWindow.new" and the "window.show"* commands tell Ruby to display and run all code within the class. Without these commands the game window would never be presented to the user.

5. ADDITION OF A MODULE TO ENABLE DEPTH.

Next we will add a module to our program. A Ruby Module is similar to a class. It is a container that has the ability to hold a collection of methods or constants. But unlike classes they can not hold objects. In this example we are using it as a way of telling Ruby which layer of our 2 dimensional game window we wish our shape to occupy. This allows object layering and more complex interactions to take place. A good way to think about it is from the perspective of a photographer. Each value in the module represents either the foreground, middle or background of a picture. Place the following code under your "*require*" statement and above the global constants we created earlier.

module ZOrder

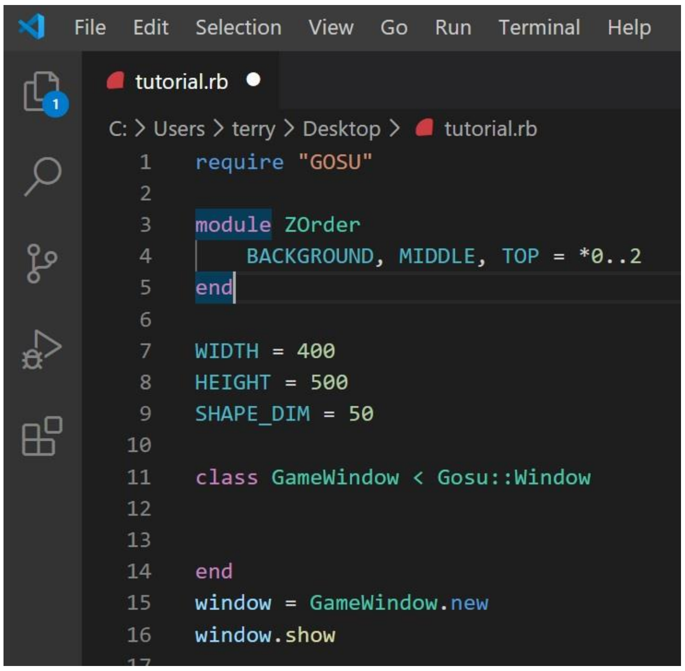
*BAKCGROUND, MIDDLE, TOP = *0..2*

END

Again note the case of this code. It will be identical when we reference it in later code. The statement "*=*0..2*" tells Ruby what value we have assigned to each of or options. Ruby stores this information in the form of an Array. Signified by the *.

Please see figure 5 as reference for your code.

Something to start paying attention to at this stage is the indenting of the lines. Note, everything within our project starts as far to the left of our document as allows. Whilst the "*BACKGROUND*" written on line 4, is indented slightly. This is done to increase readability of our code and is considered good practice. You will notice later that all code will follow this convention.



```
require "GOSU"

module ZOrder
    BACKGROUND, MIDDLE, TOP = *0..2
end

WIDTH = 400
HEIGHT = 500
SHAPE_DIM = 50

class GameWindow < Gosu::Window


end
window = GameWindow.new
window.show
```

Figure 5. (Above) Step 4, Showing the addition of our module.

6. Lets initialize our program

Next we include an initialize function. This function tells Ruby what to load and what variables to initialize when the program is started. Whilst this section of the window holds no code that is responsible for physically drawing our moveable shape it does include code that tells that code how it should function. Aspects of the program such as sprits, instance variables, soundtracks and backgrounds can all be initialized within this section. We also use this space to tell Ruby the specifics of our window. Include the following code in your project. Pay special attention to the use of special characters, and indentation.

def initialize

super WIDTH, HEIGHT, false

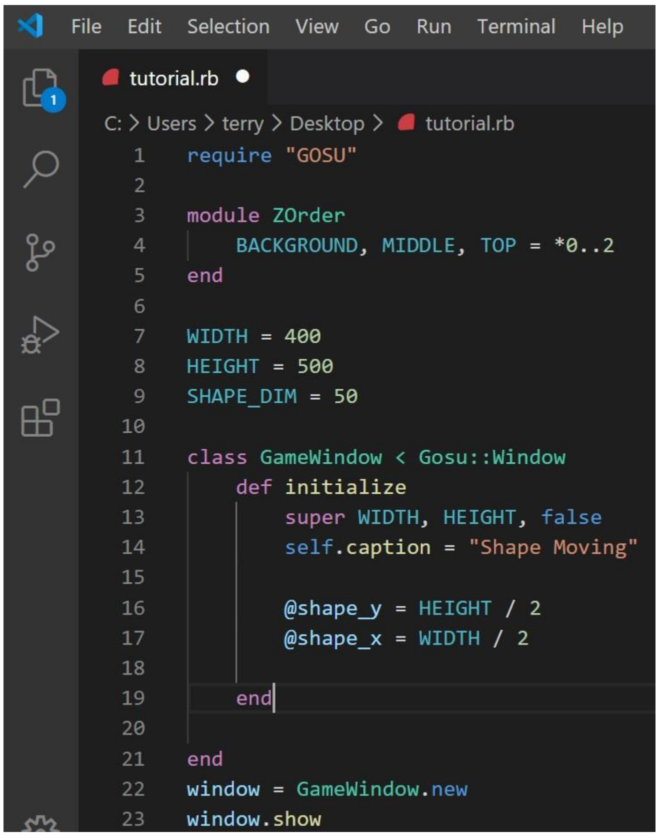
self.caption = "Our first moving shape"

@shape_y = HEIGHT / 2

@shape_x = WIDTH / 2

end

This code is fairly readable. A great attribute of the Ruby programming language. Note the use of the global variables we defined earlier. "WIDTH" and "HEIGHT". They are being used here to tell Ruby the physical dimensions of our game window. The *"self.caption = …"* is code used to label our game window. You can populate between the " " as you like. Your code should look like figure 6 to the right.



```
require "GOSU"

module ZOrder
    BACKGROUND, MIDDLE, TOP = *0..2
end

WIDTH = 400
HEIGHT = 500
SHAPE_DIM = 50

class GameWindow < Gosu::Window
    def initialize
        super WIDTH, HEIGHT, false
        self.caption = "Shape Moving"

        @shape_y = HEIGHT / 2
        @shape_x = WIDTH / 2

    end

end
window = GameWindow.new
window.show
```

Figure 6. (Above) Our code. Now containing our initialize.

7. Let's draw our shape!

Next, we can draw our shape! Gosu has tools built in to help us with this task. But first lets talk a little bit about the coordinate system Ruby uses to identify different sections of our game window. Imagine a graph from high school. You have an "x" and a "y" axis. Ruby has something very similar. "y" is your height whilst "x" is your width. The coordinates 0,0 represent the top left corner of your window. Whilst "HEIGHT" and "WIDTH" (500, 500) represent the bottom right corner of the screen. In the above code you will notice our instance variables "*@shape_x"* and "*@shape_y"*. In this example these instance variables represent the initial "x" and "y" coordinates our shape will populate when it is rendered. Note also the " *= HEIGHT / 2"* and *"= WIDTH / 2"*. What this code does is tell Ruby the initial "x", "y" coordinates of our shape are a value that is half the height of our window as well as half the width of our window.

We can now draw our shape. Enter the following code under initialize as follows.

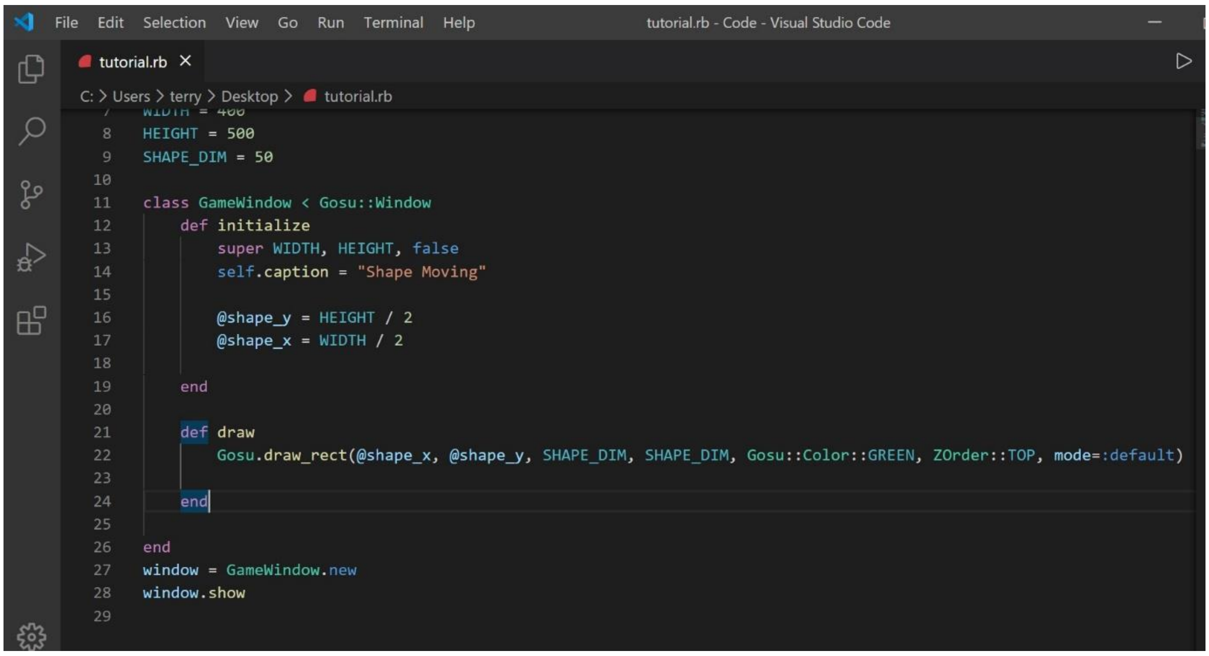
def draw

Gosu.draw_rect(@shape_x, @shape_y, SHAPPE_DIM, SHAPE_DIM, Gosu::Color::GREEN, ZOrder::TOP, mode=:default)

end

The following code ties in all of the ideas we have learnt up to this point. *"Gosu.draw_rect"* is an inbuilt ruby function that helps us draw our shape. Note how we have then used our "*@shape_x"* and "*@shape_y"* again. That is because the next peace of information Gosu needs to draw our shape is it's initial "x" and "y" coordinates. Additionally, you'll notice the use of our global constant. This is used to define our shapes width and height. As this value is equal for both we will be drawing a square. "*Gosu::Color::GREEN"* defines the color of our shape. "*ZOrder::TOP"* is then used to reference the module we created in step 5. We have told Ruby to render this shape on the upper most value of our 2d game window. "*mode=:default*" is then used to define our blending mode. Your development space should look as follows.

The draw function in ruby whilst using Gosu is a function that tells Ruby what to draw on the physical screen. All code in here will be rendered for the user to experience.

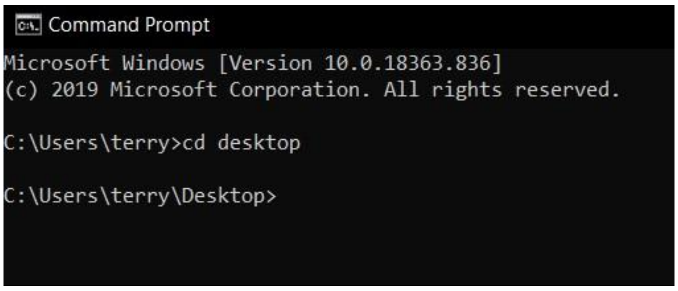

File Edit Selection View Go Run Terminal Help
tutorial.rb - Code - Visual Studio Code
tutorial.rb
C: > Users > terry > Desktop > tutorial.rb

```ruby
WIDTH = 400
HEIGHT = 500
SHAPE_DIM = 50

class GameWindow < Gosu::Window
    def initialize
        super WIDTH, HEIGHT, false
        self.caption = "Shape Moving"

        @shape_y = HEIGHT / 2
        @shape_x = WIDTH / 2

    end

    def draw
        Gosu.draw_rect(@shape_x, @shape_y, SHAPE_DIM, SHAPE_DIM, Gosu::Color::GREEN, ZOrder::TOP, mode=:default)

    end

end
window = GameWindow.new
window.show
```

Figure 7. (Above) Code with the completed draw function.

8. Testing our program

Iterative testing is important to develop a working bug free program. Now that we have functioning code open up the CMD and navigate to the directory you saved your Ruby file in. In this example we saved our file on the desktop to make this step a little quicker. Type the following commands to navigate to the save directory and run your program



```
Microsoft Windows [Version 10.0.18363.836]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\terry>cd desktop

C:\Users\terry\Desktop>
```

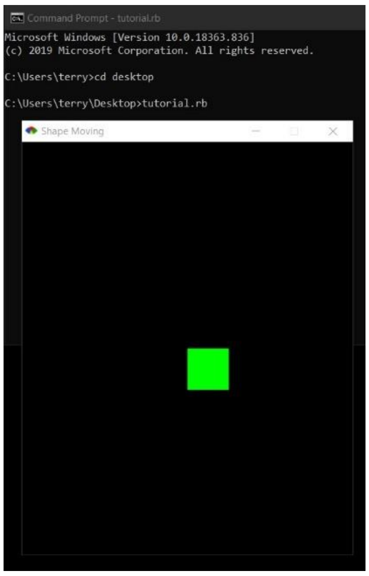
Figure 8. (Above) Image of the CMD with commands used to navigate to our desktop.

cd desktop

"cd" navigates to a location within the current directory. Next run the following command.

tutorial.rb

The next command instructs the CMD to run your ruby file. If you have followed along with this tutorial you should be presented with your game window. This can be seen in figure 9. Notice now the square we have drawn is slightly of center. This is because the "x" and "y" coordinates of the shape start at the top left corner and Ruby has alleged that with our starting position. A way to mitigate this could be to minus



```
Microsoft Windows [Version 10.0.18363.836]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\terry>cd desktop

C:\Users\terry\Desktop>tutorial.rb
```

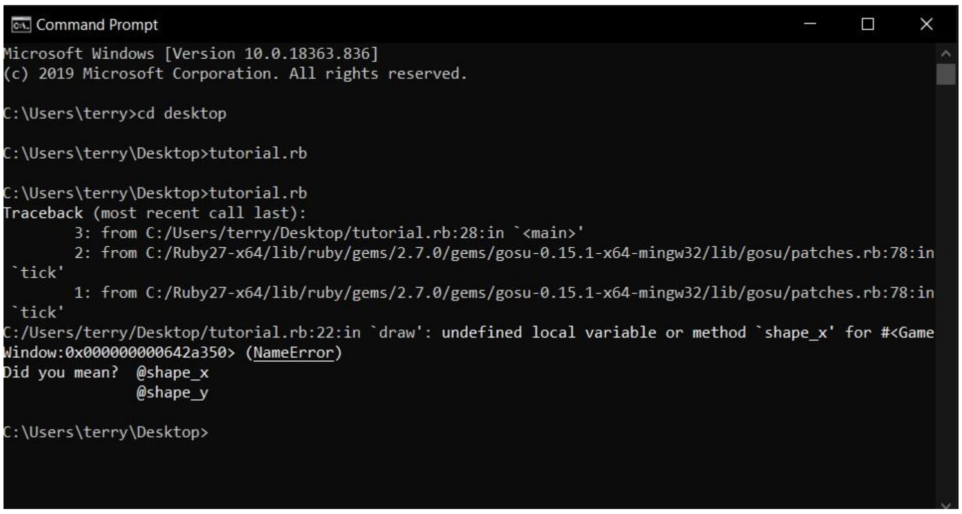
Figure 9. (Right) The resulting game window running.

half of the "SHAPE_DIM" value from both the *"@x_shape"* and *"@y_shape" variables as follows.*

"@x_shape – 25". This step will not be used in the tutorial however.

If your window does not display you may be presented with an error message in the CMD. An example of this has been provided in figure 10 (Bellow)

When we look more closely, we see specifics are given. This message is indicating an issue on line 22 with one of our instance variables, specifically *"@shape_x". The CMD then offers suggestions to help fix this issue. In the case of this example a "@" was removed from or code. This resulted in the following error.*



```
Microsoft Windows [Version 10.0.18363.836]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\terry>cd desktop

C:\Users\terry\Desktop>tutorial.rb

C:\Users\terry\Desktop>tutorial.rb
Traceback (most recent call last):
        3: from C:/Users/terry/Desktop/tutorial.rb:28:in `<main>'
        2: from C:/Ruby27-x64/lib/ruby/gems/2.7.0/gems/gosu-0.15.1-x64-mingw32/lib/gosu/patches.rb:78:in
 `tick'
        1: from C:/Ruby27-x64/lib/ruby/gems/2.7.0/gems/gosu-0.15.1-x64-mingw32/lib/gosu/patches.rb:78:in
 `tick'
C:/Users/terry/Desktop/tutorial.rb:22:in `draw': undefined local variable or method `shape_x' for #<Game
Window:0x000000000642a350> (NameError)
Did you mean?  @shape_x
               @shape_y

C:\Users\terry\Desktop>
```

Figure 10. (Above) A simulated error in our program.

If you run into problems like this take careful note of the line number and compare your code to the code issued in this tutorial.

9. Customization and experimentation

Before we move on lets take a further look at the code we have written. If you change our global constants note the changes in our program. Try changing the *"WIDTH"* and *"HEIGHT"* constants we defined and re-run the program. Notice the window will change size accordingly. The shape we have drawn responds in a similar manner to our game window. Try increasing and decreasing the value of "SHAPE_DIM" and observe the programs response.

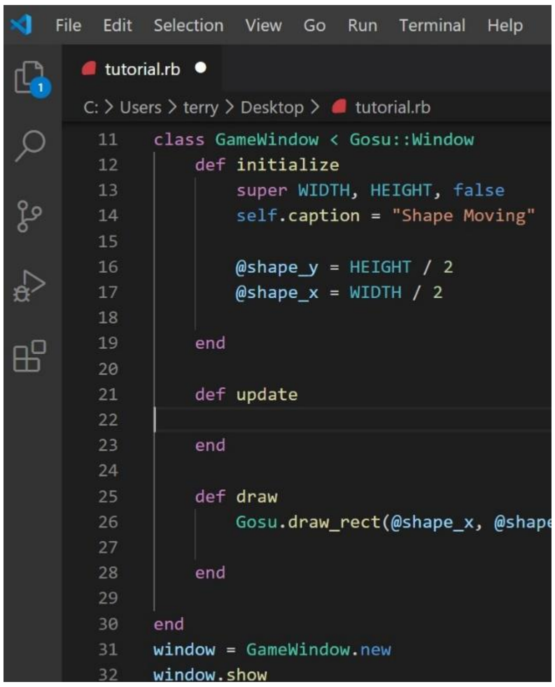
Similar experimentation can be done within the draw function. Try changing the color of your square by replacing *"GREEN"* with *"BLUE"*. Note the color of our shape changes when the program is re-run using CMD.

10. Set up our update method.

The next step is to add some user input to dynamically control our shape according to shifting instance variables. The following code is treated differently in Gosu. The "update" method is called 60 times a second and is responsible for processing all work done within the window. Include the following code in your file.

def update

end



```
class GameWindow < Gosu::Window
    def initialize
        super WIDTH, HEIGHT, false
        self.caption = "Shape Moving"

        @shape_y = HEIGHT / 2
        @shape_x = WIDTH / 2

    end

    def update

    end

    def draw
        Gosu.draw_rect(@shape_x, @shape

    end

end
window = GameWindow.new
window.show
```

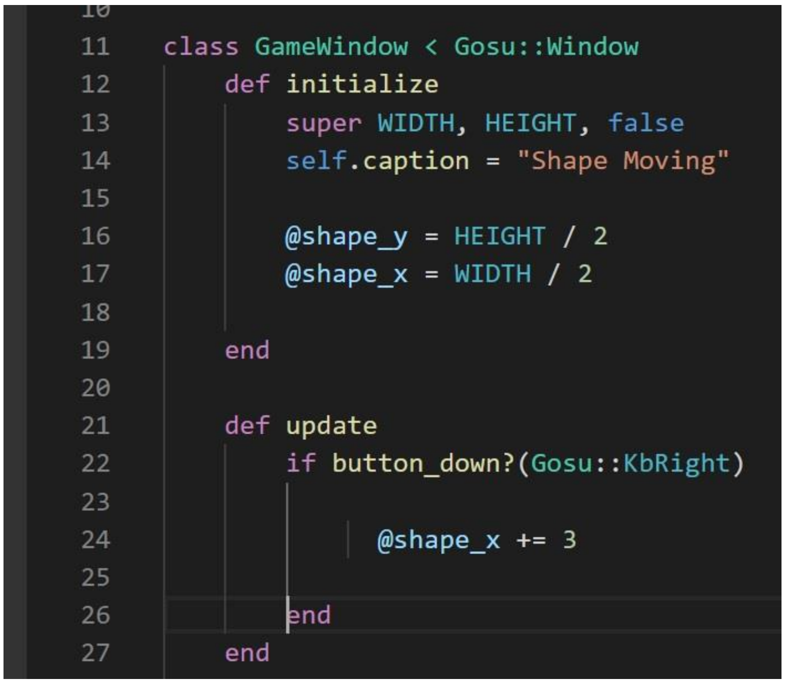
Figure 11. (Above) Addition of our update method.

11. Response from keyboard input

A common way of dealing with user input is by using inbuilt Gosu functions and if statement. An if statement is a conditional statement that allows a block of code to be executed so long as following conditions are met. Include the following code within the update method of your program. Be aware of code indentation as well as brackets and special characters.

```
if button_down?(Gosu::KBRight)
  @shape_x += 3
end
```

The above code is responsible for detecting user input from the defined key. In this case "(*Gosu::KbRight)"* is making reference to the right keyboard key. The "*@shape_x += 3*" tells ruby that every time the program detects input from the user on the right arrow key the program should increment the "x" position of our shape by 3. Please see figure 12 as reference.



```
class GameWindow < Gosu::Window
    def initialize
        super WIDTH, HEIGHT, false
        self.caption = "Shape Moving"

        @shape_y = HEIGHT / 2
        @shape_x = WIDTH / 2

    end

    def update
        if button_down?(Gosu::KbRight)

            @shape_x += 3

        end
    end
```

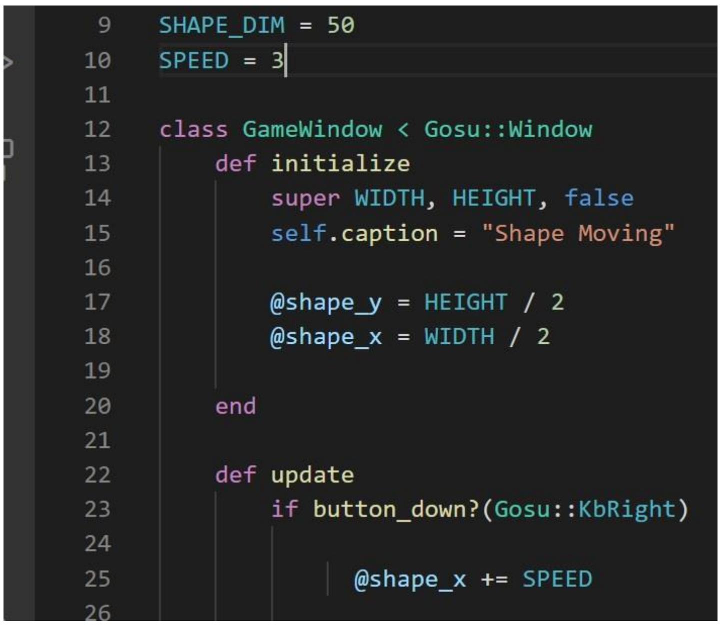
Figure 12. (Above) The addition of our listener to detect user input from the right arrow key.

Note the amount by which we are incrementing the "x" position of our shape. It is a soft coded value. We are going to have to repeat this line multiple times in order to ensure movement of our shape in all directions. This is a perfect opportunity to create another global constant.

Replace the *"3"* with *"SPEED"* and create a global constant up with the others we created in step 3 using the following code.

SPEED = 3

Your code should now look like that depicted in figure 13.



```
SHAPE_DIM = 50
SPEED = 3

class GameWindow < Gosu::Window
    def initialize
        super WIDTH, HEIGHT, false
        self.caption = "Shape Moving"

        @shape_y = HEIGHT / 2
        @shape_x = WIDTH / 2

    end

    def update
        if button_down?(Gosu::KbRight)

            @shape_x += SPEED
```

Figure 13. (Above) Amendment of the soft coded value "3" and the addition of a new global constant SPEED = 3.

Re-run your program using the methods outlined in step 8 and observe the behavior of our shape when the right arrow key is pressed. Note how the object moves past the maximum limit defined by our game window and is lost.

12. Creating limits around our game window

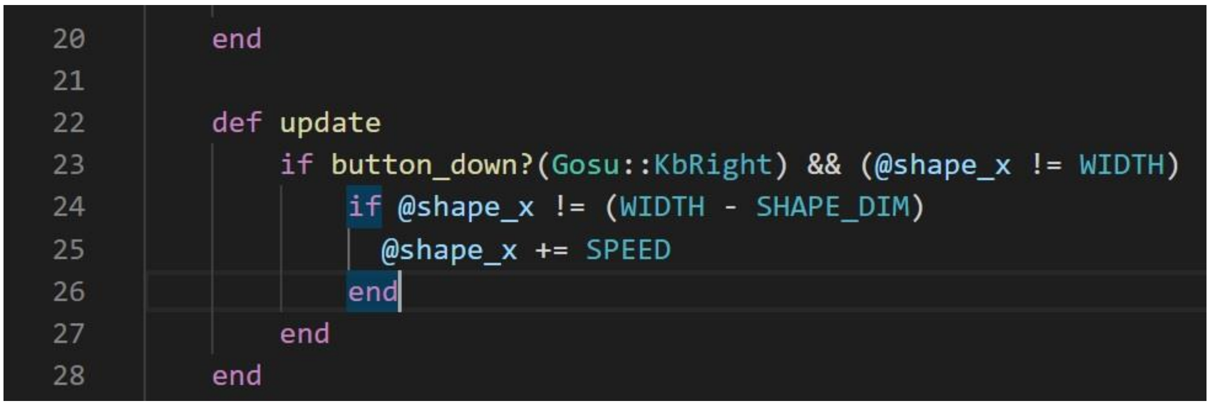
To keep our shape within the game window we need to include one more if statement.

Include the following code in your program. Pay special attention to the placement of the additional if statement as well as the && statement, as this code needs to be carefully integrated into our existing code in order for it to have the desired effect. Please reference figure 14 for specific placement of this code.

&& (@shape_x != WIDTH)

if @shape_x != (WIDTH – SHAPE_DIM)

end



```ruby
    end

    def update
        if button_down?(Gosu::KbRight) && (@shape_x != WIDTH)
            if @shape_x != (WIDTH - SHAPE_DIM)
              @shape_x += SPEED
            end
        end
    end
```

Figure 14. (Above) Integration of an additional if and && statement to create a barrier on the right side of our game window.

The "*&&"* symbol is telling Ruby that both bracketed conditions need to be true in order for the subsequent block of code to be performed. Ruby reads this as "If the right key is pressed and the x position of the shape does not equal the total width of the window".

The following if statement tells ruby to only increment our object by *"SPEED"* if the "x" position of our code is not a value that is equal to the width of our screen minus the shape dimensions.
This statement subsequently creates a virtual barrier within our game window that is maintained by comparing our shapes "x" position, minus the shapes size in relation to the total width of our window.

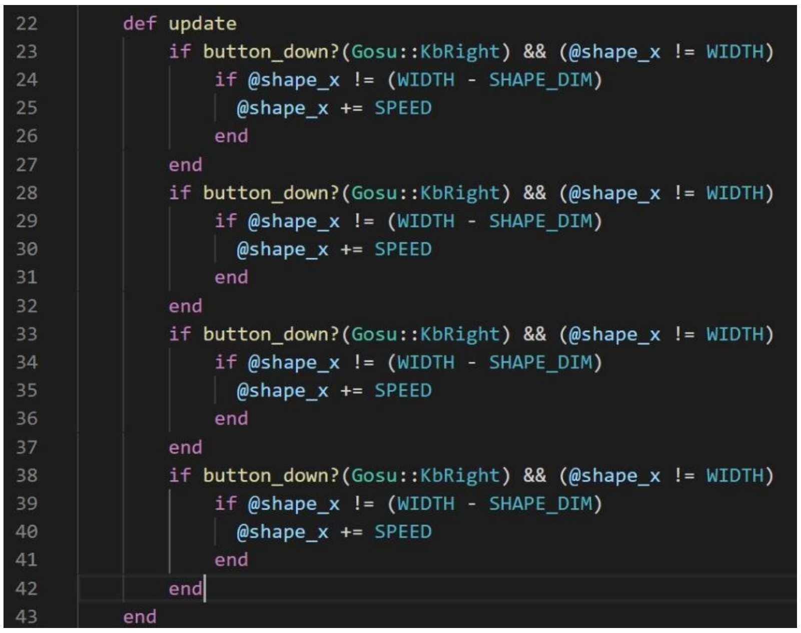
Re-run your program and observe the results. Our shape is now limited on the right-hand side of our game window.

13. Modularity in program design.

The way code in figure 14(Above) has been written, allows us to fully test the functionality of the code before moving onto the next section. By creating a fully functional block we are able then copy and paste this function to compute the rest of the movement in the remaining directions. This is a concept related to good program design.

14. Remaining directions and limits.

To get our shape to behave in the same way but in various other directions we can copy, paste and modify the following code. Copy lines 22 through 28 that were written in section 12 and past them bellow. Make sure all *"end"* lines are maintained and that the code is pasted within the update method. Your update method should now look like figure 15. If you re-run the program, you'll notice that the only change has been the speed your shape moves. This is because Ruby is executing the same command 4 times. If you set your *"SPEED"* to 3 the speed will now be processed at 12.



```
22      def update
23          if button_down?(Gosu::KbRight) && (@shape_x != WIDTH)
24              if @shape_x != (WIDTH - SHAPE_DIM)
25                @shape_x += SPEED
26              end
27          end
28          if button_down?(Gosu::KbRight) && (@shape_x != WIDTH)
29              if @shape_x != (WIDTH - SHAPE_DIM)
30                @shape_x += SPEED
31              end
32          end
33          if button_down?(Gosu::KbRight) && (@shape_x != WIDTH)
34              if @shape_x != (WIDTH - SHAPE_DIM)
35                @shape_x += SPEED
36              end
37          end
38          if button_down?(Gosu::KbRight) && (@shape_x != WIDTH)
39              if @shape_x != (WIDTH - SHAPE_DIM)
40                @shape_x += SPEED
41              end
42          end
43      end
```

Figure 15. (Above) Our movement statements copied four times.

Next we will modify this code in order to get the remaining directions functional. Replace the code in your update method with the code displayed bellow (1 through 3). Being sure to leave your original if statement on line 23 through 27 untouched. As this code is functional and handles our right direction.

1. If statement to move the shape towards the left side of the game window

```
        if button_down?(Gosu::KbLeft) && (@shape_x != WIDTH - WIDTH)
            if @shape_x != (WIDTH + SHAPE_DIM) && (@shape_x > 0)
              @shape_x -= SPEED
            end
        end
```

1. If statement to move the shape towards the top of the game window

```
        if button_down?(Gosu::KbUp) && (@shape_y != HEIGHT - HEIGHT)
            if @shape_x != (HEIGHT + SHAPE_DIM) && (@shape_y > 0)
              @shape_y -= SPEED
            end
        end
```

2. If statement to move the shape towards the bottom of the game window

```
        if button_down?(Gosu::KbDown)
            if @shape_y < (HEIGHT - SHAPE_DIM) && (@shape_y != HEIGHT)
              @shape_y += SPEED
            end
        end
```

If we look more closely at code 1 through 3 above we see slight variations in the way the window limits are defined. This is because the value of each limit changes based on the direction and needs to be calculated differently. The change is sign that modifies the value of our instance variables "*@shape_x*" and "I@shape_y" also differ. Either adding or subtracting the values depending on the direction of travel. Imagine points within a grid. As you add and subtract from your point you can navigate through all points.

15. Test your program.

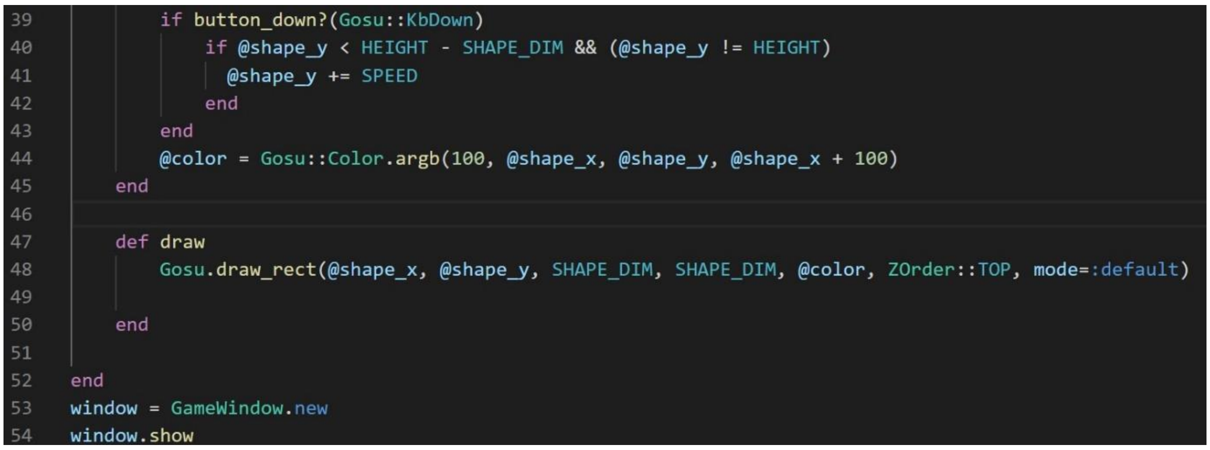
The code written between steps 2 and 14 has created a fully interactive square a user can manipulate using keyboard input. Test your program as per the steps in section 8. Should errors arise try reading the CMD line to pinpoint the mistake. If your shape does not behave as expected a common source for errors is within the update method. Compare with section 14.

16. Adding some variation in color based on "x" and "y" positions.

The color of our shape up to this point has been dictated by the "*Gosu::Color::GREEN*" line within the "*Gosu.draw_rect*" line.
Include the following lines of code in your update method under your final button-down statement. Please note this codes position outside of the code responsible for movement but inside the update method.

@color = Gosu::Color.argb(100, @shape_x, @shape_y, @shape_x + 100)

Then include the new instance variable we just defined in the "*Gosu.draw_rect*" line. Replacing "*Gosu::Color::GREEN*".



```
39          if button_down?(Gosu::KbDown)
40              if @shape_y < HEIGHT - SHAPE_DIM && (@shape_y != HEIGHT)
41                @shape_y += SPEED
42              end
43          end
44          @color = Gosu::Color.argb(100, @shape_x, @shape_y, @shape_x + 100)
45      end
46
47      def draw
48          Gosu.draw_rect(@shape_x, @shape_y, SHAPE_DIM, SHAPE_DIM, @color, ZOrder::TOP, mode=:default)
49
50      end
51
52  end
53  window = GameWindow.new
54  window.show
```

Figure 16. (Above) Included code for dynamic colour response according to x, y position of the rendered shape.

Re-run the program and observe the response. As the shape moves around the shape the color updates. By changing the variables and values within our instance variable deceleration on line 44 we can change the colors that appear to the user as interaction occurs.

As "*@color*" is written within the update method the instance variable is being updated 60 times a second with the values of "*@shape_x*" and "*@shape_y*". This then updates the "*Gosu.draw_rect*" color declaration. This is an example of how more dynamic user manipulated shapes can be created in Gosu.

Finally replace each of the values in "*@color*" with "*Gosu::random(0, 255)*" and observe the effects.

17. Further reading.

Further information relating to Ruby and the Gosu library can be found here.

https://www.rubydoc.info/github/gosu/gosu