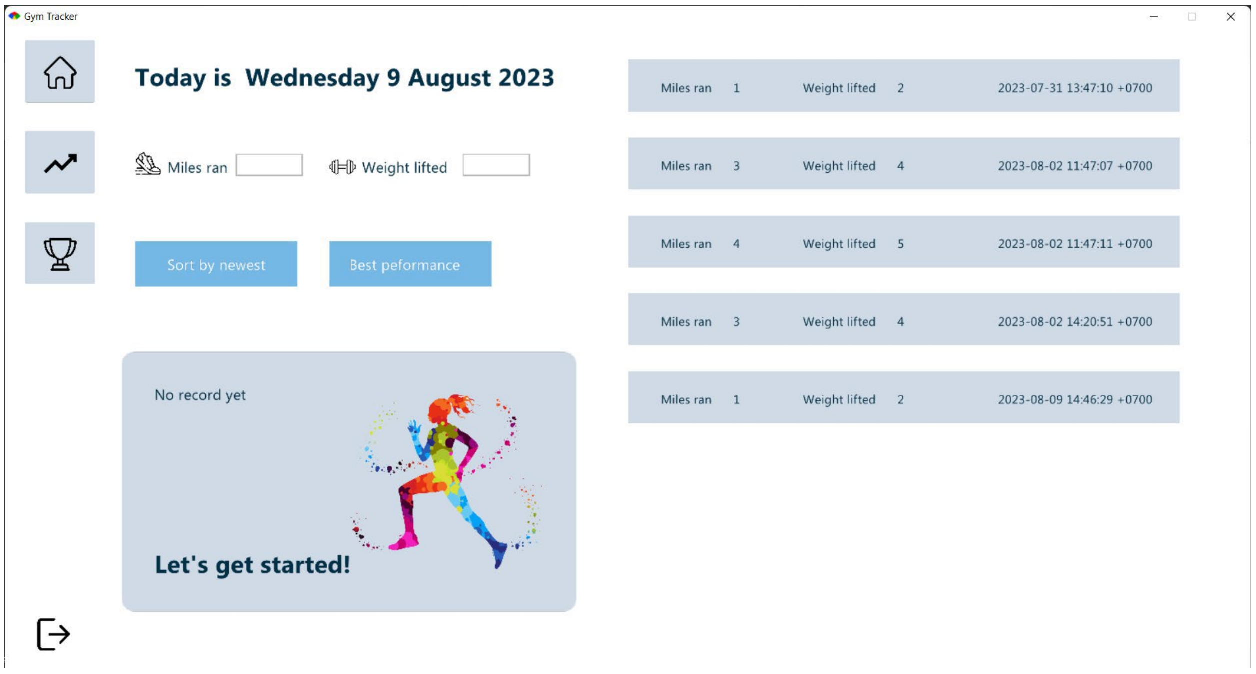
Tutorial on creating a basic Gosu interactive GUI and more

Gosu GUI has always been a challenging and important aspect in a ruby program. In this report, I will attempt to demonstrate how to create an interactive Gosu GUI in which we can input text, draw data and utilize merge sort to sort data.

1. Introduction to GUIs and Gosu:

A computer user had to input lengthy commands in the past to do anything. This command-line interface had the potential to be effective and was undoubtedly simple. However, the commands weren't very user-friendly because they were frequently complicated and hard to recall. A GUI eliminates the need to input lengthy stretches of code in order to accomplish goals. Users can easily complete their jobs by swapping out this code for icons

Therefore, Gosu was introduced to adhere to ruby, helping to add an GUI to the otherwise boring terminal. In this report, we will be creating this menu:



Miles ran	1	Weight lifted	2	2023-07-31 13:47:10 +0700
Miles ran	3	Weight lifted	4	2023-08-02 11:47:07 +0700
Miles ran	4	Weight lifted	5	2023-08-02 11:47:11 +0700
Miles ran	3	Weight lifted	4	2023-08-02 14:20:51 +0700
Miles ran	1	Weight lifted	2	2023-08-09 14:46:29 +0700

2. Setting Up the Development Environment:

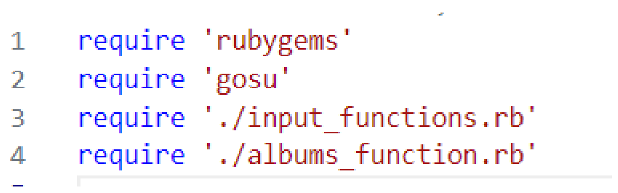
Required tools and software: Gosu library, a code editor (e.g., Visual Studio Code), and Ruby.

3. Basics of Gosu:

Gosu is a versatile 2D graphics library and game development framework that's particularly popular among Ruby developers. It offers a wide range of features and capabilities for creating interactive applications, games, and graphical user interfaces (GUIs)

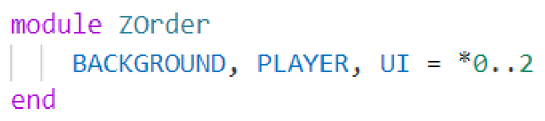
4. Creating the Basic Interactive GUI:

1/First we would need to code in all of the required gems for the ruby to run.



```ruby
require 'rubygems'
require 'gosu'
require './input_functions.rb'
require './albums_function.rb'
```

2/We can establish the order of the images drawn using Module



```ruby
module ZOrder
    BACKGROUND, PLAYER, UI = *0..2
end
```

1. The module ZOrder statement begins the definition of a Ruby module named ZOrder.

2. Inside the module, three constants are defined: BACKGROUND, PLAYER, and UI.

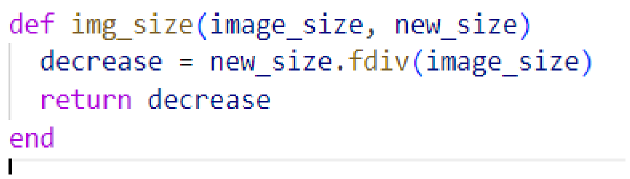
3. These constants are assigned values using the splat operator (*) and a range of integers from 0 to 2.

4. The BACKGROUND constant is assigned the value 0, the PLAYER constant is assigned the value 1, and the UI constant is assigned the value 2.

In many game development frameworks, including Gosu, these z-order constants are used to determine the drawing order of graphical elements. For example, elements with a higher z-order value will be drawn on top of elements with a lower z-order value. This helps achieve the desired layering and visual hierarchy in the graphics of an application.

3/img_size def

This is used to resize the images for better convenience.



```ruby
def img_size(image_size, new_size)
  decrease = new_size.fdiv(image_size)
  return decrease
end
```

This code defines a method named img_size that calculates the necessary decrease factor to resize an image from its original image_size to a desired new_size. Let's break down each part of the code:

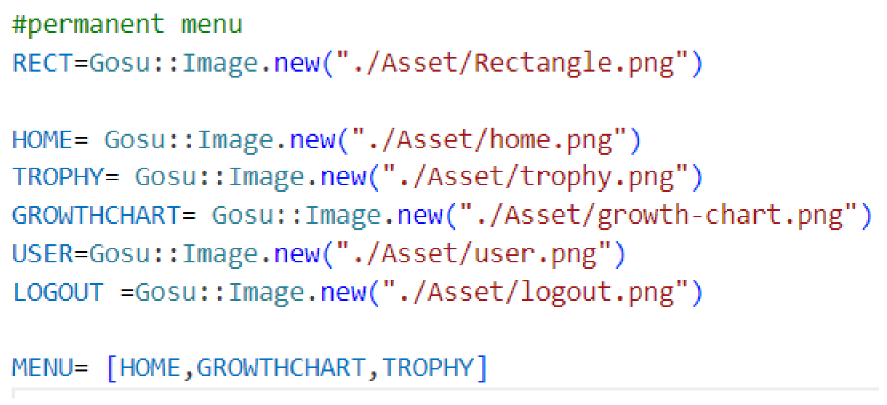
def img_size(image_size, new_size): This line defines the method img_size with two parameters: image_size and new_size. The method is intended to calculate the decrease factor required to resize an image from its original size to the new size.

decrease = new_size.fdiv(image_size): In this line, the method calculates the decrease factor by dividing the new_size by the image_size. The .fdiv method is used for floating-point division, ensuring that the result is a floating-point number.

For example, if the image_size is 1000 and the new_size is 500, then decrease would be calculated as 0.5. This indicates that the image needs to be scaled down by a factor of 0.5 (or 50%) to fit the new size.

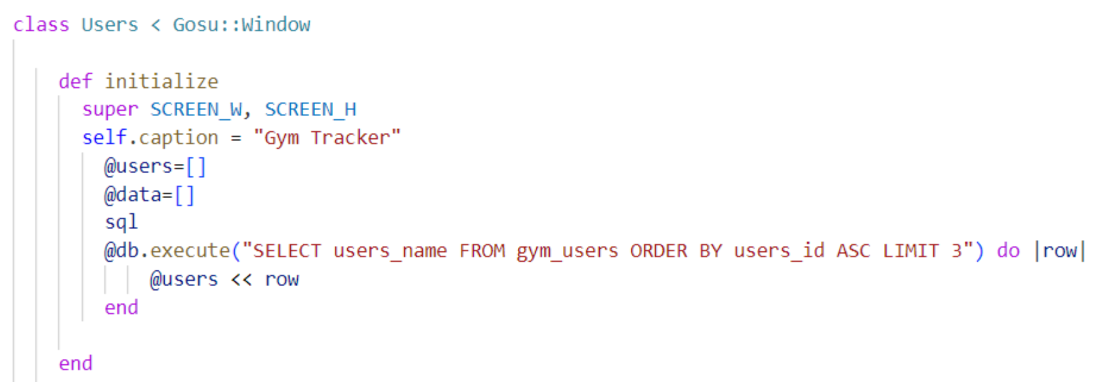
return decrease: This line returns the calculated decrease factor as the result of the method. The decrease value indicates how much the image needs to be scaled down to achieve the desired new size.

4/ Establish the constants to be used later



```
#permanent menu
RECT=Gosu::Image.new("./Asset/Rectangle.png")

HOME= Gosu::Image.new("./Asset/home.png")
TROPHY= Gosu::Image.new("./Asset/trophy.png")
GROWTHCHART= Gosu::Image.new("./Asset/growth-chart.png")
USER=Gosu::Image.new("./Asset/user.png")
LOGOUT =Gosu::Image.new("./Asset/logout.png")

MENU= [HOME,GROWTHCHART,TROPHY]
```

5/ Program the Gosu class



```
class Users < Gosu::Window

  def initialize
    super SCREEN_W, SCREEN_H
    self.caption = "Gym Tracker"
      @users=[]
      @data=[]
      sql
      @db.execute("SELECT users_name FROM gym_users ORDER BY users_id ASC LIMIT 3") do |row|
          @users << row
      end

  end
```

class Users < Gosu::Window: This line begins the definition of the Users class, which inherits from the Gosu::Window class. This indicates that the Users class will create a graphical window using the Gosu library.

def initialize: This line defines the constructor method initialize for the Users class. The method is called when an instance of the class is created using the new method.

super SCREEN_W, SCREEN_H: This line calls the constructor of the superclass Gosu::Window and passes in the width and height (SCREEN_W and SCREEN_H) of the window. This initializes the graphical window with the specified dimensions.

self.caption = "Gym Tracker": This line sets the title or caption of the window to "Gym Tracker," which will be displayed in the window's title bar.

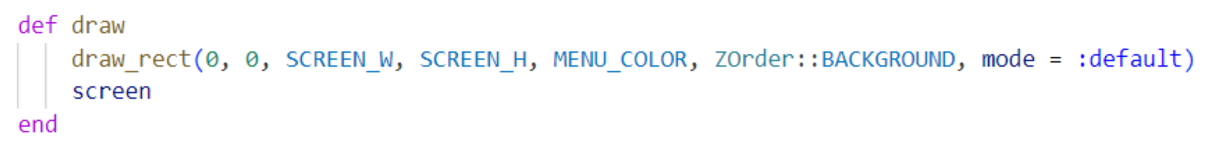
@users=[]: This line initializes an instance variable @users as an empty array. This variable will be used to store user names retrieved from a database query.

@data=[]: This line initializes another instance variable @data as an empty array. The purpose of this variable is to communicate between the Gosu and SQL tables, to draw out the texts

sql: This method initializes the sql which will be used later on

The rest of the code corelates to the user selection screen which is related but is not mentioned in this report.

6/Drawing the screen

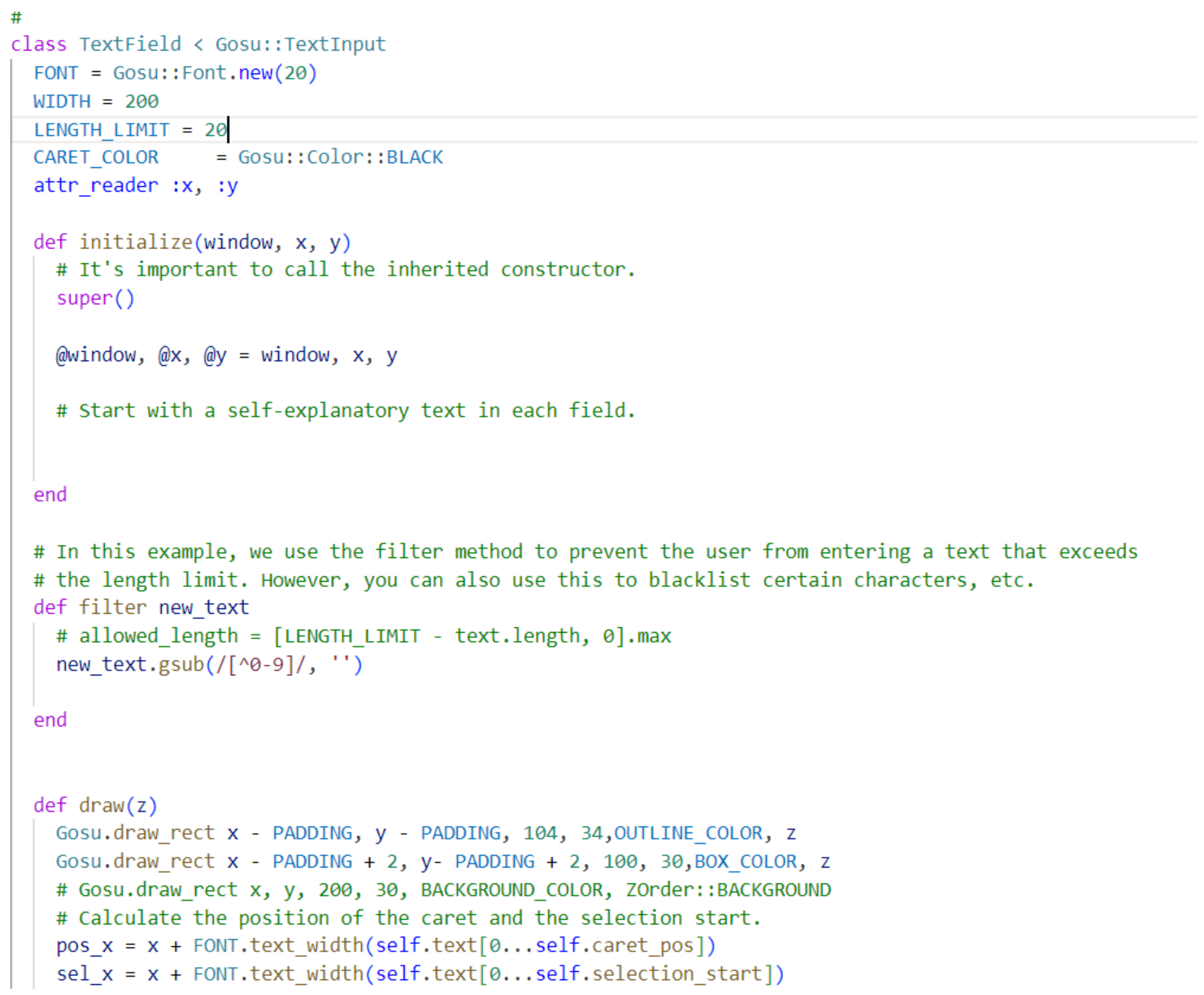


```
def draw
    draw_rect(0, 0, SCREEN_W, SCREEN_H, MENU_COLOR, ZOrder::BACKGROUND, mode = :default)
    screen
end
```

draw_rect(0, 0, SCREEN_W, SCREEN_H, MENU_COLOR, ZOrder::BACKGROUND, mode = :default): This line draws a rectangle that covers the entire window's canvas. The parameters specify the position (0, 0) as the top-left corner, the dimensions SCREEN_W and SCREEN_H as the width and height of the rectangle, MENU_COLOR as the fill color, ZOrder::BACKGROUND as the drawing layer, and mode = :default as the drawing mode. This rectangle effectively serves as the background color for the entire window.

screen: This line calls the screen method, which is where all of the drawn data is stored

7/Establish the text input



```ruby
#
class TextField < Gosu::TextInput
  FONT = Gosu::Font.new(20)
  WIDTH = 200
  LENGTH_LIMIT = 20
  CARET_COLOR      = Gosu::Color::BLACK
  attr_reader :x, :y

  def initialize(window, x, y)
    # It's important to call the inherited constructor.
    super()

    @window, @x, @y = window, x, y

    # Start with a self-explanatory text in each field.


  end

  # In this example, we use the filter method to prevent the user from entering a text that exceeds
  # the length limit. However, you can also use this to blacklist certain characters, etc.
  def filter new_text
    # allowed_length = [LENGTH_LIMIT - text.length, 0].max
    new_text.gsub(/[^0-9]/, '')

  end


  def draw(z)
    Gosu.draw_rect x - PADDING, y - PADDING, 104, 34,OUTLINE_COLOR, z
    Gosu.draw_rect x - PADDING + 2, y- PADDING + 2, 100, 30,BOX_COLOR, z
    # Gosu.draw_rect x, y, 200, 30, BACKGROUND_COLOR, ZOrder::BACKGROUND
    # Calculate the position of the caret and the selection start.
    pos_x = x + FONT.text_width(self.text[0...self.caret_pos])
    sel_x = x + FONT.text_width(self.text[0...self.selection_start])
```

The provided code defines a custom class named `TextField` that inherits from `Gosu::TextInput`. This class represents a text input field that can be used in a graphical application. It enables the user to input and display text within a defined area. Let's break down the code to understand its functionality and components:

1. `TextField` class: This class is defined to represent a custom text input field.

2. `FONT = Gosu::Font.new(20)`: A constant `FONT` is created with a font size of 20. This font will be used to render the text within the text field.

3. `WIDTH = 200`: The constant `WIDTH` defines the width of the text field.

4. `LENGTH_LIMIT = 20`: The constant `LENGTH_LIMIT` specifies the maximum number of characters that can be entered in the text field.

5. `CARET_COLOR = Gosu::Color::BLACK`: The constant `CARET_COLOR` sets the color of the caret (text cursor) to black.

6. `attr_reader :x, :y`: An attribute reader method is defined for the `x` and `y` coordinates of the text field.

7. `def initialize(window, x, y)`: The constructor method `initialize` is defined for the `TextField` class. It takes three parameters: `window`, `x`, and `y`. The `window` parameter represents the graphical window, and `x` and `y` represent the position of the text field.

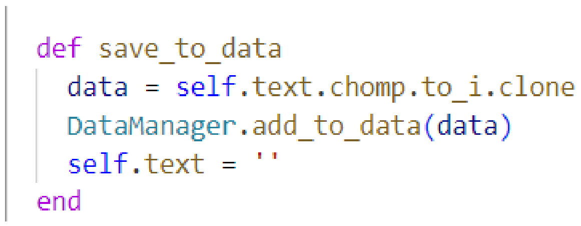
8. `super()`: This line calls the constructor of the superclass (`Gosu::TextInput`) to initialize the inherited functionality for handling text input.

9. `@window, @x, @y = window, x, y`: Instance variables are assigned the values of the parameters for later use.

10. `filter new_text`: This method is used to filter the text input based on the specified criteria. In this case, it uses a regular expression to allow only numeric characters (0-9) and remove any other characters.

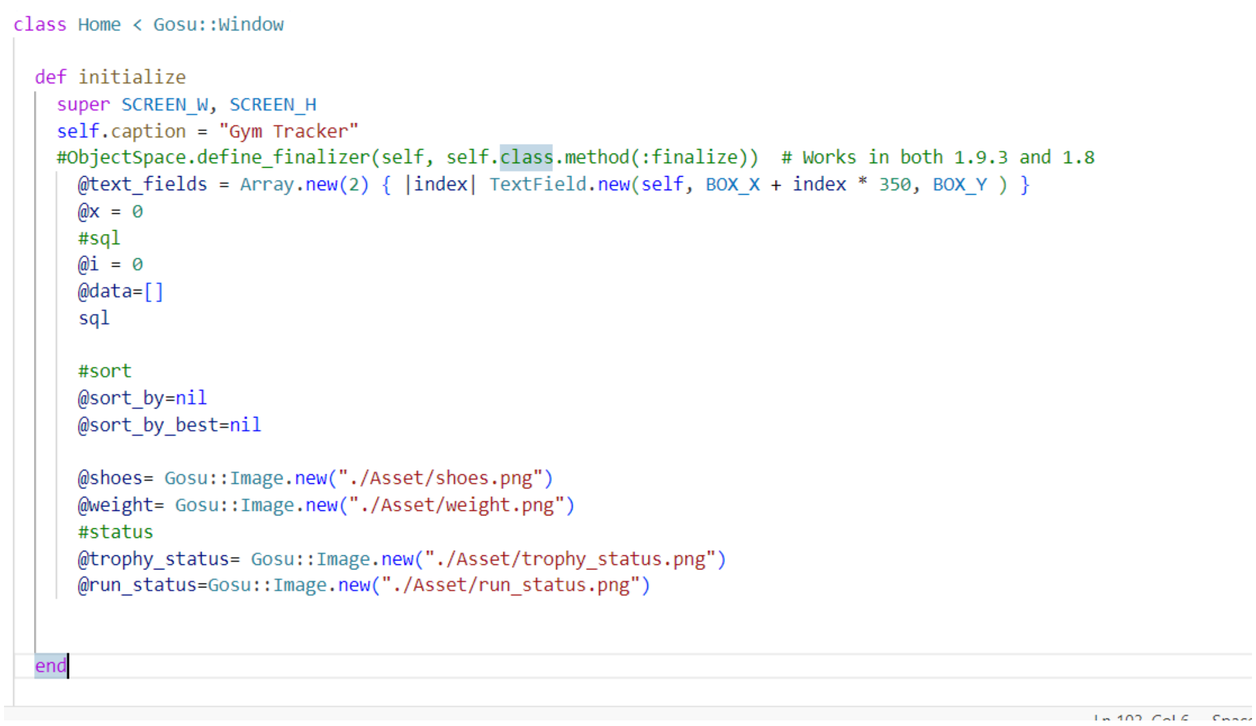
The provided code snippet is part of a larger class definition that represents a text input field in a graphical application. The `TextField` class provides methods for filtering input, rendering the text field on the screen, moving the caret cursor, and more. It's designed to be used as part of a larger GUI where users can input and manipulate text.

For now, there are a few things you should remember for later on:

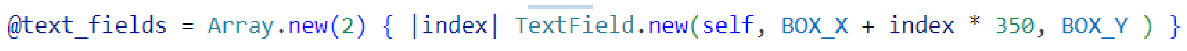


```ruby
def save_to_data
  data = self.text.chomp.to_i.clone
  DataManager.add_to_data(data)
  self.text = ''
end
```

The def save_to_data clone to text and saves it to the data manager. We'll see it in action in our other Gosu window.



```
class Home < Gosu::Window

  def initialize
    super SCREEN_W, SCREEN_H
    self.caption = "Gym Tracker"
    #ObjectSpace.define_finalizer(self, self.class.method(:finalize))  # Works in both 1.9.3 and 1.8
      @text_fields = Array.new(2) { |index| TextField.new(self, BOX_X + index * 350, BOX_Y ) }
      @x = 0
      #sql
      @i = 0
      @data=[]
      sql

      #sort
      @sort_by=nil
      @sort_by_best=nil

      @shoes= Gosu::Image.new("./Asset/shoes.png")
      @weight= Gosu::Image.new("./Asset/weight.png")
      #status
      @trophy_status= Gosu::Image.new("./Asset/trophy_status.png")
      @run_status=Gosu::Image.new("./Asset/run_status.png")


  end
```



```
  @text_fields = Array.new(2) { |index| TextField.new(self, BOX_X + index * 350, BOX_Y ) }
```

@text_fields: This is an instance variable that will hold an array of TextField objects. This array is used to manage and keep track of multiple text input fields.

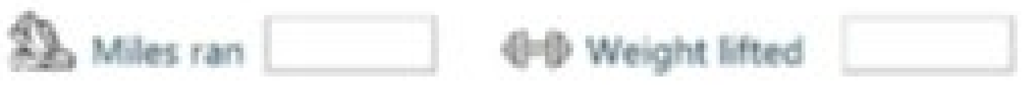
Array.new(2) { |index| TextField.new(self, BOX_X + index * 350, BOX_Y) }:

Array.new(2): This part creates a new array with a length of 2. The array will contain two elements.

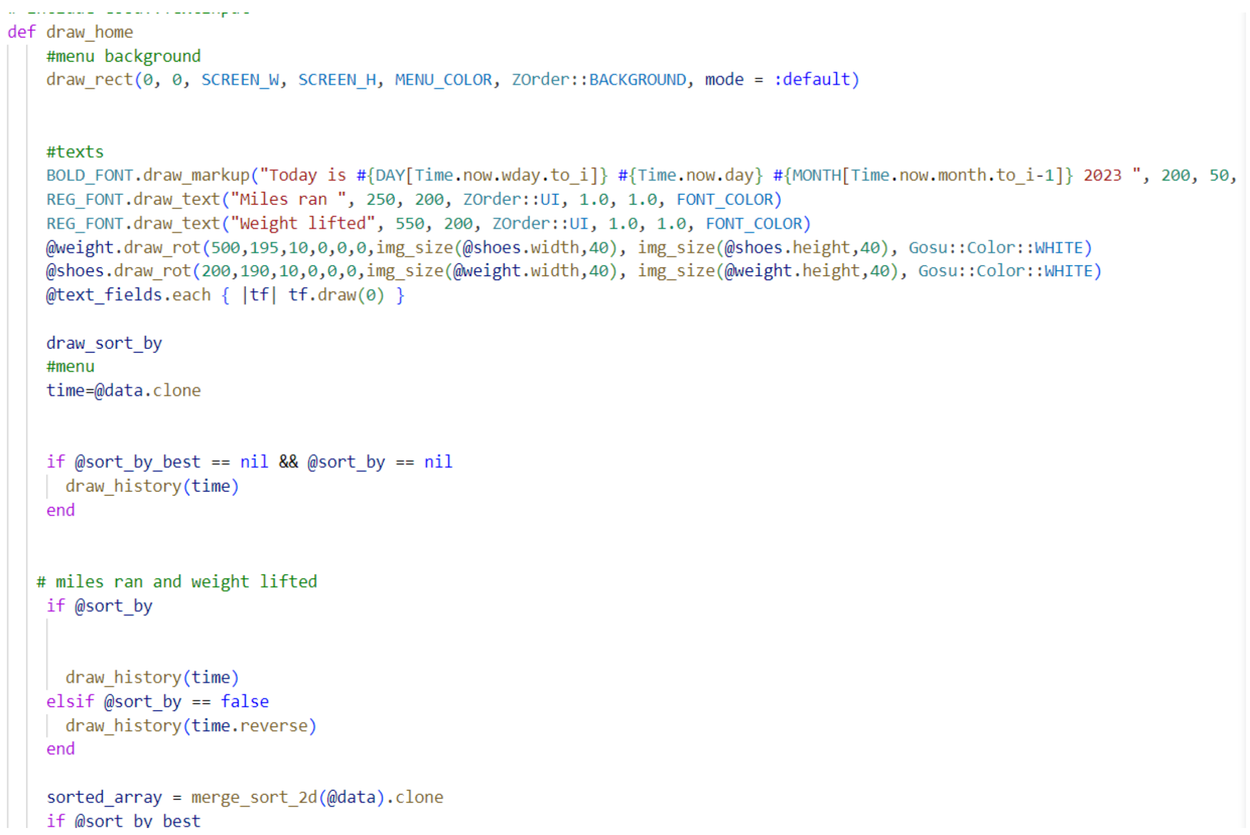
{ |index| TextField.new(self, BOX_X + index * 350, BOX_Y) }: This is a block that's executed for each element of the array being created. The index variable is the index of the current element being created (0 for the first element, 1 for the second element).

TextField.new(self, BOX_X + index * 350, BOX_Y): Inside the block, a new TextField object is created for each element of the array. The self parameter represents the graphical window (presumably an instance of Gosu::Window), which is passed to the TextField constructor. The BOX_X + index * 350 calculation sets the X-coordinate position of the text field based on the index, creating some spacing between them horizontally. BOX_Y sets the common Y-coordinate position for both text fields.

Using this, we drew



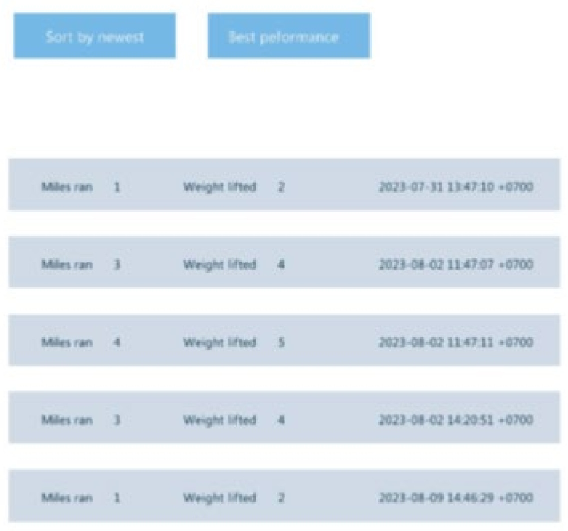
Finall, we move onto drawing the history.



```
def draw_home
    #menu background
    draw_rect(0, 0, SCREEN_W, SCREEN_H, MENU_COLOR, ZOrder::BACKGROUND, mode = :default)


    #texts
    BOLD_FONT.draw_markup("Today is #{DAY[Time.now.wday.to_i]} #{Time.now.day} #{MONTH[Time.now.month.to_i-1]} 2023 ", 200, 50,
    REG_FONT.draw_text("Miles ran ", 250, 200, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
    REG_FONT.draw_text("Weight lifted", 550, 200, ZOrder::UI, 1.0, 1.0, FONT_COLOR)
    @weight.draw_rot(500,195,10,0,0,0,img_size(@shoes.width,40), img_size(@shoes.height,40), Gosu::Color::WHITE)
    @shoes.draw_rot(200,190,10,0,0,0,img_size(@weight.width,40), img_size(@weight.height,40), Gosu::Color::WHITE)
    @text_fields.each { |tf| tf.draw(0) }

    draw_sort_by
    #menu
    time=@data.clone


    if @sort_by_best == nil && @sort_by == nil
      draw_history(time)
    end


   # miles ran and weight lifted
    if @sort_by


      draw_history(time)
    elsif @sort_by == false
      draw_history(time.reverse)
    end

    sorted_array = merge_sort_2d(@data).clone
    if @sort_by_best
```

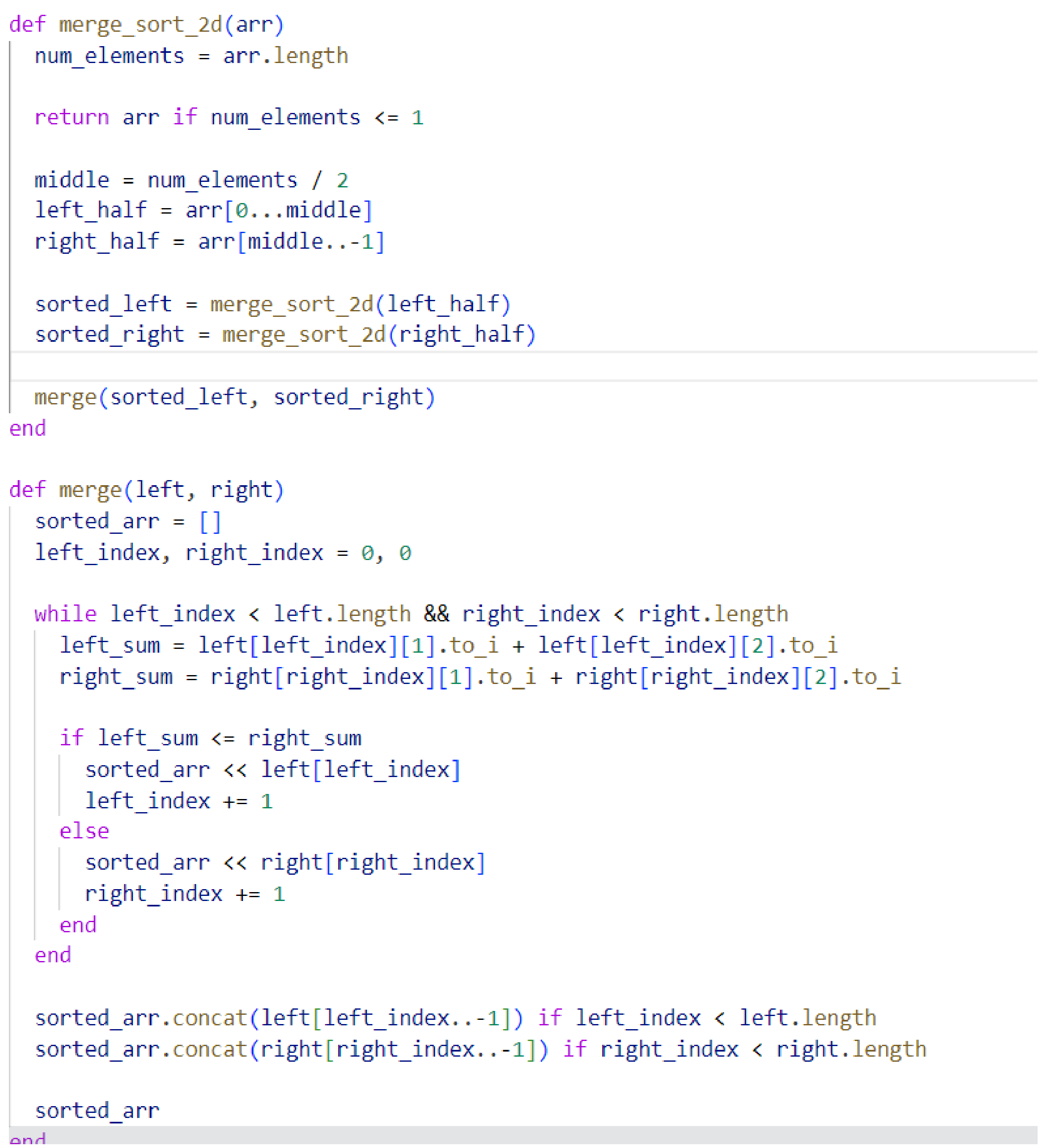
The date is drawn using Time.now and the constant array that contains all the strings for each days. Therefore it can accurately draw the current date and time.

The history are drawn by an if Boolean. It is used to toggle between buttons.



```
Miles ran   1     Weight lifted   2     2023-07-31 13:47:10 +0700

Miles ran   3     Weight lifted   4     2023-08-02 11:47:07 +0700

Miles ran   4     Weight lifted   5     2023-08-02 11:47:11 +0700

Miles ran   3     Weight lifted   4     2023-08-02 14:20:51 +0700

Miles ran   1     Weight lifted   2     2023-08-09 14:46:29 +0700
```

The data are sorted by best using merge sort, which is additionally in gym_functions



```ruby
def merge_sort_2d(arr)
  num_elements = arr.length

  return arr if num_elements <= 1

  middle = num_elements / 2
  left_half = arr[0...middle]
  right_half = arr[middle..-1]

  sorted_left = merge_sort_2d(left_half)
  sorted_right = merge_sort_2d(right_half)

  merge(sorted_left, sorted_right)
end

def merge(left, right)
  sorted_arr = []
  left_index, right_index = 0, 0

  while left_index < left.length && right_index < right.length
    left_sum = left[left_index][1].to_i + left[left_index][2].to_i
    right_sum = right[right_index][1].to_i + right[right_index][2].to_i

    if left_sum <= right_sum
      sorted_arr << left[left_index]
      left_index += 1
    else
      sorted_arr << right[right_index]
      right_index += 1
    end
  end

  sorted_arr.concat(left[left_index..-1]) if left_index < left.length
  sorted_arr.concat(right[right_index..-1]) if right_index < right.length

  sorted_arr
end
```

merge_sort_2d(arr): This function takes a 2D array arr as input and returns a sorted 2D array. The sorting is based on the sum of specific elements within each row of the 2D array.

num_elements = arr.length: Calculates the number of elements (rows) in the input array.

return arr if num_elements <= 1: If the array has only one element or is empty, it is already considered sorted, so it's returned as is.

middle = num_elements / 2: Calculates the middle index of the array to divide it into two halves.

left_half = arr[0...middle]: Divides the input array into the left half, containing elements from index 0 to middle - 1.

right_half = arr[middle..-1]: Divides the input array into the right half, containing elements from index middle to the end.

sorted_left = merge_sort_2d(left_half): Recursively applies the merge sort algorithm to the left half.

sorted_right = merge_sort_2d(right_half): Recursively applies the merge sort algorithm to the right half.

merge(sorted_left, sorted_right): Calls the merge function to combine and sort the left and right halves.

merge(left, right): This function takes two sorted arrays, left and right, and merges them together while sorting them based on the sum of specific elements in each row.

sorted_arr = []: Initializes an empty array to hold the sorted result.

left_index, right_index = 0, 0: Initializes index variables for tracking the positions in the left and right arrays.

The while loop compares elements from both the left and right arrays and selects the element with the smaller sum of the specified elements.

left_sum = left[left_index][1].to_i + left[left_index][2].to_i: Calculates the sum of the elements at specific indices within the row in the left array.

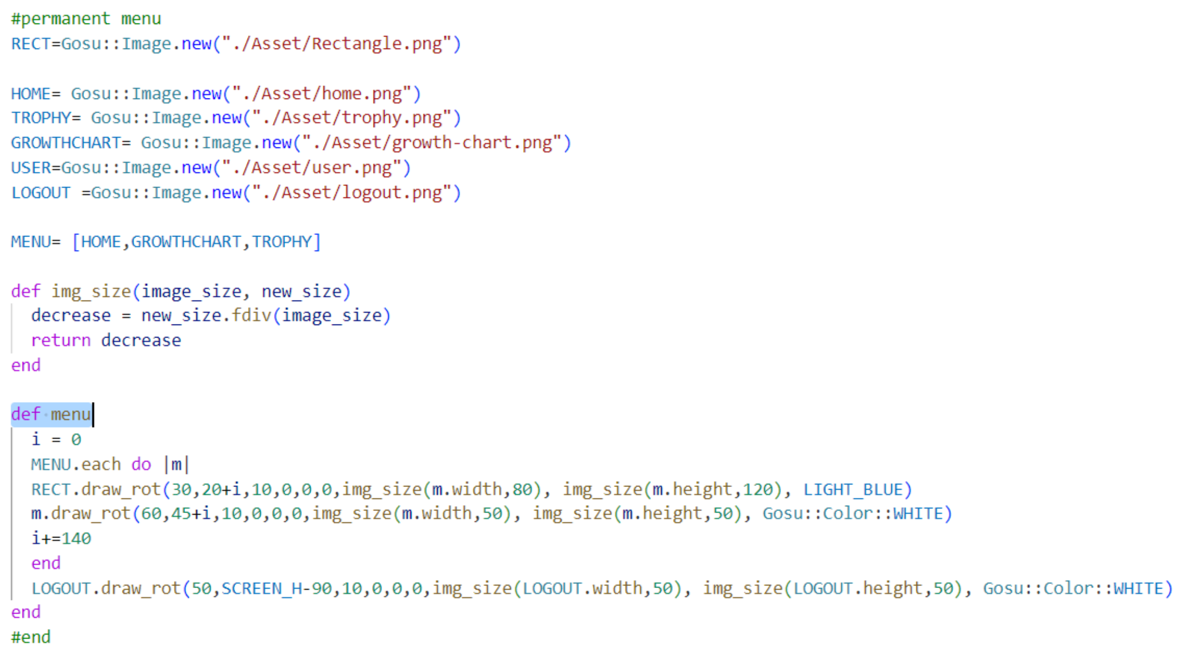
right_sum = right[right_index][1].to_i + right[right_index][2].to_i: Calculates the sum of the elements at specific indices within the row in the right array.

The loop compares left_sum and right_sum and appends the corresponding row to sorted_arr, updating the respective index.

After the loop, any remaining elements from the left or right array are concatenated to sorted_arr.

Finally, the sorted and merged array sorted_arr is returned.

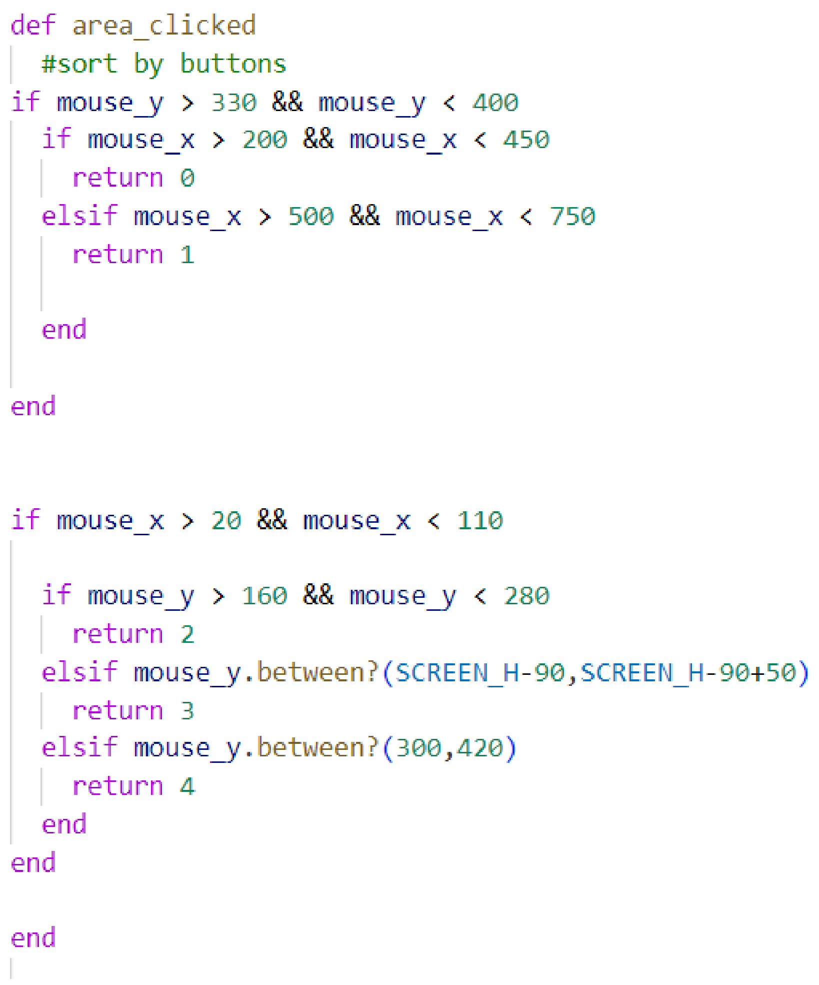
Drawing the menu



```
#permanent menu
RECT=Gosu::Image.new("./Asset/Rectangle.png")

HOME= Gosu::Image.new("./Asset/home.png")
TROPHY= Gosu::Image.new("./Asset/trophy.png")
GROWTHCHART= Gosu::Image.new("./Asset/growth-chart.png")
USER=Gosu::Image.new("./Asset/user.png")
LOGOUT =Gosu::Image.new("./Asset/logout.png")

MENU= [HOME,GROWTHCHART,TROPHY]

def img_size(image_size, new_size)
  decrease = new_size.fdiv(image_size)
  return decrease
end

def menu
  i = 0
  MENU.each do |m|
  RECT.draw_rot(30,20+i,10,0,0,0,img_size(m.width,80), img_size(m.height,120), LIGHT_BLUE)
  m.draw_rot(60,45+i,10,0,0,0,img_size(m.width,50), img_size(m.height,50), Gosu::Color::WHITE)
  i+=140
  end
  LOGOUT.draw_rot(50,SCREEN_H-90,10,0,0,0,img_size(LOGOUT.width,50), img_size(LOGOUT.height,50), Gosu::Color::WHITE)
end
#end
```

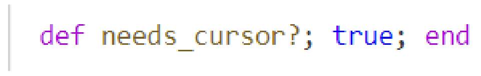
We utilized each loop and increment them to draw the menu

5. Adding Interactivity:



```
def area_clicked
  #sort by buttons
if mouse_y > 330 && mouse_y < 400
  if mouse_x > 200 && mouse_x < 450
    return 0
  elsif mouse_x > 500 && mouse_x < 750
    return 1

  end

end


if mouse_x > 20 && mouse_x < 110

  if mouse_y > 160 && mouse_y < 280
    return 2
  elsif mouse_y.between?(SCREEN_H-90,SCREEN_H-90+50)
    return 3
  elsif mouse_y.between?(300,420)
    return 4
  end
end

end
```

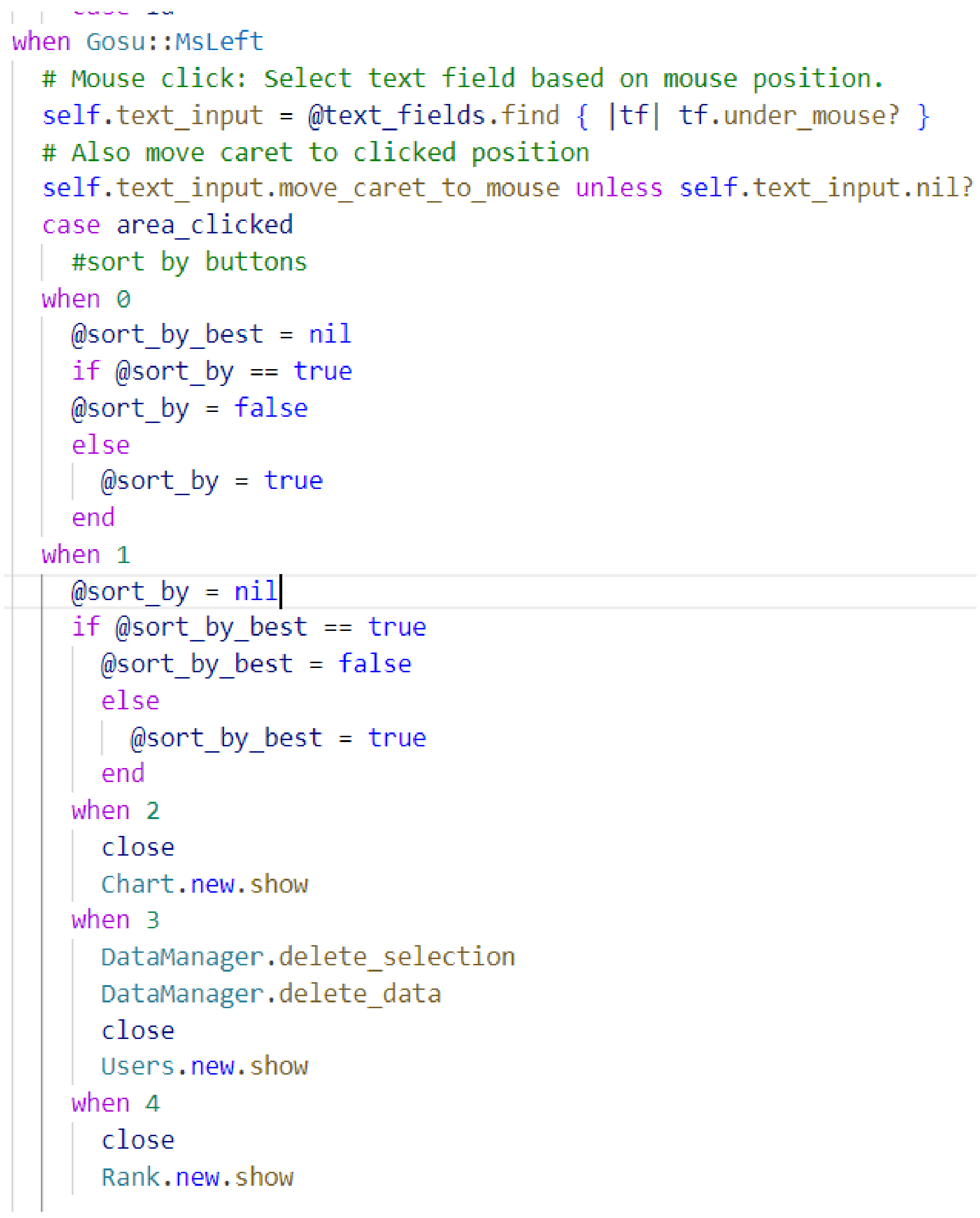
We’ll first establish the area clicked for our case function in the button down def



```
def needs_cursor?; true; end
```

First we need the def need_cursor? To activate cursor censor

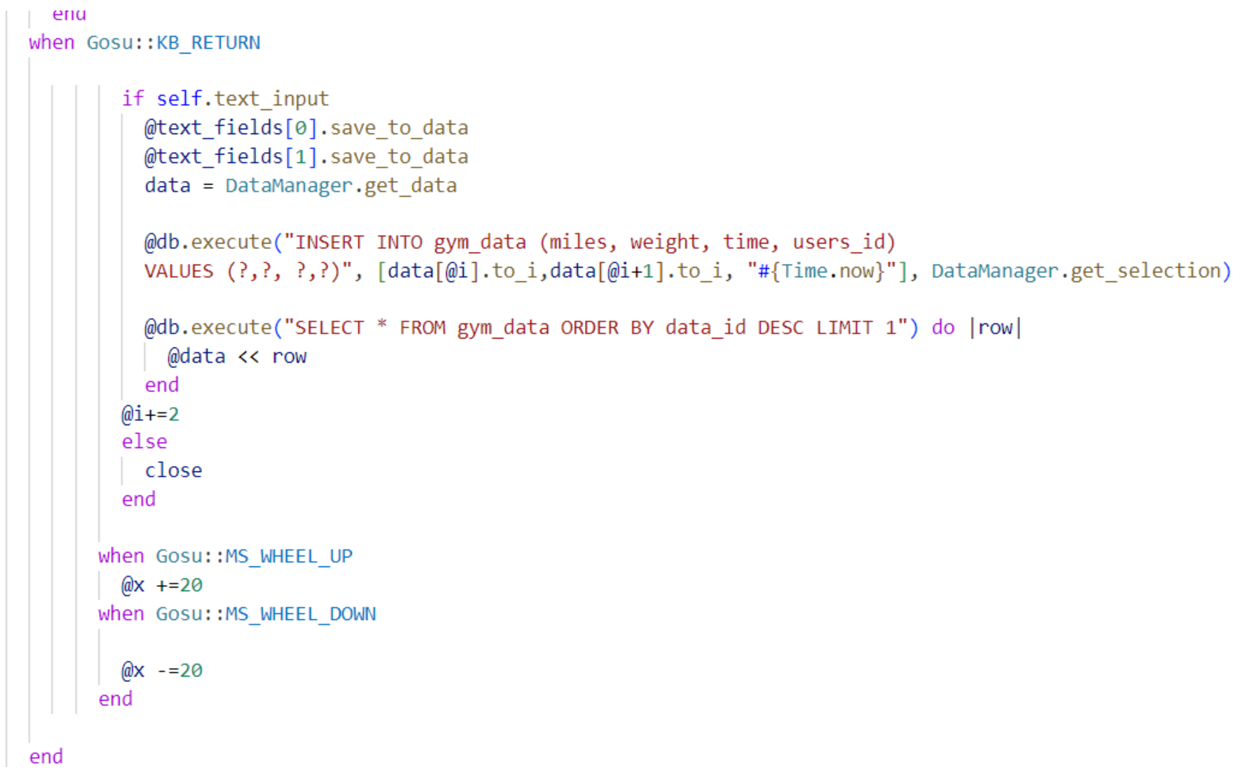
Next is our button down def



```
when Gosu::MsLeft
  # Mouse click: Select text field based on mouse position.
  self.text_input = @text_fields.find { |tf| tf.under_mouse? }
  # Also move caret to clicked position
  self.text_input.move_caret_to_mouse unless self.text_input.nil?
  case area_clicked
    #sort by buttons
  when 0
    @sort_by_best = nil
    if @sort_by == true
    @sort_by = false
    else
      @sort_by = true
    end
  when 1
    @sort_by = nil
    if @sort_by_best == true
      @sort_by_best = false
      else
        @sort_by_best = true
      end
    when 2
      close
      Chart.new.show
    when 3
      DataManager.delete_selection
      DataManager.delete_data
      close
      Users.new.show
    when 4
      close
      Rank.new.show
```

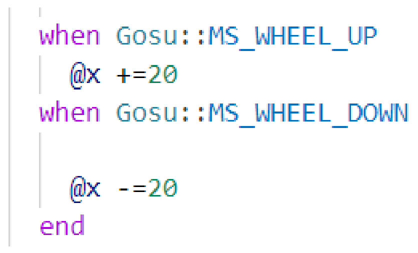
When you clicked on sort by best and sort alternates from nil to true and false. The other buttons are used to have the interactive menu, each of the closes the program and opens a new one.

Next is the interactive button ENTER



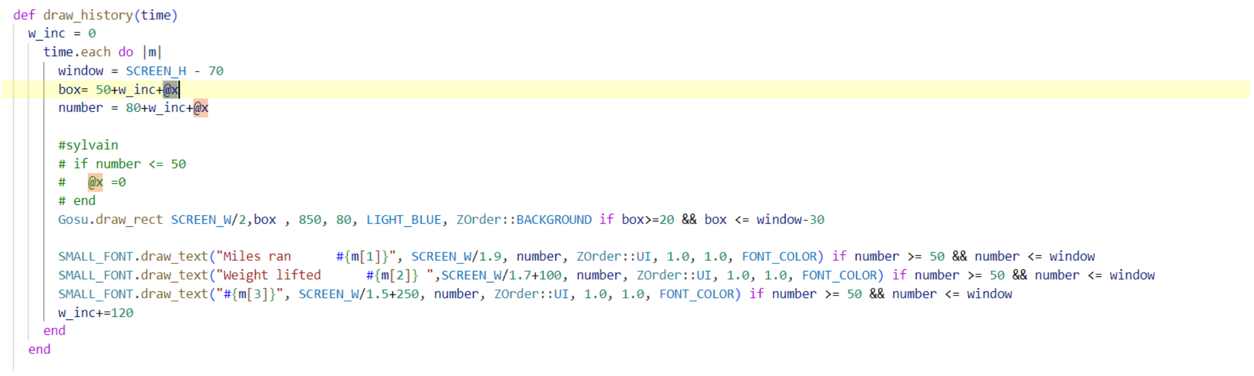
```
  end
when Gosu::KB_RETURN

        if self.text_input
          @text_fields[0].save_to_data
          @text_fields[1].save_to_data
          data = DataManager.get_data

          @db.execute("INSERT INTO gym_data (miles, weight, time, users_id)
          VALUES (?,?, ?,?)", [data[@i].to_i,data[@i+1].to_i, "#{Time.now}"], DataManager.get_selection)

          @db.execute("SELECT * FROM gym_data ORDER BY data_id DESC LIMIT 1") do |row|
            @data << row
          end
        @i+=2
        else
          close
        end

      when Gosu::MS_WHEEL_UP
        @x +=20
      when Gosu::MS_WHEEL_DOWN

        @x -=20
      end

end
```

When we press KB_RETURN, it will automatically save to the data array module. Then it is get and established as data, then the sql execute the data to insert it into sql table and consequently @data table for real-time update.



```
when Gosu::MS_WHEEL_UP
  @x +=20
when Gosu::MS_WHEEL_DOWN

  @x -=20
end
```

This helps scrolls the data drawn up and down



```
def draw_history(time)
  w_inc = 0
    time.each do |m|
      window = SCREEN_H - 70
      box= 50+w_inc+@x
      number = 80+w_inc+@x

      #sylvain
      # if number <= 50
      #   @x =0
      # end
      Gosu.draw_rect SCREEN_W/2,box , 850, 80, LIGHT_BLUE, ZOrder::BACKGROUND if box>=20 && box <= window-30

      SMALL_FONT.draw_text("Miles ran      #{m[1]}", SCREEN_W/1.9, number, ZOrder::UI, 1.0, 1.0, FONT_COLOR) if number >= 50 && number <= window
      SMALL_FONT.draw_text("Weight lifted      #{m[2]} ",SCREEN_W/1.7+100, number, ZOrder::UI, 1.0, 1.0, FONT_COLOR) if number >= 50 && number <= window
      SMALL_FONT.draw_text("#{m[3]}", SCREEN_W/1.5+250, number, ZOrder::UI, 1.0, 1.0, FONT_COLOR) if number >= 50 && number <= window
      w_inc+=120
    end
  end
```

Scrolling will push the Y axis of the history up and down.

8. References:

https://ruby-doc.org/

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

9. Conclusion:

In conclusion, this simple Gosu GUI tutorial introduced you to the basics of creating an interactive graphical user interface (GUI) using the Gosu library. We covered several key aspects of developing a basic GUI, from setting up the environment to handling user input and rendering elements on the screen.