CUSTOM PROJECT REPORT

COS10009 – INTRODUCTION TO PROGRAMMING

TA QUANG TUNG

104222196

Overview

This report outlines the main structure and features of my extended GUI music player and goes in-depth into the implementation of one of the critical UI elements of the program – the scrollable content box. This report will be divided into four sections. The first section summarizes the key features of the program. The second section lists the tools I used to build the music player. The third section provides a structure chart to illustrate the main structure of the program. The final section delves into the code and logic behind the scrollable content box, which appears thrice in the music player. The last section aims to provide Gosu beginners the basic idea behind the scrollable content box UI element so that they can build a simple one for their program.

Section 1 – Key features

My extended GUI music player was designed to closely resemble a real music player (Spotify being a huge inspiration and the main source of reference). It includes a user-friendly, clean interface and contains most features seen in a modern music player. Users can create, view and maintain their music library, which can consist of albums and playlists. Albums and playlists are read from two text files, albums.txt and playlists.txt respectively. Users can create and delete as many playlists as they want, and they can add songs to playlists or change the playlists' cover image for easier identification.

The music player supports play, pause, skip, loop and volume control operations. It also lets users shuffle an album or a playlist to play tracks in randomized order. Tracks can be played automatically or manually when the user clicks on them. The player has a queue which can be filled by the user for one-time playback of songs, giving them the benefit of playing any song they want without having to switch playlists.

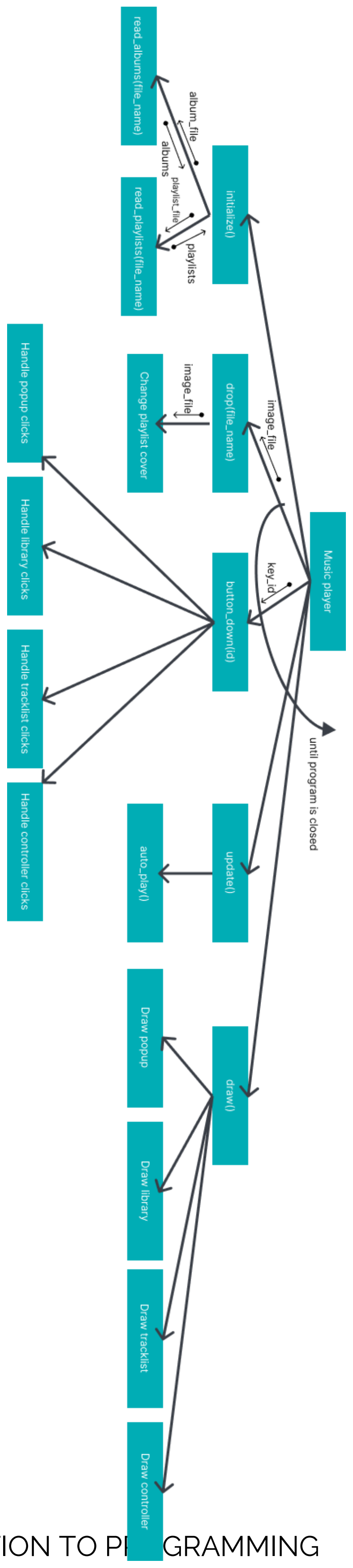
Section 2 – Languages and libraries

My extended music player is built entirely using the Gosu library in the Ruby programming language. Since Gosu is a very general-purpose and lightweight graphic library, it does not have any built-in UI features. All UI elements on the screen are created from scratch, with rendering and user interaction handled by Gosu.

The player also briefly uses the fileutils module built into Ruby. It provides a method to copy files from one directory to another. This is used exclusively for the feature that allows users to drag and drop an image file onto a playlist to change its cover image.

Section 3 – Structure chart

Below is a simple structure chart that illustrates the key structure of the music player program.



Section 4 – Implementation of the scrollable content box

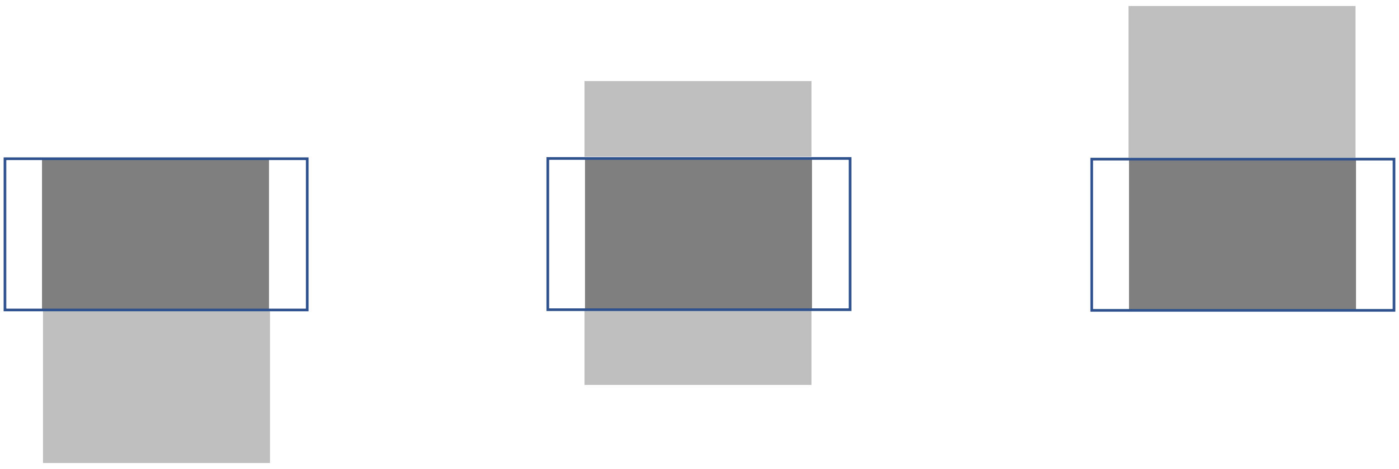
This section provides a detailed discussion on the logic behind and implementation of the scrollable content box (SCB), a UI element that appears in three different places in my music player. It also serves as a tutorial for Gosu beginners who want to implement a simple SCB in their own programs.

Motivation

Oftentimes, a window has to display content that exceeds its height or width. In these situations, only a portion of the content may be displayed at a time. This motivates a mechanism that lets the user move across the content to see the portion they want. For blocks of content involving discrete elements (a grid of cards, for example), a simple solution would be to group several elements into a page and then display one page at a time, with the user switching page as needed. However, for long and continuous content, a scrollable content box (SCB) may be necessary.

What is an SCB?

An SCB can be thought of as a window (whether the entire program window or just a region within it) through which the content is seen. It can have fixed or dynamic dimensions, as is required by the user. At one time, only a portion of the content is rendered within the window. The excess content is clipped out and is hidden away from the user. A scrollbar often comes attached to an SCB. The orientation of the scrollbar determines the scroll direction. Horizontal scrollbars let the user scroll left or right and vertical scrollbars let the user scroll up or down.



How an SCB renders its content, as the content is moved from top to bottom. Dark gray indicates content being rendered. Light gray indicates clipped content.

Building a simple SCB in Gosu

The basic features of an SCB can be easily implemented with the methods provided by Gosu. This tutorial will mainly focus on the vertically scrollable content box. However, the same ideas presented can be extended to horizontal scrolling.

Getting started

To get started, we will need a record to represent the box itself. For it, we will need the following class:

```ruby
class ScrollableContentBox
    attr_accessor :x, :y, :width, :height, :content_top, :content_height
    def initialize(x, y, w, h, ctop, cheight)
        @x = x
        @y = y
        @width = w
        @height = h
        @content_top = ctop
        @content_height = cheight
    end
end
```

The fields `x`, `y`, `width` and `height` represent the position and dimensions of the box. The field `content_top` represents the topmost y-coordinate at which content will be drawn. Think of it as the ceiling of the content block. Content will be drawn with respect to this `content_top` field. `content_height` represents the total height of the content within the box.

Drawing the content within the box

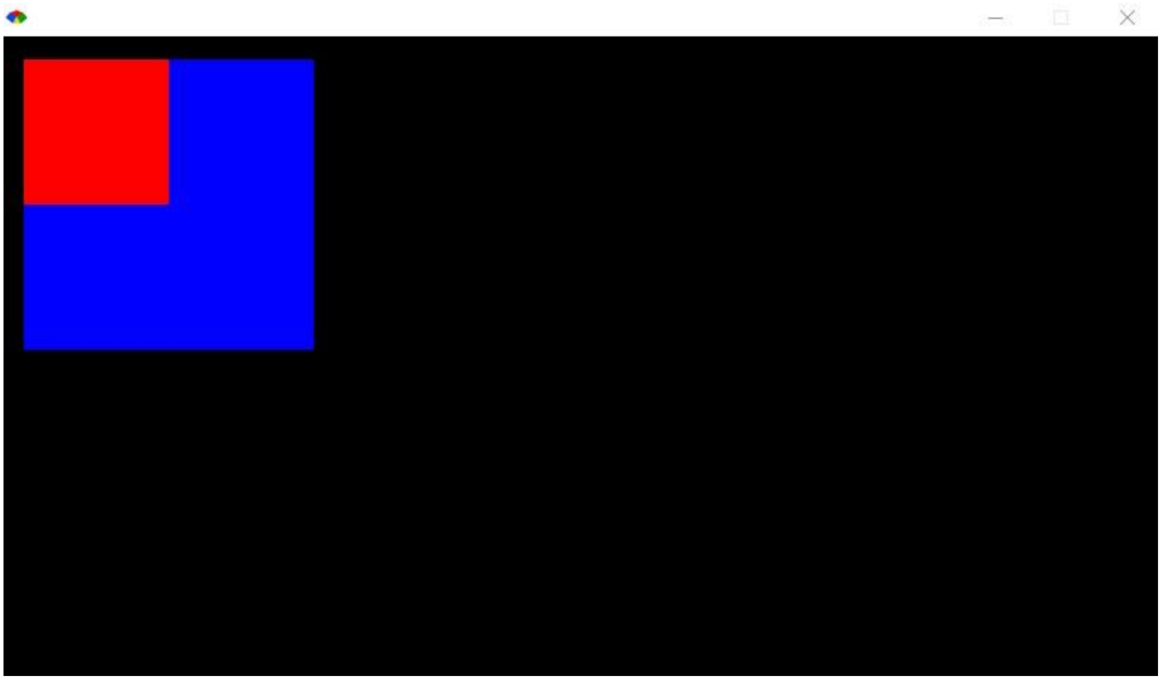
As mentioned previously, at any given time, only a portion of the content should be rendered while the rest is clipped out. This can be easily achieved within Gosu with the `Gosu.clip_to` function. As per the official 1.4.3 Gosu documentation, the function takes four parameters, `x`, `y`, `w`, `h`, representing the position and dimensions of the clipping area, which in this case is the SCB itself. It also takes a block containing draw function calls. Blocks in Ruby are similar to methods – but they are functions passed into methods themselves (GeeksforGeeks, 2022). Think of them as parameters to other methods. You might have used blocks before in functions such as `each`

```
myArray.each do |array_element|
    puts(array_element)
end
```

In the above example, everything from do to end is a block. This block takes as parameter an array element and prints that element into the console. Blocks received by `Gosu.clip_to` does not take any arguments.

Here is some sample code and output to illustrate the use of `Gosu.clip_to`, using the viewport class we created earlier:

```
class GameWindow < Gosu::Window
    def initialize()
        super(800, 450)
        @scrollbox = ScrollableContentBox.new(16, 16, 100, 100, 16, 0)
    end

    def draw_content()
        # Draw a square bigger than the scrollable content box (SCB)
        Gosu.draw_rect(16, 16, 200, 200, Gosu::Color::RED)
    end

    def draw() # This method is called every frame
        # For reference, the full content will be drawn in a different
color.
        Gosu.draw_rect(16, 16, 200, 200, Gosu::Color::BLUE)
        Gosu.clip_to(@scrollbox.x, @scrollbox.y, @scrollbox.width,
@scrollbox.height) do
            draw_content() # Only a portion of this content will be shown
        end
    end
end
GameWindow.new().show()
```



To make the content move up and down as we scroll, it should not be drawn at a fixed position within the SCB. Instead, it should be drawn with respect to `content_top`. Next, we will add code to change the value of `content_top` as we scroll the content.

Scrolling the content

There should ideally be two ways to scroll the content – either with the middle mouse wheel or with the scrollbar on the side of the SCB. We will first look at how to do this with the mouse wheel.

When the user tries to scroll the content with the mouse wheel, the first thing to check is if their mouse is within the region of the SCB. If their mouse if outside the SCB, nothing should happen. We can check this with the following function:

```
def mouse_over(x, y, width, height)
    return mouse_x > x && mouse_x < x + width && mouse_y > y && mouse_y <
y + height
end
```

Where `x`, `y`, `width`, `height` are the position and dimensions of the SCB.

Next, we need to set some limits for the position of the content. After all, we don't want the user to scroll the content all the way to negative or positive infinity! Ideally, the top of the content should not be below the top of the SCB and the bottom of the content should not be above the bottom of the SCB. We can express this relation in terms of the `content_top` field as follows:

content_top <= scb.y

content_top + content_height >= scb.y + scb.height

Remember that we can get the bottom of the content and the SCB with their respective `y + height`. Reordering the terms, we get this range for `content_top`:

scb.y + scb.height – content_height <= content_top <= scb.y

You might notice that all this information is available in the `ScrollableContentBox` record.

Now we want to change the value of `content_top` whenever the user scrolls the middle wheel. We can detect this event with `Gosu.button_down(id)`

```
def button_down(id)
    case (id)
    when Gosu::MS_WHEEL_DOWN
        # Scroll down
    when Gosu::MS_WHEEL_UP
        # Scroll up
    end
```

```
end
```

To scroll down, we subtract `content_top` by a certain amount and check that it is still within the range we established above. To scroll up, we do the opposite. You might think it is counterintuitive to subtract when we want to scroll down, since this effectively moves the content up. However, notice that as the content moves up, the SCB moves down relative to the content, so the rendered content portion actually moves down!

```
def scroll(scrollbox, distance)
    scrollbox.content_top += distance; # This could be any amount you see
fit
    if (scrollbox.content_top < scrollbox.y + scrollbox.height -
scrollbox.content_height)
        scrollbox.content_top = scrollbox.y + scrollbox.height -
scrollbox.content_height
    elsif (scrollbox.content_top > scrollbox.y)
        scrollbox.content_top = scrollbox.y
    end
end

def button_down(id)
    case (id)
    when Gosu::MS_WHEEL_DOWN
        if (mouse_over(@scrollbox.x, @scrollbox.y, @scrollbox.width,
@scrollbox.height))
            scroll(@scrollbox, -10)
        end
    when Gosu::MS_WHEEL_UP
        if (mouse_over(@scrollbox.x, @scrollbox.y, @scrollbox.width,
@scrollbox.height))
            scroll(@scrollbox, 10)
        end
    end
end
```

In the `scroll(scrollbar, distance)` function, instead of using if statements to force the value to stay within the range, you can also use the `clamp` function provided by Ruby. The `clamp` function operates on a number and takes as parameters two values representing the `min` and `max`. If the number is less than `min`, it will be set to `min`. If the number is greater than `max`, it will be set to `max`. Otherwise, it stays the same.

```
def scroll(scrollbox, distance)
    content_top_min = scrollbox.y + scrollbox.height -
scrollbox.content_height
    content_top_max = scrollbox.y
    scrollbox.content_top = (scrollbox.content_top +
distance).clamp(content_top_min, content_top_max)
end
```

We also want the user to be able to scroll with the scrollbar attached to the SCB. To do this, we first need to find the position and dimensions of the scrollbar. For the case of a vertical scrollbar, its x-coordinate (of the top-left corner) and width should be fixed. For our scrollbar, we will fix it to the far right of the SCB and set its width to 6 pixels. This leaves us with the y-coordinate and height to determine.

We will find the equation for the scrollbar height first since it will be useful later to calculate the y-coordinate. If we think of the scrollbar height as reflecting what proportion of the content is displayed within the SCB, then intuitively, the ratio between the height of the scrollbar and the height of the SCB should equal the ratio between the height of the SCB and the height of the content. In other words:

```
scrollbar_height = scb_height / content_height * scb_height (1)
```

Notice from the above equation that when `content_height` gets larger, `scrollbar_height` gets smaller, which matches our intuition. Because `scb_height` and `content_height` are already information within the SCB record, we can use these to get the scrollbar height.

The y-coordinate (of the topleft corner) of the scrollbar will vary depending on where the content is in relation to the SCB. Even though the content is moving, it might be easier to imagine it as fixed and the SCB moving in this case. Visualize the top of the SCB sliding down the content. It should run between `content_top` and `content_bottom - SCB_height`. The y-coordinate of the scrollbar will reflect this movement. As `SCB_top` slides from `content_top` to `content_bottom - SCB_height`, the y-coordinate of the scrollbar will slide from `SCB_top` to `SCB_bottom - scrollbar_height`. We can express this relation with:

$$\frac{\texttt{scrollbar\_y - SCB\_top}}{\texttt{SCB\_bottom - scrollbar\_height - SCB\_top}} = \frac{\texttt{SCB\_top - content\_top}}{\texttt{content\_bottom - SCB\_height - content\_top}}$$

Each side of the equation can be thought of as what percentage `scrollbar_y` / `SCB_y` is away from `SCB_top` / `content_top` and towards `SCB_bottom - scrollbar_height` / `content_bottom - SCB_height`, respectively.

Manipulating this expression (by leaving one side with just `scrollbar_y`, replacing `SCB_bottom` with `SCB_top + SCB_height`, `content_bottom` with `content_top + content_height`, and `scrollbar_height` with the expression in (1)), we get:

$$\texttt{scrollbar\_y = SCB\_top +} \frac{\texttt{SCB\_height * (SCB\_top - content\_top)}}{\texttt{content\_height}} \quad (2)$$

Equations (1) and (2) gives us the final pieces we need for the position and dimensions of the scrollbar. We can create a function to calculate these for a given SCB:

```
def scrollbar_dimensions(scb)
    x = scb.x + scb.width - 6 # 6 pixels away from the far right of the
SCB
    y = scb.y + ((scb.y - scb.content_top) * scb.height).to_f /
scb.content_height # We are dealing with floating point values in this
division, so convert the first operand into float so that the result is
also a float
    w = 6 # Our fixed value
    h = scb.height * scb.height / scb.content_height
    return [x, y, w, h] # We return the coordinates and dimensions in a 4-
element array
end
```

With the position and dimensions of the scrollbar found, we can detect when the user clicks on it with the `mouse_over` function defined previously. Once the user left clicks on the scrollbar, they should be able to drag it up and down, during which their mouse may go outside the scrollbar without the drag action being stopped. The drag action will take place in Gosu's update function. When the user lets go of their left mouse, the drag should be stopped. To do this, we need to add two extra fields to the SCB class: `scrollbar_active` and `last_mouse_y`.

`scrollbar_active` is a boolean value indicating whether or not the scrollbar is active and the drag action should take place. It should be set to `true` when the user clicks and holds on the scrollbar and `false` when the user lets go of the left mouse.

`last_mouse_y` keeps track of the last mouse position and is used to determine how much to drag the scrollbar. The drag amount will be `mouse_y - last_mouse_y`. This drag amount is for the scrollbar only but can be converted into the distance `content_top` should move with the following formula:

```
distance_to_move = -1 * (mouse_y – last_mouse_y) / SCB height * content height
```

The term `-1` is here since the scrollbar always move in the opposite direction of `content_top`.

We do not have to `clamp` the drag distance at all. This is already handled by the `scroll` function. If the drag is so great that `content_top` escapes its upper and lower limit, `content_top` will be clamped.

Putting all this information together, we have the following code to achieve scrolling with the scrollbar:

```ruby
def button_down(id)
    case (id)
    when Gosu::MS_LEFT
        dims = scrollbar_dimensions(@scrollbox)
        if (mouse_over(dims[0], dims[1], dims[2], dims[3]))
            @scrollbox.scrollbar_active = true
            @scrollbox.last_mouse_y = mouse_y
        end
    end
end

def button_up(id)
    case (id)
    when Gosu ::MS_LEFT
        @scrollbox.scrollbar_active = false
    end
end

def update()
    if (@scrollbox.scrollbar_active == true)
        distance = (@scrollbox.last_mouse_y - mouse_y) *
@scrollbox.content_height / @scrollbox.height
        scroll(@scrollbox, distance)
        @scrollbox.last_mouse_y = mouse_y
    end
end

def draw()
    dims = scrollbar_dimensions(@scrollbox)
    Gosu.draw_rect(dims[0], dims[1], dims[2], dims[3], Gosu::Color::RED) #
Draw a red scrollbar
end
```

If your program has multiple scrollbars, it is recommended to extract the scrollbar activation, drag and draw logic into their own functions for reusability.

```
# mouse_y parameter is needed if this method lies outside the class that
represents the program (i.e. the one inheriting from Gosu::Window)
def activate_scrollbar(scb, mouse_y)
    scb.scrollbar_active = true
    scb.last_mouse_y = mouse_y
end
def drag_scrollbar(scb, mouse_y)
    distance = (scb.last_mouse_y - mouse_y) * scb.content_height /
scb.height
    scroll(scb, distance)
    scb.last_mouse_y = mouse_y
end
def draw_scrollbar(scb)
    dims = scrollbar_dimensions(scb)
    Gosu.draw_rect(dims[0], dims[1], dims[2], dims[3], Gosu::Color::RED)
end
```

Interacting with content in the SCB

Inside your SCB, you might have interactible elements such as buttons or forms. When these elements are outside the SCB, the user should not be able to click on them. A simple way to achieve this is to modify the `mouse_over` function so that it also takes as parameters the coordinates and dimensions of a SCB:

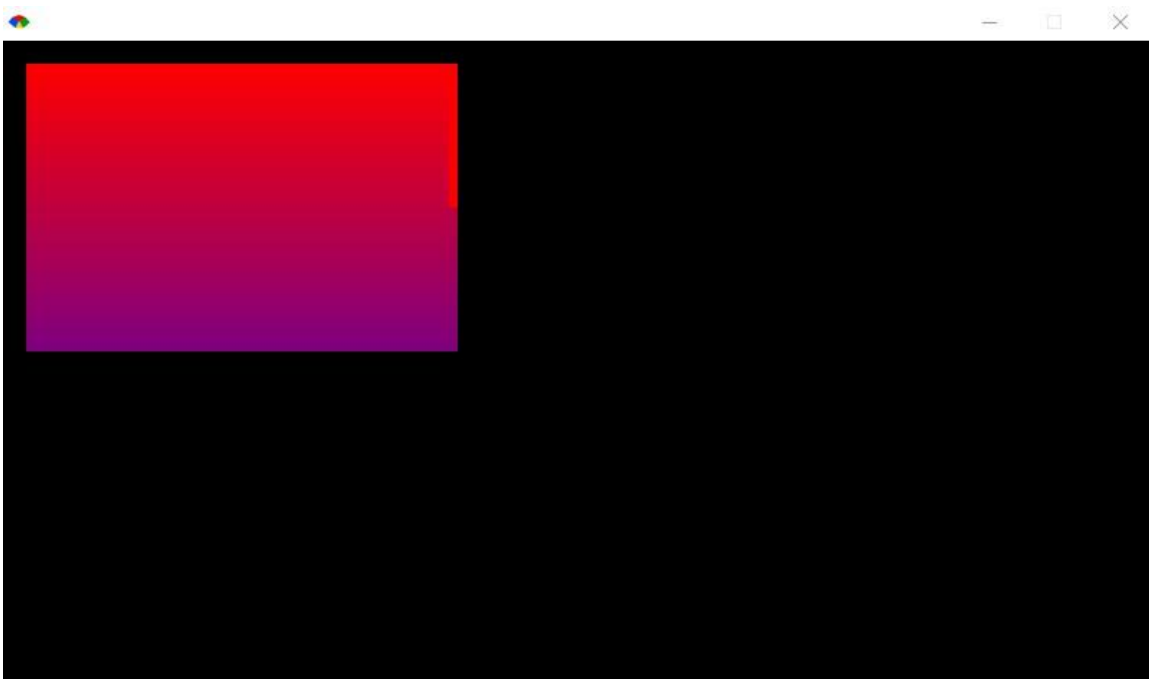
```
def mouse_over(x, y, width, height, scb_x = 0, scb_y = 0, scb_width =
SCREEN_WIDTH, scb_height = SCREEN_HEIGHT)
    if (x < scb_x)
        left = scb_x
    else
        left = x
    end
    if (y < scb_y)
        top = scb_y
    else
        top = y
    end
    if (x + width > scb_x + scb_width)
        right = scb_x + scb_width
    else
        right = x + width
    end
    if (y + height > scb_y + scb_height)
        bot = scb_y + scb_height
    else
        bot = y + height
    end
    return mouse_x > left && mouse_x < right && mouse_y > top && mouse_y <
bot
end
```

Demo program

```
require "gosu"

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 450

class ScrollableContentBox
    attr_accessor :x, :y, :width, :height, :content_top, :content_height,
:scrollbar_active, :last_mouse_y

    def initialize(x, y, width, height, ctop, cheight)
        @x = x
        @y = y
        @width = width
        @height = height
        @content_top = ctop
        @content_height = cheight
    end
end

def scroll(scb, distance)
    content_top_min = scb.y + scb.height - scb.content_height
    content_top_max = scb.y
    scb.content_top = (scb.content_top + distance).clamp(content_top_min, 
content_top_max)
end

def activate_scrollbar(scb, mouse_y)
    scb.scrollbar_active = true
    scb.last_mouse_y = mouse_y
end

def scrollbar_dimensions(scb)
    x = scb.x + scb.width - 6
    y = scb.y + ((scb.y - scb.content_top) * scb.height).to_f / 
scb.content_height
    w = 6
    h = scb.height * scb.height / scb.content_height
    return [x, y, w, h]
end

def draw_scrollbar(scb)
    dims = scrollbar_dimensions(scb)
    Gosu.draw_rect(dims[0], dims[1], dims[2], dims[3], Gosu::Color::RED)
end

def drag_scrollbar(scb, mouse_y)
    distance = (scb.last_mouse_y - mouse_y) * scb.content_height / 
scb.height
    scroll(scb, distance)
    scb.last_mouse_y = mouse_y
end

class Program < Gosu::Window
    def initialize()
        super(SCREEN_WIDTH, SCREEN_HEIGHT)
        @scrollbox = ScrollableContentBox.new(16, 16, 300, 200, 16, 400)
    end

    def mouse_over(x, y, width, height, scb_x = 0, scb_y = 0, scb_width = 
SCREEN_WIDTH, scb_height = SCREEN_HEIGHT)
        if (x < scb_x)
```

```
            left = scb_x
        else
            left = x
        end
        if (y < scb_y)
            top = scb_y
        else
            top = y
        end
        if (x + width > scb_x + scb_width)
            right = scb_x + scb_width
        else
            right = x + width
        end
        if (y + height > scb_y + scb_height)
            bot = scb_y + scb_height
        else
            bot = y + height
        end
        return mouse_x > left && mouse_x < right && mouse_y > top &&
mouse_y < bot
    end

    def update()
        if (@scrollbox.scrollbar_active == true)
            drag_scrollbar(@scrollbox, mouse_y)
        end
    end

    def draw()
        Gosu.clip_to(@scrollbox.x, @scrollbox.y, @scrollbox.width,
@scrollbox.height) do
            Gosu.draw_quad(16, @scrollbox.content_top, Gosu::Color::RED,
316, @scrollbox.content_top, Gosu::Color::RED, 16, @scrollbox.content_top
+ 400, Gosu::Color::BLUE, 316, @scrollbox.content_top + 400,
Gosu::Color::BLUE);
        end
        draw_scrollbar(@scrollbox);
    end

    def button_down(id)
        case (id)
        when Gosu::MS_WHEEL_DOWN
            if (mouse_over(@scrollbox.x, @scrollbox.y, @scrollbox.width,
@scrollbox.height))
                scroll(@scrollbox, -10)
            end
        when Gosu::MS_WHEEL_UP
```

```
            if (mouse_over(@scrollbox.x, @scrollbox.y, @scrollbox.width,
@scrollbox.height))
                scroll(@scrollbox, 10)
            end
        when Gosu::MS_LEFT
            dims = scrollbar_dimensions(@scrollbox)
            if (mouse_over(dims[0], dims[1], dims[2], dims[3]))
                activate_scrollbar(@scrollbox, mouse_y)
            end
        end
    end

    def button_up(id)
        case (id)
        when Gosu::MS_LEFT
            @scrollbox.scrollbar_active = false
        end
    end
end

Program.new().show()
```



References

Module: Gosu. Module: Gosu - Documentation for gosu (1.4.3). (n.d.). Retrieved November 30, 2022, from https://www.rubydoc.info/gems/gosu/Gosu#clip_to-class_method

Ruby: Blocks. GeeksforGeeks. (2022, August 25). Retrieved November 30, 2022, from https://www.geeksforgeeks.org/ruby-blocks/