

Chapter 1

Project: Button Masher

In the last chapter, we covered the basics of C++ programming, as well as the useful tool of logging. In this chapter, we'll finally get some graphics on the screen and create a very simple game about mashing a button as fast as you can. This will get you familiar with the basics of the SFML library.

1.1 Concepts

- **Flowchart:** How to use a flowchart for planning out game logic
- **Game Loop:** How to continually handle display, input, and logic for the game
- **Window:** How to create and manage the game window in SFML
- **Sprites:** How to import a sprite into our project and how to display it in the game
- **Audio:** How to import audio into the project and how to play it in game
- **Text Display:** How to show text on the screen, how to use fonts, and how to edit text from code
- **Mouse Click:** How to detect a user's mouse click, and how to tell if the click was on a sprite

- **Timer:** How to create a simple countdown timer and do something when the time runs out

1.2 Game Brief

A single sprite “button” sits on the screen. The player must click the button to start. The goal is to click on the button as much as possible within a set time limit. Each click adds to the score, which is displayed on the screen. A timer is also shown on the screen. When the timer counts down to 0, the final score is displayed, and the player can click on the button to restart the game. Audio should be included for game actions and background music.

1.3 Planning

Just as in our first project, we need to break down the brief into a list of features so we can implement them more easily.

Challenge



Go through the Game Brief section and do your best to list all of the features you can. When you are done, look at the following Feature List that I came up with and compare your results.

1.3.1 Feature List

- **Object - Button:** A visual representation of a button on the game screen
- **Input - Mouse Click On Object:** Detect when the player has clicked on the button
- **Object - Score Display:** Our current score should be displayed as text on screen

- **Object - Time Display:** The remaining time should be displayed as text on screen
- **Reaction - Start Game:** When the button is first clicked, start the timer.
- **Reaction - Add To Score:** When a button is clicked after the game has started, add to the player's score
- **Reaction - Audio - Button Click:** Play a sound effect when the button is clicked
- **Behaviour - Count Down:** Our timer should count down towards zero
- **Reaction - Game Over Text:** Display game ending text when timer is zero
- **Reaction - Audio - Game Over:** Play game over sound effect when timer is zero
- **Reaction - Restart Game:** When the button is clicked after the game is over, restart the game by setting score to 0 and starting the timer again.
- **Audio - Music:** Play music for the game

There are a few more features here than in our last game, but it is still a manageable list. We've been pretty granular with how we list these features - as you become more experienced, you can lump some features together so your list doesn't get massive for larger games.

1.3.2 Visual Planning

Concepts:

- **Flowchart:** How to use a flowchart for planning out game logic

Button Masher is a very simple game, so we don't need much planning this time around. However, we could still use a simple **flowchart** to visualise how

the game logic will work. Flowcharts are also a useful tool to think about small pieces of logic from more complex games.

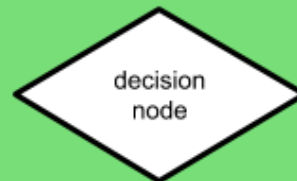
Concept

Flowchart: A programming flowchart is a diagram of the sequence of events in a program. This can be used before writing code to reason out how the code should be constructed.

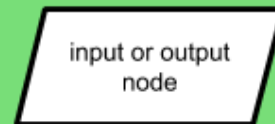
Flowcharts are made up of nodes and connections. There are many different symbols for both nodes and connections that have different meanings, but in practice only a handful are required for most flowcharts.



beginning or ending
node



process node

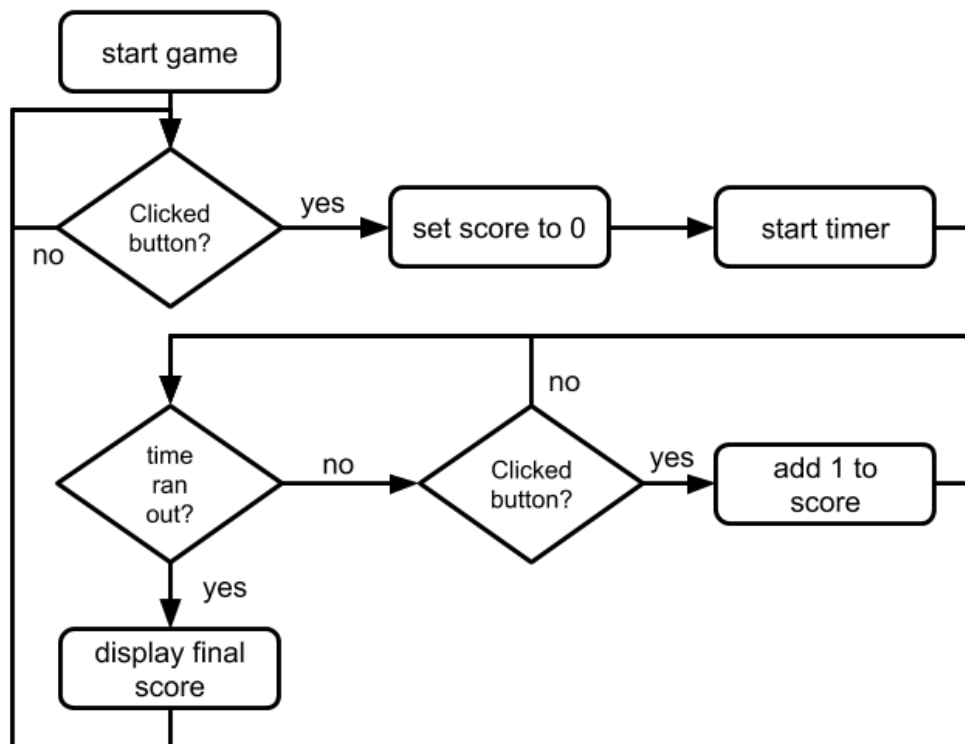


flowline arrow
connector



You have already seen some example flowcharts in the last chapter when we introduced if statements and while loops. Go back and have a look at them and identify the different symbols used.

Now that we know the symbols used for flowcharts, let's create a flowchart for our Button Masher game.



Study this flowchart and assure yourself it follows the game brief. Follow the flow of the chart as if you were a player playing the game.

As we go through the process of coding the game, we will assemble this flowchart piece by piece until we get to the full chart. To illustrate this, I'll include a partial copy of the flowchart each time we add a feature to the game.

Add flow chart
at end of each
section

1.4 Assets

- Sprite - Button (can be anything!)
- Font - UI
- Music - Background
- Sound Effect - Button Press (can be anything!)

- Sound Effect - Time Up

You can use your own assets for this project (and any project in this book), but if you don't want to bother, there are assets available in the downloadable content for this book.

The visual and sound effect assets I provide are from the Kenney asset packs at <https://kenney.nl/assets>. In some cases I have modified these for ease of use in our projects.

The music provided was found using <http://freemusicarchive.org>. This archive website has thousands of music tracks, and you can search based on genre and license to find almost any kind of music you may want for your games.

All assets provided in the downloadable content are licensed under the creative **commons zero license** (also known as **public domain** or **no rights reserved**), so you can do anything you like with them, including sell content created using them. You don't even have to credit the original creator, though of course doing so is always nice! A text file with the sources for each asset is included in the downloadable content for this purpose.

1.5 Implementation

Start off by creating a new project for your Button Masher game. Refer back to the cheat sheet in Chapter 2 if you need a refresher on how to do this.

Go ahead and remove everything from your main function except for the return 0 statement, just as we did at the beginning of Chapter 3.

1.5.1 The Game Loop

Concepts:

- **Game Loop:** How to continually handle display, input, and logic for the game

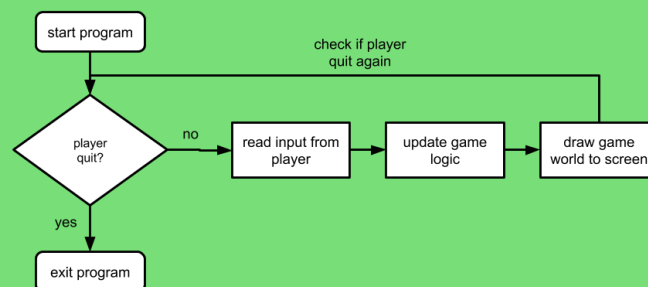
In the last project, we used `system("PAUSE")` to keep our window open. This is a neat trick for consoles, but we need something more advanced for graphical games. For one thing, it wouldn't be good if the logic of the game stopped updating just because the graphics weren't changing! This is where the concept of the **game loop** comes into play.

Concept



Game Loop: The game loop is a special loop that contains the entire game - everything except the initial set up! Inside this loop, the game does three main tasks:

- Checks for input from the player
- Updates the game world based on the input and existing conditions in the game
- Draws the game graphics to the screen



If you think about how often the graphics get updated on screen, you should realise that this loop has to carry out its functionality many times per second.

Let's add the game loop to our code. Inside your `main()` function, add the following lines:

```
while (true)
{
    // Game Loop
}
```

You should recognize the **while loop** that we went over in the last chapter.

This time, our condition looks a little weird - it's just the literal value **true** - meaning, the condition will always be true! This means the loop will never end on its own.

Have a think about what will happen when this code runs. Then, when you think you have an idea, press the play button!

Testing Point



You should see a blank console window sitting open. That's because our while loop is keeping it open, but not printing anything to it. The program will not exit on its own, so it's up to you to close the console window when you're ready.

This is a pretty poor game loop to start with. Let's at least add some comment placeholders for where our future code will go! Make your code match the following:

```
int main()
{
    // Game Loop
    while (true)
    {
        // TODO: Check for input

        // TODO: Update game state

        // TODO: Draw graphics
    }
    // End of Game Loop
    return 0;
}
```

That's a good framework to start with - we'll add to this loop as we go!

1.5.2 SFML Window

Concepts:

- **Window:** How to create and manage the game window in SFML

In our previous project, we used the text console to output data to the player. In this project, we'll instead use SFML to draw content to the screen. We'll start by creating a window using SFML.

Concept



SFML Window: A window in SFML is used to contain all the graphics we will draw with our game. It uses a typical window from the operating system, but can also be fullscreen. SFML also allows the creation of multiple windows - but for our purposes we just need one!

Once we have a window, we can draw graphics on to it. We can also respond to events sent from the window, such as the player resizing or closing it. Finally, we can close the window, so we can implement things like using an escape key to exit the program.

The first thing we need to do is include the SFML graphics library. Add the following line at the very top of your Main.cpp file:

```
#include <SFML/Graphics.hpp>
```

This will allow us to create and manage windows using SFML. It is also needed for creating and manipulating sprites and text.

Let's create our window. Add the following line of code at the top of your main() function, just before your while loop:

```
sf::RenderWindow gameWindow;
```

This is a **variable declaration**, like the ones we used in the last project to store integer number and character text variables. However in this case, we are declaring a variable of a special type defined by SFML: the **sf::RenderWindow** type. We'll learn more about custom defined types later - for now, suffice it to say that the variable not only has data inside, but also some functionality that we can use.

The **sf** in sf::RenderWindow is short for **SFML** - any types or functions from

SFML will need this `sf::` at the front. The `RenderWindow` is the name of the custom type - so this variable will contain our window.

Finally, the **name** of our variable is **gameWindow** - this could be anything, it is just the label we are choosing to refer to this variable. Remember to use descriptive variable names so you can always remember what your variables represent.

Just declaring the variable won't be enough - we need to set up our window by giving it some settings. To do that, we'll call the **create() function**, and pass in some parameters. Remember, **functions** are like shortcuts to a chunk of code, and **parameters** package up some data to send to that code for it to use.

Add the following line right after your window variable declaration:

```
gameWindow.create(sf::VideoMode(800, 600), "Button Masher");
```

The `create` function requires two parameters. The first is the **`sf::VideoMode`** type - this is another custom SFML type. We create the video mode parameter by calling the **`sf::VideoMode()` function** - this is another example of nesting functions, like we saw in the last chapter. The `VideoMode()` function itself takes two parameters - two numbers to define how big the window should be. For now, we'll use 800 and 600.

The second parameter to the `create()` function is a piece of text which will be used as the title for the window. Let's pass in the name of our game, "Button Masher".

Your full program should now look something like this:

```
// Library Includes
#include <SFML/Graphics.hpp> // Library needed for creating and
                             ↪ managing SFML windows

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    ↪ window name
    gameWindow.create(sf::VideoMode(800, 600), "Button Masher");
```

```
// Game Loop
while (true)
{
    // TODO: Check for input

    // TODO: Update game state

    // TODO: Draw graphics
}
// End of Game Loop
return 0;
}
// End of main() Function
```

Testing Point



Run your program. A window with the title “Button Masher” should appear. The contents of the window may be a white or black colour, or some other graphic - since we aren’t drawing anything there ourselves, it will be whatever happens to be in the computer’s memory.

Unfortunately, the window will be unresponsive - you’ll need to close the accompanying console window or press stop in Visual Studio in order to quit the program.

Try changing the numbers you pass into the VideoMode() function and see what happens to the window.

Our window is unresponsive because we aren’t processing any events on it yet. You can’t resize, close, or move the window, and hovering over it results in a busy cursor.

To process events on the window, add the following lines of code inside our game loop, where we left a TODO to add input:

```
sf::Event gameEvent;
while (gameWindow.pollEvent(gameEvent))
```

```
{  
    // This section will be repeated for each event waiting to be  
    ↪ processed  
}
```

The first part of this is a declaration of another variable - this one will hold each event as we process it.

The next section is a while loop. This while loop looks a little different from the ones we have seen before - it has a function in the condition section. This means that when it comes time to check the condition, that function will be run. The function, **pollEvent()**, goes to our window and checks if there are any events waiting to be processed. If there are, it puts the event into the **gameEvent** variable (that's why we put that variable in as a parameter). It will then return the value **true** so our while loop will continue and the event can be processed. If there are no events left, it will return the value **false** so our while loop will finish.

At the moment we'll leave the body of our while loop empty - this will get the events from the window but won't act on them.

Testing Point



Run the program again. This time, the window will be responsive - you can move it and resize it. The resizing may look a little odd, because we aren't doing anything to draw the window differently when it is resized. You also still won't be able to close it.

To close the window, we need to check for the close event and act on it. Add the following lines of code inside your new event polling while loop:

```
if (gameEvent.type == sf::Event::Closed)  
{  
    gameWindow.close();  
}
```

Each time an event is processed, this code will check if that event is a **sf::Event::Closed** event. This event is sent to the window when the player tries to click the close button. If it was a **sf::Event::Closed** event, we call the **close()** function on our window. This will tell the window to close.

Notice this if statement does not have an else section. The else section for if statements is optional - if there is no else section, it just means that nothing happens if the condition is not true. So in our case, if the event is not a `sf::Event::Closed` event, we just don't do anything.

Testing Point



Run the program again. Now, when you close the window, it will actually close. However, you'll notice the console window does not close and the program is technically still running.

The last step to handle the close event is to not only close the visual window but quit the program entirely. Remember that to quit, we simply need to reach the program **exit point** - the **return 0** at the bottom of our **main() function**. At the moment, we are prevented from ever reaching that exit point by our game loop - because the condition is the literal value **true**, it will never end.

Change the first line of the game loop to the following:

```
while (gameWindow.isOpen())
```

This will change our game loop's condition. Now, each time the game loop runs, it will check if the window is still open. If it is, the game loop will continue as normal. If the window has been closed, the game loop will exit and the program will too.

Testing Point



Run your code again. This time when you close the SFML window, the console window and program as a whole should close as well.

To make things easier while we learn development, let's change a couple of our window settings. First, let's make it so we can't resize the window. To do this, we need to add an optional parameter to our `create()` function. Change

your `create()` function to match the following:

```
gameWindow.create(sf::VideoMode(800, 600), "Button Masher", sf::
    ↳ Style::Titlebar | sf::Style::Close);
```

The third parameter here is a style setting for the window. We are combining two styles here - first, we say we want our window to have a titlebar by using `sf::Style::Titlebar`. We can combine style settings by using a **single vertical bar |** (called the **bitwise or operator**). Using this operator, we add a second style - `sf::Style::Close`. This allows us to close the window, but not to resize it.

Finally, let's make our window full screen, but keep the titlebar and border. Change your `create()` function to match the following:

```
gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    ↳ Masher", sf::Style::Titlebar | sf::Style::Close);
```

Instead of passing a specific resolution to our `videoMode`, we now use the function `sf::VideoMode::getDesktopMode()` which will get whatever resolution the player's computer is currently using.

Testing Point



Run the code again. You should see a full screen window with close buttons. You can still move it, but can't resize it. You may notice the window is slightly too big - that is because the video resolution we set doesn't account for the size of the titlebar and borders. We won't worry about fixing this now - for our final game, we'll switch to a full screen mode with no borders or title bar for a more professional look.

Your current program should look something like this:

```
// Library Includes
#include <SFML/Graphics.hpp> // Library needed for creating and
    ↳ managing SFML windows

// The main() Function - entry point for our program
int main()
```

```
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    ↪ window name
    gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    ↪ Masher", sf::Style::Titlebar | sf::Style::Close);

    // Game Loop
    // Repeat as long as the window is open
    while (gameWindow.isOpen())
    {
        // -----
        // Input Section
        // -----
        // Declare a variable to hold an Event, called gameEvent
        sf::Event gameEvent;
        // Loop through all events and poll them, putting each one
        ↪ into our gameEvent variable
        while (gameWindow.pollEvent(gameEvent))
        {
            // This section will be repeated for each event
            ↪ waiting to be processed

            // Did the player try to close the window?
            if (gameEvent.type == sf::Event::Closed)
            {
                // If so, call the close function on the window.
                gameWindow.close();
            }
        }
        // End event polling loop

        // -----
        // Update Section
        // -----
        // TODO: Update game state

        // -----
        // Draw Section
        // -----
        // TODO: Draw graphics

    }
    // End of Game Loop
    return 0;
}
// End of main() Function
```

Notice that I added section comments for our game loop, similar to what we did in Chapter 2. This is to help us find our way around our program as it gets larger. Add these into your own program so you can follow along.

1.5.3 Button Sprite

Concepts:

- **Sprites:** How to import one into our project, how to display it in the game

Features:

- **Object - Button:** A visual representation of a button on the game screen

We have a window, but we aren't drawing anything inside it. To fix that, we need to learn how to import and use sprites in our game.

Concept

Sprites: Sprites are visual assets that are part of a game. In SFML, a sprite contains all the information needed to display a specific copy of an image in a specific location and orientation on the screen.

Each sprite requires a **texture** - the texture contains information about the source image file, but not any information about where and how it will be displayed - that information is in the sprite. Multiple sprites can use the same texture. For example, a game may have many coin pickups on screen at one time, all looking the same. They would each need their own sprite, since they are drawn in different locations. However, they would only need one texture since they would all use the same image.

Textures reference **image files** - these files need to be in your project folder, and will also need to be included with the final game in order to be used. SFML supports the following image file types: bmp, png, tga, jpg, gif, psd, hdr and pic.

To use sprites and textures in our game, we need the SFML graphics library, which we already included in the previous section. But where do we create the sprite? We don't want to do this inside our game loop because then it would happen every frame - we'd end up with hundreds of sprites in no time at all!

Let's instead add a section, just before the game loop starts, for setting up the game. Add the following comment above your game loop:

```
// -----  
// Game Setup  
// -----
```

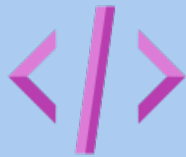
Now we have a section for creating our sprite. The first step is to create the texture - the texture will be used to create the sprite. Underneath your new comment lines, add the following:

```
sf::Texture buttonTexture;  
buttonTexture.loadFromFile("graphics/button.png");
```

The first line declares a variable for our texture, of the type **sf::Texture**. This is a custom type defined by SFML that will hold textures. We name this variable `buttonTexture`, and that is how we'll reference the variable later in code.

The next line calls the function **loadFromFile()** on our texture variable. This will attempt to load a texture from file and store it in our texture variable. It takes one parameter - text for the texture image's file path. If the texture can be created from that image, this function will return the value `true`. If there is some problem, it will return the value `false`. We could then use these values in the code to determine if the load was successful - this is a concept called error handling.

Coding Best Practice



Error handling is a vital skill to learn as a programmer. It's easy to assume your code will work and that the image file will be there and everything will be fine. However, even the best programmers will make mistakes, especially as code changes over time for large projects. It is best to always include sections to handle or at least report errors, so they can be tracked down and fixed.

For our purposes, the default error reporting that SFML does for us is enough. If the texture cannot be created, it will print an error into the console stating this.

Testing Point



Run your game. You won't see anything different in your window yet - even if our texture was successfully created, we haven't drawn it to the screen yet. However, you should see an message in your console window from SFML. This is because we have not yet put the image file into our project, so the game cannot find it!

To get our image to load from file properly, we need to add the file to our project folder. Browse to your project folder in windows (remember that you can right-click on the project in the **Solution Explorer** on the right of the **Visual Studio** window and select **Open Folder in File Explorer** to get there quickly). In your project folder, create a new folder called “**graphics**”. Copy your button image into this folder, and name it “**button**”.

Testing Point



If you run the game now, you still won't see anything on screen - but the error message should be gone! If it is not, make sure your names match the text you are passing to the **loadFromFile()** function exactly. You may need to change the file extension in the code if your button image is a different file type.

We have successfully created a texture, but we have not yet assigned it to a sprite and displayed that sprite to the screen. Add the following code after the **loadFromFile()** function call:

```
sf::Sprite buttonSprite;  
buttonSprite.setTexture(buttonTexture);
```

The first line declares a sprite variable, using the type **sf::Sprite** - another custom type from SFML that holds information about the position, rotation, scale, tint, and more for a sprite. The second line tells our sprite what texture it should be using by calling the **setTexture()** function and passing the texture as a parameter. Multiple sprites can use the same texture - this is because you may want to draw the same image in multiple places on the screen, or in different orientations, etc.








With these two lines, we have created our sprite. However, it still won't show on screen as we have not yet drawn it. Time to move into our game loop. In the draw section of the game loop, add the following code:

```
// Clear the window to a single colour  
gameWindow.clear(sf::Color::Black);  
  
// Draw everything to the window  
gameWindow.draw(buttonSprite);  
// TODO: Draw more stuff
```

```
// Display the window contents on the screen
gameWindow.display();
```

You can see we are using our **gameWindow** variable in each of these lines. To draw things in SFML, we need to use our window - so we are calling functions on that window.

The first function is **clear()**, which can have a colour passed into it. When drawing in SFML, the first thing we have to do is cover the entire screen in a single colour. This will wipe out whatever was on screen in the last frame, so we can replace it with the new draw frame. In this example we clear to the colour black, but you can use any of SFML's pre-defined colours or even define your own:

Code	Appearance
<code>sf::Color::Black</code>	
<code>sf::Color::White</code>	
<code>sf::Color::Red</code>	
<code>sf::Color::Green</code>	
<code>sf::Color::Blue</code>	
<code>sf::Color::Yellow</code>	
<code>sf::Color::Magenta</code>	
<code>sf::Color::Cyan</code>	
<code>sf::Color(r, g, b)</code>	Create your own colour using numbers for each component, between 0 and 255.

The second function is **draw()**, which takes our sprite as a parameter. This will draw the sprite to the window, on top of the solid colour we cleared it to. When we have more stuff to draw, we'll add more **draw()** function calls here and pass each sprite or text to those functions to draw them.

The last function is **display()**. This simply takes our window, which we have now cleared and drawn our sprites onto, and displays that window to the computer screen. Without this function, you wouldn't see the results of your **draw()** function calls.

Testing Point



Run the program. This time, you should see a black screen with your button sprite drawn in the top left corner. If you don't see it, check the console for error messages and make sure you have included all three function calls in your game loop's draw section.

Challenge



Try changing the colour you screen is clearing to, using the table given previously. Choose a pre-set colour, then try making your own using RGB values.

We have visuals! However, you may have noticed your sprite is drawn in the far upper left corner. This isn't ideal - we want it to be in the centre of the screen. To do that, we need to change the sprite's position.

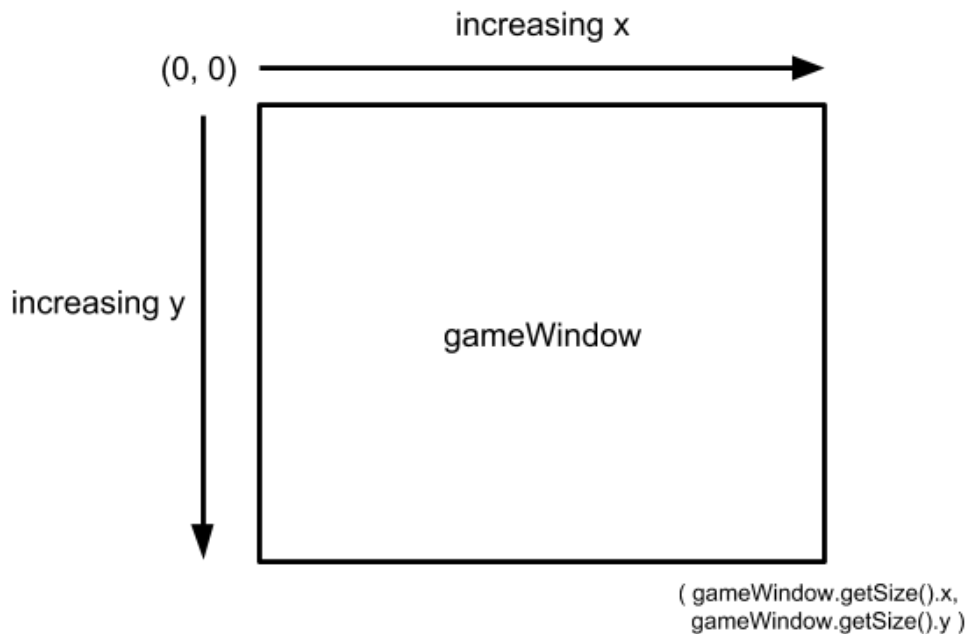
Add the following code up in your game setup section towards the beginning of the program, after we create our sprite and set its texture:

```
buttonSprite.setPosition(  
    gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,  
    gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2  
);
```

This calls the function **setPosition()** on our sprite. This function requires two parameters - the x and y coordinates in pixels for where our sprite will

be displayed on screen.

In SFML, the origin coordinate (0, 0) is located at the top left of the screen. Positive x coordinates move to the right, and positive y coordinates move downward. This may be a bit confusing if you are used to traditional maths coordinates, or to 3D game engines where the y coordinates are commonly positive going upward. However, this downward y convention is pretty common for 2D graphics.



Hint

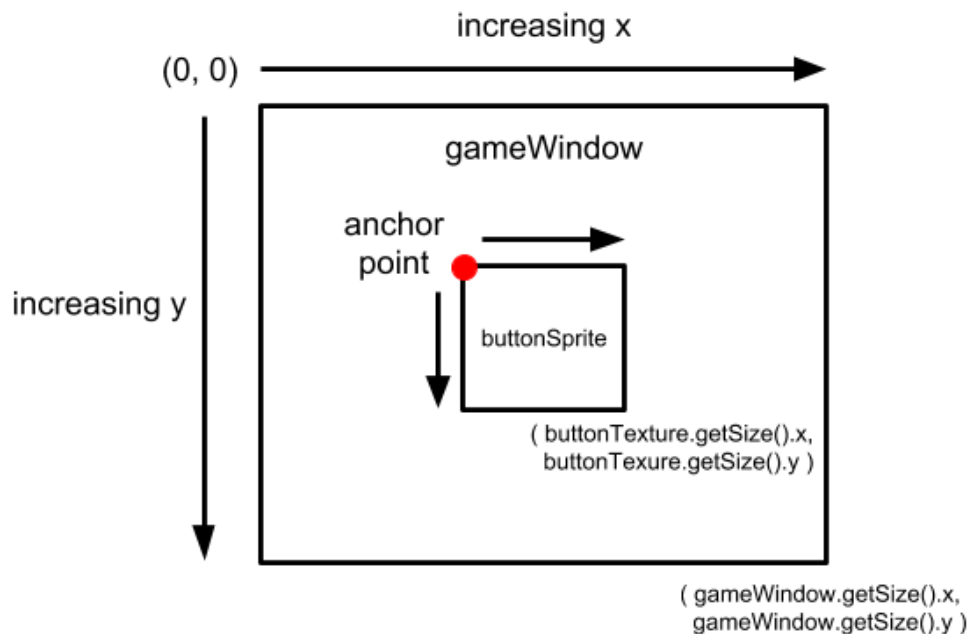


If you need help remembering the downward positive y convention, think of it like you would read a document. You start at the top left, and move both right and down. So right and down are positive, and (0,0) is at the top left!

Because of how the coordinate system is laid out, in order to get our button in the centre of the window, we need to set the position of our sprite to one half of the total width and height of the window. We use the function `getSize()` on our window in order to do this. This function returns an `sf::Vector2f` - this is a custom type which contains a pair of x and y coordinates. So,

after we call `getSize()`, we use `.x` and `.y` to access the x and y portions of the coordinate. We then need to divide this number by 2 in order to get the halfway point.

Finally, we also need to subtract this number by half the size of our button. This is because, by default, sprites in SFML are positioned based on their top left corner. You can think of this corner as their origin or anchor point. If we didn't subtract half our button size, then our button's top left corner would be positioned in the centre of the screen, so the button would be slightly too far to the right and down. To fix this, we call `getSize()` on our texture, get the correct coordinate using `.x` or `.y`, and divide this by 2 to get the amount by which we need to adjust our position.



Testing Point



Run your game. You should see your button drawn in the centre of the window. Go ahead and try changing the coordinates passed in to the `setPosition()` function to see how it changes where the sprite is drawn on the screen!

Your code should now look something like this:

```

// Library Includes
// Library needed for using sprites and textures
#include <SFML/Graphics.hpp>

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    ↪ window name
    gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    ↪ Masher", sf::Style::Titlebar | sf::Style::Close);

    // -----
    // Game Setup
    // -----
    // Button Sprite
    // Declare a texture variable called buttonTexture
    sf::Texture buttonTexture;
    // Load up our texture from a file path
    buttonTexture.loadFromFile("graphics/button.png");
    // Create a sprite variable called buttonSprite
    sf::Sprite buttonSprite;
    // Link our sprite to the texture we created previously, so
    ↪ it knows what to draw
    buttonSprite.setTexture(buttonTexture);
    // Set our button's position to the centre of the screen.
    buttonSprite.setPosition(
        gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,
        gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2
    );

    // -----
    // Game Loop
    // -----
    // Repeat as long as the window is open
    while (gameWindow.isOpen())
    {
        // -----
        // Input Section
        // -----
        // Declare a variable to hold an Event, called gameEvent
        sf::Event gameEvent;
        // Loop through all events and poll them, putting each one
        ↪ into our gameEvent variable
        while (gameWindow.pollEvent(gameEvent))
        {
            // This section will be repeated for each event

```



```
↪ waiting to be processed

    // Did the player try to close the window?
    if (gameEvent.type == sf::Event::Closed)
    {
        // If so, call the close function on the window.
        gameWindow.close();
    }
}
// End event polling loop

// -----
// Update Section
// -----
// TODO: Update game state

// -----
// Draw Section
// -----
// Clear the window to a single colour
gameWindow.clear(sf::Color::Black);

// Draw everything to the window
gameWindow.draw(buttonSprite);
// TODO: Draw more stuff

// Display the window contents on the screen
gameWindow.display();

}
// End of Game Loop
return 0;
}
// End of main() Function
```

1.5.4 Music

Concepts:

- **Audio:** How to import audio into the project, how to play it in game

Features:

- **Audio - Music:** Play music for the game

Now that we have visuals, let's get some audio playing! Audio is a bit simpler than visuals, as we'll see.

Concept



Audio: Audio assets include both music and sound effects. In SFML, audio uses three types: **music**, **sound**, and **sound buffer**. **Music** is a special type for playing longer audio, so that the computer doesn't have to load the entire large song file at once and instead can load it bit by bit from the computer's hard disk. Use this for any audio that lasts several minutes.

For shorter sounds, we use the **sound** and **sound buffer** - these are very similar to the sprite and texture we learned about in the previous section. A **sound** is like a sprite, and contains all the specific information to play a bit of audio, including volume, pitch, and whether the sound should loop.

Each sound requires a **sound buffer**, the same way a sprite requires a texture. The sound buffer contains information about the source audio file, but not any information about how it will be played. Multiple sounds can use the same sound buffer.

Sound buffers and music both reference **sound files** - these files need to be in your project folder, and will also need to be included with the final game in order to be used. SFML supports the following sound file types: WAV, OGG/Vorbis and FLAC. Due to licensing issues MP3 is not supported. There are free online converters that allow you to change MP3 into other file types.

As you may guess by now, the first step to getting this new functionality into our code is to include the library for it. Add this code at the top of your file,

with your other include statements:

```
#include <SFML/Audio.hpp>
```

This will allow us to use the audio library from SFML - which allows us to play music and sound effects.

Next, we need to set up our music and start it playing. Add the following three lines to your project in the setup section, just below where you created your sprite:

```
sf::Music gameMusic;  
gameMusic.openFromFile("audio/music.ogg");  
gameMusic.play();
```

The first line here declares a variable of type **sf::Music**, named `gameMusic`. `sf::Music` is a custom SFML type that contains data and functions for playing music.

The second line calls the **openFromFile()** function on the `gameMusic` variable, and passes through the file name as a parameter. This is very similar to how we loaded our textures in the previous section.

Finally, we start our music playing, by calling the **play()** function on the `gameMusic` variable. This will tell SFML to play whatever file has been set for this `sf::Music` object. Unlike with graphics, audio does not have to be told to play each frame, so we only have to do this once at the beginning of the game for our music!

Testing Point



Try running the game. You won't hear any audio, and you should see an error message in your console. This is because, as with the graphics, we need to place the audio files in the project folder before the game can find them.

To get our music to load and play, we need to add the file to our project folder. Browse to your project folder in windows and create a new folder called **“audio”**. Copy your music into this folder, and name it **“music”**. Make sure the music is one of the supported audio types!

Testing Point

Run your game again. This time, you should hear your music playing! If not, double check that your code matches the file path and type that are actually in the folder.

Your code should now look something like this:

```
// Library Includes
// Library needed for using sprites and textures
#include <SFML/Graphics.hpp>
// Library needed for playing music and sound effects
#include <SFML/Audio.hpp>

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    //   ↳ window name
    gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
        ↳ Masher", sf::Style::Titlebar | sf::Style::Close);

    // -----
    // Game Setup
    // -----
    // Button Sprite
    // Declare a texture variable called buttonTexture
    sf::Texture buttonTexture;
    // Load up our texture from a file path
    buttonTexture.loadFromFile("graphics/button.png");
    // Create a sprite variable called buttonSprite
    sf::Sprite buttonSprite;
    // Link our sprite to the texture we created previously, so
    //   ↳ it knows what to draw
    buttonSprite.setTexture(buttonTexture);
    // Set our button's position to the centre of the screen.
    buttonSprite.setPosition(
        gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,
        gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2
    );

    // Game Music
    // Declare a music variable called gameMusic
```

```
sf::Music gameMusic;
// Load up our audio from a file path
gameMusic.openFromFile("audio/music.ogg");
// Start our music playing
gameMusic.play();

// -----
// Game Loop
// -----
// Repeat as long as the window is open
while (gameWindow.isOpen())
{
    // -----
    // Input Section
    // -----
    // Declare a variable to hold an Event, called gameEvent
    sf::Event gameEvent;
    // Loop through all events and poll them, putting each one
    ↪ into our gameEvent variable
    while (gameWindow.pollEvent(gameEvent))
    {
        // This section will be repeated for each event
        ↪ waiting to be processed

        // Did the player try to close the window?
        if (gameEvent.type == sf::Event::Closed)
        {
            // If so, call the close function on the window.
            gameWindow.close();
        }
    }
    // End event polling loop

    // -----
    // Update Section
    // -----
    // TODO: Update game state

    // -----
    // Draw Section
    // -----
    // Clear the window to a single colour
    gameWindow.clear(sf::Color::Black);

    // Draw everything to the window
    gameWindow.draw(buttonSprite);
    // TODO: Draw more stuff

    // Display the window contents on the screen
```

```
        gameWindow.display();  
  
    }  
    // End of Game Loop  
    return 0;  
}  
// End of main() Function
```

We now have both graphics and audio! The type of audio we have used here is the `sf::Music` type. Later on, when we add sound effects for clicking the button, we'll go over two different types - the `sf::Sound` and `sf::SoundBuffer`. But first, let's get some text in the game!

1.5.5 Score Display

Concepts:

- **Text Display:** How to show text on the screen, how to use fonts, how to edit text from code

Features:

- **Object - Score Display:** Our current score should be displayed as text on screen

Button Masher, like most games, will need to write some text to the screen. To do this, we need to use SFML's font and text objects.

Concept

Text Display: Text display in SFML follows the now-familiar pattern that sprites and textures, or sounds and sound buffers, used in previous sections. **Text** objects are used to draw a specific string of words, using specific colours and other visual settings, to the screen.

Text objects require a reference to a **font**. Fonts contain information about all the letters, numbers, and symbols that particular font uses, so that text with those letters know how to draw them.

Fonts are loaded from **font files**. The supported font file formats are: TrueType, Type 1, CFF, OpenType, SFNT, X11 PCF, Windows FNT, BDF, PFR and Type 42.

Let's start with something simple - displaying the name of the game at the top of the screen. We don't need to include any new libraries for this, since text display functionality is contained in the graphics library that we have already included.

In the game setup section, just below where we set up and play our music, add the following code:

```
sf::Font gameFont;  
gameFont.loadFromFile("fonts/mainFont.ttf");
```

This will declare a variable of the type **sf::Font**, another custom type from SFML which holds all the information for all the letters, numbers, and symbols in a font. We name this variable `gameFont`.

On the next line, we call the **loadFromFile()** function, passing in the file location as a parameter. This works just like the `loadFromFile()` functions for textures and music - it finds that file in the project folder and loads it up to be used by the game.

Next, we need to actually use this font by creating a text object. Add the following code just after the call to `loadFromFile()` for our font:

```
sf::Text titleText;  
titleText.setFont(gameFont);  
titleText.setString("Button Masher!");  
titleText.setPosition(gameWindow.getSize().x / 2 - titleText.  
    ↪ getLocalBounds().width / 2, 30);
```

The first line here declares a variable of type `sf::Text`. This custom SFML type holds the information and functionality to display a specific **string** of text on the screen. Since this text will contain the title of our game, we'll call it `titleText`.

Warning



It's getting a bit confusing to talk about text objects vs the actual sequence of letters that will be displayed by the text object - we can't call them both text. In programming, those letter sequences are called **strings**. I have been avoiding this terminology so far to keep things simpler for beginners, but from now on, we'll be referring to sequences of letters as strings, and text will refer to the objects that display these strings on the screen.

The second line uses the `setFont()` function to tell our text object what font it should use to display the text. We'll pass in our `gameFont` variable as a parameter to this function.

The next line uses the `setString()` function to set the string that this text object will display to the screen. For the title text, we'll just use the literal string value "Button Masher!".

Finally, we need to use the `setPosition()` function to set the text object's position on the screen. We do this in a similar fashion to how we positioned the button previously - we need to set the x position to half the window's width, then subtract again the width of our title text. To do this, we use the x component of `getSize()` from our game window variable, and subtract half of the width component of `getLocalBounds()` from our text variable. We don't need to do this for the y component since we want the text to display near the top of the screen - instead, we just use the value 30 which will anchor the top of our text 30 pixels from the top of the window.

If we were to run the game now, we'd get an error because our font isn't

where the game expects it to be. Copy your font into a new folder called “fonts” inside your project folder, just as we did for the graphics and audio previously. Make sure the file path to your font matches what you have written in the `loadFromFile()` function in your code.

We have now set up our text, but like graphics, we need to draw the text to the screen each frame in our game loop. Add the following line of code after we draw the button sprite in the draw section of our game loop:

```
gameWindow.draw(titleText);
```

As with the button sprite, this simply tells the game window to draw the passed in variable to the screen - in this case, our title text.

Testing Point



Run your code - if all goes well, you should see your text printed at the top of the screen. By default, this text is white - so if you have changed your window to clear to white in your draw section, you may not see the text. Change the clear colour to a different colour temporarily so you can make sure your text is working - we’ll learn how to change the text colour shortly!

You can do a variety of other things to text using SFML. Add the following lines of code **before** your `setPosition()` function call for your text variable in the game setup section:

```
titleText.setCharacterSize(24);  
titleText.setFillColor(sf::Color::Cyan);  
titleText.setStyle(sf::Text::Bold | sf::Text::Italic);
```

Warning

It's important that you set these settings **before** you set the position of the text. That's because the `setPosition()` call includes a check of the text object's bounds, `getLocalBounds()`, and this will calculate the width of the text. This width changes based on these settings, so the settings change has to happen first. If not, your text position will be slightly off.

This demonstrates three useful functions for styling text. The first is the `setCharacterSize()` function, which sets how big the text should be, in pixels.

The second is the `setFillColor()` function, which changes the colour of the text. This uses the same colours shown in the Button Sprite section we went over previously.

The third is the `setStyle()` function. This takes a style type as a parameter. These styles can be combined by using the bitwise or operator (the single bar `|`). The following text styles are supported:

Code	Appearance
<code>sf::Text::Regular</code>	Regular text with no formatting
<code>sf::Text::Bold</code>	Bold text with a thicker line width
<code>sf::Text::Italic</code>	<i>Italic text drawn with a slant</i>
<code>sf::Text::Underlined</code>	<u>Text drawn with a line underneath</u>
<code>sf::Text::StrikeThrough</code>	Text drawn with a line through it

Testing Point

Try running the game. Your title text should now be cyan, larger, bold, and italicized. Try changing the formatting to suit your own preferences!

Challenge

Add a new text object to proclaim you as the author of your game. Remember that you don't need a new font object for this - just another text object. You can duplicate a lot of the code we used for the title text object, but remember to change the string and the position. And don't forget to draw your new text object in the draw section of the game loop! If you get stuck, refer to the example code.

Your code should now look something like this:

```
// Library Includes
// Library needed for using sprites and textures
#include <SFML/Graphics.hpp>
// Library needed for playing music and sound effects
#include <SFML/Audio.hpp>

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    // ↪ window name
    gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    ↪ Masher", sf::Style::Titlebar | sf::Style::Close);

    // -----
    // Game Setup
    // -----
    // Button Sprite
    // Declare a texture variable called buttonTexture
    sf::Texture buttonTexture;
    // Load up our texture from a file path
    buttonTexture.loadFromFile("graphics/button.png");
    // Create a sprite variable called buttonSprite
    sf::Sprite buttonSprite;
    // Link our sprite to the texture we created previously, so
    // ↪ it knows what to draw
    buttonSprite.setTexture(buttonTexture);
    // Set our button's position to the centre of the screen.
    buttonSprite.setPosition(
        gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,
```

```

gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2
);

// Game Music
// Declare a music variable called gameMusic
sf::Music gameMusic;
// Load up our audio from a file path
gameMusic.openFromFile("audio/music.ogg");
// Start our music playing
gameMusic.play();

// Game Font
// Declare a font variable called gameFont
sf::Font gameFont;
// Load up the font from a file path
gameFont.loadFromFile("fonts/mainFont.ttf");

// Title Text
// Declare a text variable called titleText to hold our game
    ↪ title display
sf::Text titleText;
// Set the font our text should use
titleText.setFont(gameFont);
// Set the string of text that will be displayed by this text
    ↪ object
titleText.setString("Button Masher!");
// Set the size of our text, in pixels
titleText.setCharacterSize(24);
// Set the colour of our text
titleText.setFillColor(sf::Color::Cyan);
// Set the text style for the text
titleText.setStyle(sf::Text::Bold | sf::Text::Italic);
// Position our text in the top center of the screen
titleText.setPosition(gameWindow.getSize().x / 2 - titleText.
    ↪ getLocalBounds().width / 2, 30);

// Author Text
sf::Text authorText;
authorText.setFont(gameFont);
authorText.setString("by Sarah Herzog");
authorText.setCharacterSize(16);
authorText.setFillColor(sf::Color::Magenta);
authorText.setStyle(sf::Text::Italic);
authorText.setPosition(gameWindow.getSize().x / 2 -
    ↪ authorText.getLocalBounds().width / 2, 60);

// -----
// Game Loop
// -----

```

```
// Repeat as long as the window is open
while (gameWindow.isOpen())
{
    // -----
    // Input Section
    // -----
    // Declare a variable to hold an Event, called gameEvent
    sf::Event gameEvent;
    // Loop through all events and poll them, putting each one
    ↪ into our gameEvent variable
    while (gameWindow.pollEvent(gameEvent))
    {
        // This section will be repeated for each event
        ↪ waiting to be processed

        // Did the player try to close the window?
        if (gameEvent.type == sf::Event::Closed)
        {
            // If so, call the close function on the window.
            gameWindow.close();
        }
    }
    // End event polling loop

    // -----
    // Update Section
    // -----
    // TODO: Update game state

    // -----
    // Draw Section
    // -----
    // Clear the window to a single colour
    gameWindow.clear(sf::Color::Black);

    // Draw everything to the window
    gameWindow.draw(buttonSprite);
    gameWindow.draw(titleText);
    gameWindow.draw(authorText);
    // TODO: Draw more stuff

    // Display the window contents on the screen
    gameWindow.display();
}
// End of Game Loop
return 0;
}
```

```
// End of main() Function
```

Now we have a handle on static text - time to make some text that can change based on code. Let's create our score text!

First, we need a variable to hold our numerical score. Add the following line at the bottom of our game setup section:

```
int score = 0;
```

This creates an integer (whole number) variable called "score" and sets the initial value in the variable to 0. Later we can add to this when the player clicks on the button.

Next we need to create a new text object for our score text. Add the following after the score declaration:

```
sf::Text scoreText;  
scoreText.setFont(gameFont);  
scoreText.setString("Score: 0");  
scoreText.setCharacterSize(16);  
scoreText.setFillColor(sf::Color::White);  
scoreText.setPosition(30, 30);
```

This creates a new `sf::Text` variable called `scoreText` and sets it up. Notice this time we put it in the top left corner, so we don't need to get the size of the window or the text.

Since we are adding a new text object, we need to make sure to draw it down in the draw section of our game loop:

```
gameWindow.draw(scoreText);
```

Testing Point



You should now see the text "Score: 0" in the top left corner of the window.

Our score is set up and displayed - all we need to do now is change the score text based on our score numerical variable!

To do this, we need a special function called `std::to_string()`. This function is part of the `string` library, so we need to include that library in our program. The `string` library is a set of functions and types for creating and manipulating strings of text. At the top of your program, add a new include statement:

```
#include <string>
```

Next, we need to update our `setString()` function call for our `scoreText` variable. Instead of just setting the string to the literal value “Score: 0”, we’ll change it so it updates score based on whatever the actual value of the score variable is. Change the `setString()` line to the following:

```
scoreText.setString("Score: " + std::to_string(score));
```

Testing Point



You should see the same thing you saw before - the “Score: 0” text in the top left corner of the window. Try changing the value that your score number variable is initialised to, run the program, and see the Score text changes as well. Make sure to change your initial value back to 0 when you are done testing!

This is great to start with, but at the moment we only set our score string once and never change it. This means when our score increases later in the game, it won’t update the display. We need to update the score text in our game loop. As a test, we will also increase the score by one each frame - make sure to take this out once we are done testing this section.

Add the following to the Update section of our game loop (which is currently empty):

```
score = score + 1;  
scoreText.setString("Score: " + std::to_string(score));
```

The first line adds one to our current score, and the second updates the score

text's string to match the score number variable. The second line is exactly the same as the line we use in the setup section.

Testing Point



You should now see an ever increasing score value! Once you see this has worked, take out the line that adds one to the score. The score should now stay at 0 when you run the game.

Your code should now look something like this:

```
// Library Includes
// Library needed for using sprites and textures
#include <SFML/Graphics.hpp>
// Library needed for playing music and sound effects
#include <SFML/Audio.hpp>
// Library for manipulating strings of text
#include <string>

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    // ↪ window name
    gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    ↪ Masher", sf::Style::Titlebar | sf::Style::Close);

    // -----
    // Game Setup
    // -----
    // Button Sprite
    // Declare a texture variable called buttonTexture
    sf::Texture buttonTexture;
    // Load up our texture from a file path
    buttonTexture.loadFromFile("graphics/button.png");
    // Create a sprite variable called buttonSprite
    sf::Sprite buttonSprite;
    // Link our sprite to the texture we created previously, so
    // ↪ it knows what to draw
    buttonSprite.setTexture(buttonTexture);
    // Set our button's position to the centre of the screen.
    buttonSprite.setPosition(
```



```
gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,
gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2
);

// Game Music
// Declare a music variable called gameMusic
sf::Music gameMusic;
// Load up our audio from a file path
gameMusic.openFromFile("audio/music.ogg");
// Start our music playing
gameMusic.play();

// Game Font
// Declare a font variable called gameFont
sf::Font gameFont;
// Load up the font from a file path
gameFont.loadFromFile("fonts/mainFont.ttf");

// Title Text
// Declare a text variable called titleText to hold our game
    ↪ title display
sf::Text titleText;
// Set the font our text should use
titleText.setFont(gameFont);
// Set the string of text that will be displayed by this text
    ↪ object
titleText.setString("Button Masher!");
// Set the size of our text, in pixels
titleText.setCharacterSize(24);
// Set the colour of our text
titleText.setFillColor(sf::Color::Cyan);
// Set the text style for the text
titleText.setStyle(sf::Text::Bold | sf::Text::Italic);
// Position our text in the top center of the screen
titleText.setPosition(gameWindow.getSize().x / 2 - titleText.
    ↪ getLocalBounds().width / 2, 30);

// Author Text
sf::Text authorText;
authorText.setFont(gameFont);
authorText.setString("by Sarah Herzog");
authorText.setCharacterSize(16);
authorText.setFillColor(sf::Color::Magenta);
authorText.setStyle(sf::Text::Italic);
authorText.setPosition(gameWindow.getSize().x / 2 -
    ↪ authorText.getLocalBounds().width / 2, 60);

// Score
// Declare an integer variable to hold the numerical score
```

```

↪ for our game to display
int score = 0;
sf::Text scoreText;
scoreText.setFont(gameFont);
scoreText.setString("Score: " + std::to_string(score));
scoreText.setCharacterSize(16);
scoreText.setFillColor(sf::Color::White);
scoreText.setPosition(30,30);

// -----
// Game Loop
// -----
// Repeat as long as the window is open
while (gameWindow.isOpen())
{
    // -----
    // Input Section
    // -----
    // Declare a variable to hold an Event, called gameEvent
    sf::Event gameEvent;
    // Loop through all events and poll them, putting each one
    ↪ into our gameEvent variable
    while (gameWindow.pollEvent(gameEvent))
    {
        // This section will be repeated for each event
        ↪ waiting to be processed

        // Did the player try to close the window?
        if (gameEvent.type == sf::Event::Closed)
        {
            // If so, call the close function on the window.
            gameWindow.close();
        }
    }
    // End event polling loop

    // -----
    // Update Section
    // -----
    scoreText.setString("Score: " + std::to_string(score));

    // -----
    // Draw Section
    // -----
    // Clear the window to a single colour
    gameWindow.clear(sf::Color::Black);

    // Draw everything to the window
    gameWindow.draw(buttonSprite);

```

```
        gameWindow.draw(titleText);
        gameWindow.draw(authorText);
        gameWindow.draw(scoreText);

        // Display the window contents on the screen
        gameWindow.display();

    }
    // End of Game Loop
    return 0;
}
// End of main() Function
```

Our score is now ready to be increased when we click on the button, which we will add in a later section.

1.5.6 Timer

Concepts:

- **Timer:** How to create a simple countdown timer and do something when the time runs out

Features:

- **Behaviour - Count Down:** Our timer should count down towards zero
- **Object - Time Display:** The remaining time should be displayed as text on screen

We have one more dynamic text object to add to our game in addition to the score: the countdown timer.

Challenge

Create a new text object for the timer and position it in the top right corner. For now, just set the text inside to a static literal value: “Time Remaining: 0”. We’ll update this later! Don’t forget to draw the new text object in the draw loop.

You should now have a new bit of code in your game setup section that looks something like this:

```
sf::Text timerText;  
timerText.setFont(gameFont);  
timerText.setString("Time Remaining: 0");  
timerText.setCharacterSize(16);  
timerText.setFillColor(sf::Color::White);  
timerText.setPosition(gameWindow.getSize().x - timerText.  
    ↪ getLocalBounds().width - 30, 30);
```

The position is set based on the width of the window, minus the width of our text - this is because our text is anchored on the left side so we need to account for the text width in order to get it to align on the right.

You will also need the draw call in the draw section of the game loop:

```
gameWindow.draw(timerText);
```

Testing Point

You should see your new timer text in the top right corner of the screen.

Now that we have our display, we need to update it to indicate how much time the player has left. To do this, we need to use SFML’s time functions.

Concept

Timer: SFML makes it easy to create timers using the **clock** and **time** objects.

A **time** object simply holds a unit of time in an easy to use format. Using a single time object, you can get the time value in seconds, milliseconds, or microseconds.

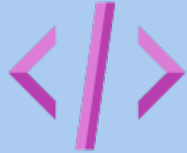
A **clock** object can be started, stopped, and restarted, and keeps track of how much time has passed while it was running. This can be used to get the time elapsed each frame, or to create individual timers for game logic.

First, we need to create a clock object to keep track of time passing in our game. In our game setup section, underneath the code for our timer display, add the following:

```
sf::Time timeLimit = sf::seconds(10.0f);  
sf::Time timeRemaining = timeLimit;  
sf::Clock gameClock;
```

The first line declares a variable of type **sf::Time**, named `timeLimit`. This is a custom SFML type which is just used to store a time value, but can help us convert between seconds, milliseconds, and microseconds. We'll just use seconds for now. We assign an initial value of ten seconds to this variable, using the **sf::seconds()** function, which just creates a new `sf::Time` value based on a number of seconds. This variable will hold the time limit for our game, and won't change through the game.

Coding Best Practice



Notice that the number for the ten seconds value looks a little odd? We add a decimal place (10.0), and an f at the end (10.0f) in order to tell the computer that the value is a **float** - a value that can have decimal points. We do this because the function we are using, the `sf::seconds()` function, expects a float type. You can pass in just 10 on its own, but you may get a compiler warning because 10 is an integer and the function is expecting a float. It's best to always use the same type of literal value as what the computer is expecting, to avoid errors when the computer converts between types.

The next line declares another `sf::Time` variable, called `timeRemaining`. We set this variable's initial value to be equal to our `timeLimit` value. This means that at the beginning of the game, we will have the same amount of time remaining as the full time limit of our game.

The last line declares a variable of type **`sf::Clock`**, named `timerClock`. This is a custom SFML type which contains functionality for tracking time passing. We will be using this inside our game loop to get the amount of time that passes each frame.

With our clock set up, we now need to update the time each frame. At the beginning of the update section of our game loop, add the following code:

```
sf::Time frameTime = gameClock.restart();
timeRemaining = timeRemaining - frameTime;
timerText.setString("Time Remaining: " + std::to_string((int)
    ↪ timeRemaining.asSeconds()));
```

The first line here may look a little odd - each frame, we are restarting the game clock. This **`restart()`** function does two things - first, it resets the clock so it will start tracking time as of that instant. Secondly, it returns the amount of time that passed since the last time it was restarted. Therefore, this line of code gets us the time passed since the last frame, and gets the clock ready to track the time for the following frame. We store the time passed since last frame in a new `sf::Time` variable, called `frameTime`.

On the next line we update our **timeRemaining** variable by subtracting the time passed in the last frame, `frameTime`.

Finally, we update the `timerText` display string to account for our new `timeRemaining` value. This is very similar to our code for updating the score display text, except instead of a simple `int`, we have a `sf::Time` variable. This means we need to do a little work to convert it into an `int` for display. First, we use the **`asSeconds()`** function to get us the time in seconds (as opposed to milliseconds or microseconds). Next, we add the **`(int)`** at the front of the `timerRemaining` variable to indicate we want to convert the value to an integer - otherwise, the value would be displayed as a float and may have many decimal places.

Testing Point



You should now have a timer counting down in the top right corner of the screen. Right now this starts right away, which we will need to change later. It also goes into negative numbers. Later, we'll make the game stop when the timer reaches zero so the numbers will never become negative - for now, we can just ignore this.

Try changing the time limit by editing the `timeLimit` variable's initial value, and see the effect in game!

Your code should now look something like this:

```
// Library Includes
// Library needed for using sprites and textures
#include <SFML/Graphics.hpp>
// Library needed for playing music and sound effects
#include <SFML/Audio.hpp>
// Library for manipulating strings of text
#include <string>

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
```

```

    ↪ window name
gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    ↪ Masher", sf::Style::Titlebar | sf::Style::Close);

// -----
// Game Setup
// -----
// Button Sprite
// Declare a texture variable called buttonTexture
sf::Texture buttonTexture;
// Load up our texture from a file path
buttonTexture.loadFromFile("graphics/button.png");
// Create a sprite variable called buttonSprite
sf::Sprite buttonSprite;
// Link our sprite to the texture we created previously, so
    ↪ it knows what to draw
buttonSprite.setTexture(buttonTexture);
// Set our button's position to the centre of the screen.
buttonSprite.setPosition(
gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,
gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2
);

// Game Music
// Declare a music variable called gameMusic
sf::Music gameMusic;
// Load up our audio from a file path
gameMusic.openFromFile("audio/music.ogg");
// Start our music playing
gameMusic.play();

// Game Font
// Declare a font variable called gameFont
sf::Font gameFont;
// Load up the font from a file path
gameFont.loadFromFile("fonts/mainFont.ttf");

// Title Text
// Declare a text variable called titleText to hold our game
    ↪ title display
sf::Text titleText;
// Set the font our text should use
titleText.setFont(gameFont);
// Set the string of text that will be displayed by this text
    ↪ object
titleText.setString("Button Masher!");
// Set the size of our text, in pixels
titleText.setCharacterSize(24);
// Set the colour of our text

```



```

titleText.setFillColor(sf::Color::Cyan);
// Set the text style for the text
titleText.setStyle(sf::Text::Bold | sf::Text::Italic);
// Position our text in the top center of the screen
titleText.setPosition(gameWindow.getSize().x / 2 - titleText.
    ↪ getLocalBounds().width / 2, 30);

// Author Text
sf::Text authorText;
authorText.setFont(gameFont);
authorText.setString("by Sarah Herzog");
authorText.setCharacterSize(16);
authorText.setFillColor(sf::Color::Magenta);
authorText.setStyle(sf::Text::Italic);
authorText.setPosition(gameWindow.getSize().x / 2 -
    ↪ authorText.getLocalBounds().width / 2, 60);

// Score
// Declare an integer variable to hold the numerical score
    ↪ for our game to display
int score = 0;
sf::Text scoreText;
scoreText.setFont(gameFont);
scoreText.setString("Score: " + std::to_string(score));
scoreText.setCharacterSize(16);
scoreText.setFillColor(sf::Color::White);
scoreText.setPosition(30, 30);

// Timer
sf::Text timerText;
timerText.setFont(gameFont);
timerText.setString("Time Remaining: 0");
timerText.setCharacterSize(16);
timerText.setFillColor(sf::Color::White);
timerText.setPosition(gameWindow.getSize().x - timerText.
    ↪ getLocalBounds().width - 30, 30);

// Create a time value to store the total time limit for our
    ↪ game
sf::Time timeLimit = sf::seconds(10.0f);
// Create a timer to store the time remaining for our game
sf::Time timeRemaining = timeLimit;

// Game Clock
// Create a clock to track time passed each frame in the game
sf::Clock gameClock;

// -----
// Game Loop

```

```

// -----
// Repeat as long as the window is open
while (gameWindow.isOpen())
{
    // -----
    // Input Section
    // -----
    // Declare a variable to hold an Event, called gameEvent
    sf::Event gameEvent;
    // Loop through all events and poll them, putting each one
    ↪ into our gameEvent variable
    while (gameWindow.pollEvent(gameEvent))
    {
        // This section will be repeated for each event
        ↪ waiting to be processed

        // Did the player try to close the window?
        if (gameEvent.type == sf::Event::Closed)
        {
            // If so, call the close function on the window.
            gameWindow.close();
        }
    }
    // End event polling loop

    // -----
    // Update Section
    // -----
    // Get the time passed since the last frame and restart
    ↪ our game clock
    sf::Time frameTime = gameClock.restart();
    // Update our time remaining based on how much time passed
    ↪ last frame
    timeRemaining = timeRemaining - frameTime;
    // Update our time display based on our time remaining
    timerText.setString("Time Remaining: " + std::to_string((
    ↪ int)timeRemaining.asSeconds()));
    // Update our score display text based on our current
    ↪ numerical score
    scoreText.setString("Score: " + std::to_string(score));

    // -----
    // Draw Section
    // -----
    // Clear the window to a single colour
    gameWindow.clear(sf::Color::Black);

    // Draw everything to the window
    gameWindow.draw(buttonSprite);

```

```
        gameWindow.draw(titleText);
        gameWindow.draw(authorText);
        gameWindow.draw(scoreText);
        gameWindow.draw(timerText);

        // Display the window contents on the screen
        gameWindow.display();

    }
    // End of Game Loop
    return 0;
}
// End of main() Function
```

1.5.7 Button Click

Concepts:

- **Mouse Click:** How to detect a user's mouse click on an object

Features:

- **Input - Mouse Click On Object:** Detect when the player has clicked on the button
- **Reaction - Add To Score:** When a button is clicked after the game has started, add to the player's score
- **Reaction - Audio - Button Click:** Play a sound effect when the button is clicked

Now that we have a button and a score, let's connect the two!

Concept

Mouse Click: In SFML, mouse clicks are part of the event system. A window will accept events, and these can be polled to process them - just as we did for the window close event.

There are a variety of mouse and keyboard events, but the one we are interested for this project is the **`sf::Event::MouseButtonPressed`** event. This detects when a mouse button is pushed down. Along with notification that the button press happened, the event includes information about which mouse button was pressed and the x and y coordinates of the mouse. By using these coordinates, we can detect if a mouse click was on our sprite or not.

Add the following code into the event polling loop in the input section of our game loop, above the if statement checking for the closed event:

```
if (gameEvent.type == sf::Event::MouseButtonPressed)
{
    if (buttonSprite.getGlobalBounds().contains(gameEvent.
        ↪ mouseButton.x, gameEvent.mouseButton.y))
    {
        score = score + 1;
    }
}
```

The first if statement here checks if the window has received a **`sf::Event::MouseButtonPressed`** type event. This event is how SFML detects mouse clicks.

If we have received this event, then we go through another if statement to check if the click was on our sprite. To do this, we call the `getGlobalBounds()` function on the sprite to get a rectangle representing our sprite's location on the window. Next, we call the **`contains()`** function on the result, which checks if a specific point is inside the rectangle. We pass the location of the mouse click in to the `contains()` function by using **`gameEvent.mouseButton.x`** and **`gameEvent.mouseButton.y`** to get the x and y coordinates, respectively.

Finally, if both if statements were true (not only was there a mouse click, but it was a mouse click in the right place), we add one to the score.

Testing Point



You should now be able to click on the button sprite in game and see the score increase. Try clicking elsewhere and see that the score only goes up when clicking on the button.

Let's add an audio effect when we click the button. First, we need to create a sound buffer and sound object, in the game setup part of our program. Add the following at the bottom of the setup section:

```
sf::SoundBuffer clickBuffer;  
clickBuffer.loadFromFile("audio/buttonclick.ogg");  
sf::Sound clickSound;  
clickSound.setBuffer(clickBuffer);
```

The first two lines create and load the `sf::SoundBuffer`. Sound buffers are similar to textures - they simply hold the information about a sound from file, without any specific settings.

The second two lines create the `sf::Sound` and associate it with the buffer using the `setBuffer()` function. Sound objects are like sprites, and contain specific settings like volume and pitch. Multiple sounds can reference a single buffer.

With this code, our sounds are created, but not played. Add the following code inside our click inner if statement, where we add one to the score:

```
clickSound.play();
```

Finally, add the `buttonclick.ogg` sound file to your audio folder in your project. Make sure the file path in your code exactly matches the file path in your project folder.

Testing Point

Clicking on the button sprite should now play a clicking sound, as well as increasing the score. Click elsewhere and make sure that the clicking sound only plays for clicks on the button sprite.

Your code so far should look something like this:

```
// Library Includes
// Library needed for using sprites and textures
#include <SFML/Graphics.hpp>
// Library needed for playing music and sound effects
#include <SFML/Audio.hpp>
// Library for manipulating strings of text
#include <string>

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    //   ↪ window name
    gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    //   ↪ Masher", sf::Style::Titlebar | sf::Style::Close);

    // -----
    // Game Setup
    // -----
    // Button Sprite
    // Declare a texture variable called buttonTexture
    sf::Texture buttonTexture;
    // Load up our texture from a file path
    buttonTexture.loadFromFile("graphics/button.png");
    // Create a sprite variable called buttonSprite
    sf::Sprite buttonSprite;
    // Link our sprite to the texture we created previously, so
    //   ↪ it knows what to draw
    buttonSprite.setTexture(buttonTexture);
    // Set our button's position to the centre of the screen.
    buttonSprite.setPosition(
        gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,
        gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2
    );
}
```

```
// Game Music
// Declare a music variable called gameMusic
sf::Music gameMusic;
// Load up our audio from a file path
gameMusic.openFromFile("audio/music.ogg");
// Start our music playing
gameMusic.play();

// Game Font
// Declare a font variable called gameFont
sf::Font gameFont;
// Load up the font from a file path
gameFont.loadFromFile("fonts/mainFont.ttf");

// Title Text
// Declare a text variable called titleText to hold our game
    ↪ title display
sf::Text titleText;
// Set the font our text should use
titleText.setFont(gameFont);
// Set the string of text that will be displayed by this text
    ↪ object
titleText.setString("Button Masher!");
// Set the size of our text, in pixels
titleText.setCharacterSize(24);
// Set the colour of our text
titleText.setFillColor(sf::Color::Cyan);
// Set the text style for the text
titleText.setStyle(sf::Text::Bold | sf::Text::Italic);
// Position our text in the top center of the screen
titleText.setPosition(gameWindow.getSize().x / 2 - titleText.
    ↪ getLocalBounds().width / 2, 30);

// Author Text
sf::Text authorText;
authorText.setFont(gameFont);
authorText.setString("by Sarah Herzog");
authorText.setCharacterSize(16);
authorText.setFillColor(sf::Color::Magenta);
authorText.setStyle(sf::Text::Italic);
authorText.setPosition(gameWindow.getSize().x / 2 -
    ↪ authorText.getLocalBounds().width / 2, 60);

// Score
// Declare an integer variable to hold the numerical score
    ↪ for our game to display
int score = 0;
sf::Text scoreText;
scoreText.setFont(gameFont);
```

```

scoreText.setString("Score: " + std::to_string(score));
scoreText.setCharacterSize(16);
scoreText.setFillColor(sf::Color::White);
scoreText.setPosition(30,30);

// Timer
sf::Text timerText;
timerText.setFont(gameFont);
timerText.setString("Time Remaining: 0");
timerText.setCharacterSize(16);
timerText.setFillColor(sf::Color::White);
timerText.setPosition(gameWindow.getSize().x - timerText.
    ↪ getLocalBounds().width - 30, 30);

// Create a time value to store the total time limit for our
    ↪ game
sf::Time timeLimit = sf::seconds(10.0f);
// Create a timer to store the time remaining for our game
sf::Time timeRemaining = timeLimit;

// Game Clock
// Create a clock to track time passed each frame in the game
sf::Clock gameClock;

// Click Sound Effect
// Create a sound buffer to hold our click sound's file
    ↪ information
sf::SoundBuffer clickBuffer;
// Tell the click sound buffer where the file is
clickBuffer.loadFromFile("audio/buttonclick.ogg");
// Create a click sound to hold the information about how the
    ↪ click sound should be played
sf::Sound clickSound;
// Associate our click sound with the click sound buffer
clickSound.setBuffer(clickBuffer);

// -----
// Game Loop
// -----
// Repeat as long as the window is open
while (gameWindow.isOpen())
{
    // -----
    // Input Section
    // -----
    // Declare a variable to hold an Event, called gameEvent
    sf::Event gameEvent;
    // Loop through all events and poll them, putting each one
    ↪ into our gameEvent variable

```



```

while (gameWindow.pollEvent(gameEvent))
{
    // This section will be repeated for each event
    ↪ waiting to be processed

    // Did the player click the mouse?
    if (gameEvent.type == sf::Event::MouseButtonPressed)
    {
        // Did they click the button sprite?
        if (buttonSprite.getGlobalBounds().contains(
    ↪ gameEvent.mouseButton.x, gameEvent.mouseButton.y))
        {
            // They clicked it!

            // Play a click sound
            clickSound.play();

            // Add to the score
            score = score + 1;
        }
    }

    // Did the player try to close the window?
    if (gameEvent.type == sf::Event::Closed)
    {
        // If so, call the close function on the window.
        gameWindow.close();
    }
}

// End event polling loop

// -----
// Update Section
// -----
// Get the time passed since the last frame and restart
    ↪ our game clock
sf::Time frameTime = gameClock.restart();
// Update our time remaining based on how much time passed
    ↪ last frame
timeRemaining = timeRemaining - frameTime;
// Update our time display based on our time remaining
timerText.setString("Time Remaining: " + std::to_string((
    ↪ int)timeRemaining.asSeconds()));
// Update our score display text based on our current
    ↪ numerical score
scoreText.setString("Score: " + std::to_string(score));

// -----
// Draw Section

```

```
// -----  
// Clear the window to a single colour  
gameWindow.clear(sf::Color::Black);  
  
// Draw everything to the window  
gameWindow.draw(buttonSprite);  
gameWindow.draw(titleText);  
gameWindow.draw(authorText);  
gameWindow.draw(scoreText);  
gameWindow.draw(timerText);  
  
// Display the window contents on the screen  
gameWindow.display();  
  
}  
// End of Game Loop  
return 0;  
}  
// End of main() Function
```

Our game is nearly complete! All we need now is some meta features to wrap things up.

1.5.8 Game Over and Restart

Features:

- **Reaction - Start Game:** When the button is first clicked, start the timer.
- **Reaction - Game Over Text:** Display game ending text when timer is zero
- **Reaction - Audio - Game Over:** Play game over sound effect when timer is zero
- **Reaction - Restart Game:** When the button is clicked again after the game is over, restart the game by setting score to 0 and starting the timer again.

We're nearly done. It's time to make this into a proper game! First, let's add some **game state**.

Definition

Game State: Simply, the game state is the status of everything in the game at a point in time. Game state can also refer to the specific mode, screen, or situation a game is currently in, and this is how we are using this term in this chapter. Our button needs to do two different things depending on if we are currently playing the game or not - this status, playing or not playing, is game state.

Let's add a new variable in our setup section to track whether we are playing the game currently or not:

```
bool playing = false;
```

This creates a **boolean** (bool) variable called playing, and sets this variable to false to start with. A boolean variable can only be either true or false.

Next, let's add a new check in our click handling code. Replace the line that adds to the score with the following code:

```
if (playing == true)
{
    score = score + 1;
}
else
{
    playing = true;
}
```

With this change, we only add to the score if we are actually playing the game. However, if we click the button and we are not yet playing, we start playing.

Let's also prevent the timer from running if the game isn't playing. Replace the line that subtracts the frameTime from timeRemaining with the following:

```
if (playing == true)
{
    timeRemaining = timeRemaining - frameTime;
    if (timeRemaining.asSeconds() <= 0)
```

```
{  
    playing = false;  
}
```

Now we only reduce the time remaining if we are currently playing. We also check if we have run out of time (if the time remaining is less than or equal to zero), and if we have, stop the game from playing.

Testing Point



Now when you start the game, the time will not be ticking down. Click on the button, and the first click will not add to the score - but it will play a sound and it will start the timer! Subsequent clicks should add the score just like they used to. When the timer runs out, you will no longer be able to add to your score, and the timer will stop going down.

Now we can start our game, play it for a set time, and stop. The only problem now is that there is no way to restart. We can change this by adding some re-initialisation when we set our playing state to true.

Inside the if statement for checking clicks, add the following in the else body after setting the playing state to true:

```
score = 0;  
timeRemaining = timeLimit;
```

This means that when the game has stopped, and we press the button again, the score will be reset to zero and the time remaining will be set to the total time limit.

Testing Point



Now when you run out of time, you can restart the game by clicking the button again.

Now we just need some prompts and game over text! First, let's set up a new text object to contain these prompts. Add the following to your setup section:

```
sf::Text promptText;  
promptText.setFont(gameFont);  
promptText.setString("Click the button to start the game!");  
promptText.setCharacterSize(16);  
promptText.setFillColor(sf::Color::White);  
promptText.setPosition(gameWindow.getSize().x / 2 - promptText.  
    ↪ getLocalBounds().width / 2, 200);
```

As usual, we need to add a draw call for this text to our draw section of the game loop:

```
gameWindow.draw(promptText);
```

Finally, we need to change this text to match the current game state. When the game ends, we should change the text to show the player's final score and to let them know they can click the button to start again. In the update section of the game loop, inside the if statement where we stop the play state, add the following code:

```
promptText.setString("Your final score was: "+std::to_string((  
    ↪ int)timeRemaining.asSeconds()) + ". Click the button to  
    ↪ start a new game!");
```

This uses the same functionality as the main score display text object, in order to display the numerical score the player achieved in their last play session.

Let's also change the prompt during gameplay to tell the player to click the button as fast as they can. In the click section, where we set the playing state to true, add the following code just after the reinitialisation of the score and time remaining:

```
promptText.setString("Click the button as fast as you can!");
```

Testing Point

Now when you play the game, you will see a prompt letting you know what to do at each stage. It will also give your final score after you finish a game session.

Challenge

As a final touch, add a new sound effect for the game ending and trigger it to play when the playing state changes to false (when the time runs out). Don't forget to add the sound file to the project!

Your final code should look something like this:

```
// Library Includes
// Library needed for using sprites and textures
#include <SFML/Graphics.hpp>
// Library needed for playing music and sound effects
#include <SFML/Audio.hpp>
// Library for manipulating strings of text
#include <string>

// The main() Function - entry point for our program
int main()
{
    // Declare our SFML window, called gameWindow
    sf::RenderWindow gameWindow;
    // Set up the SFML window, passing the dimensions and the
    //   ↳ window name
    gameWindow.create(sf::VideoMode::getDesktopMode(), "Button
    //   ↳ Masher", sf::Style::Titlebar | sf::Style::Close);

    // -----
    // Game Setup
    // -----
    // Button Sprite
    // Declare a texture variable called buttonTexture
    sf::Texture buttonTexture;
    // Load up our texture from a file path
    buttonTexture.loadFromFile("graphics/button.png");
    // Create a sprite variable called buttonSprite
```

```
sf::Sprite buttonSprite;
// Link our sprite to the texture we created previously, so
    ↪ it knows what to draw
buttonSprite.setTexture(buttonTexture);
// Set our button's position to the centre of the screen.
buttonSprite.setPosition(
gameWindow.getSize().x / 2 - buttonTexture.getSize().x / 2,
gameWindow.getSize().y / 2 - buttonTexture.getSize().y / 2
);

// Game Music
// Declare a music variable called gameMusic
sf::Music gameMusic;
// Load up our audio from a file path
gameMusic.openFromFile("audio/music.ogg");
// Start our music playing
gameMusic.play();

// Game Font
// Declare a font variable called gameFont
sf::Font gameFont;
// Load up the font from a file path
gameFont.loadFromFile("fonts/mainFont.ttf");

// Title Text
// Declare a text variable called titleText to hold our game
    ↪ title display
sf::Text titleText;
// Set the font our text should use
titleText.setFont(gameFont);
// Set the string of text that will be displayed by this text
    ↪ object
titleText.setString("Button Masher!");
// Set the size of our text, in pixels
titleText.setCharacterSize(24);
// Set the colour of our text
titleText.setFillColor(sf::Color::Cyan);
// Set the text style for the text
titleText.setStyle(sf::Text::Bold | sf::Text::Italic);
// Position our text in the top center of the screen
titleText.setPosition(gameWindow.getSize().x / 2 - titleText.
    ↪ getLocalBounds().width / 2, 30);

// Author Text
sf::Text authorText;
authorText.setFont(gameFont);
authorText.setString("by Sarah Herzog");
authorText.setCharacterSize(16);
authorText.setFillColor(sf::Color::Magenta);
```

```
authorText.setStyle(sf::Text::Italic);
authorText.setPosition(gameWindow.getSize().x / 2 -
    ↪ authorText.getLocalBounds().width / 2, 60);

// Score
// Declare an integer variable to hold the numerical score
    ↪ for our game to display
int score = 0;
sf::Text scoreText;
scoreText.setFont(gameFont);
scoreText.setString("Score: " + std::to_string(score));
scoreText.setCharacterSize(16);
scoreText.setFillColor(sf::Color::White);
scoreText.setPosition(30, 30);

// Timer
sf::Text timerText;
timerText.setFont(gameFont);
timerText.setString("Time Remaining: 0");
timerText.setCharacterSize(16);
timerText.setFillColor(sf::Color::White);
timerText.setPosition(gameWindow.getSize().x - timerText.
    ↪ getLocalBounds().width - 30, 30);

// Create a time value to store the total time limit for our
    ↪ game
sf::Time timeLimit = sf::seconds(10.0f);
// Create a timer to store the time remaining for our game
sf::Time timeRemaining = timeLimit;

// Game Clock
// Create a clock to track time passed each frame in the game
sf::Clock gameClock;

// Click Sound Effect
// Create a sound buffer to hold our click sound's file
    ↪ information
sf::SoundBuffer clickBuffer;
// Tell the click sound buffer where the file is
clickBuffer.loadFromFile("audio/buttonclick.ogg");
// Create a click sound to hold the information about how the
    ↪ click sound should be played
sf::Sound clickSound;
// Associate our click sound with the click sound buffer
clickSound.setBuffer(clickBuffer);

// Game state
bool playing = false;
```



```

// Prompt text
sf::Text promptText;
promptText.setFont(gameFont);
promptText.setString("Click the button to start the game!");
promptText.setCharacterSize(16);
promptText.setFillColor(sf::Color::White);
promptText.setPosition(gameWindow.getSize().x / 2 -
    ↪ promptText.getLocalBounds().width / 2, 200);

// Game Over Audio
sf::SoundBuffer gameOverBuffer;
gameOverBuffer.loadFromFile("audio/gameover.ogg");
sf::Sound gameOverSound;
gameOverSound.setBuffer(gameOverBuffer);

// -----
// Game Loop
// -----
// Repeat as long as the window is open
while (gameWindow.isOpen())
{
    // -----
    // Input Section
    // -----
    // Declare a variable to hold an Event, called gameEvent
    sf::Event gameEvent;
    // Loop through all events and poll them, putting each one
    ↪ into our gameEvent variable
    while (gameWindow.pollEvent(gameEvent))
    {
        // This section will be repeated for each event
        ↪ waiting to be processed

        // Did the player click the mouse?
        if (gameEvent.type == sf::Event::MouseButtonPressed)
        {
            // Did they click the button sprite?
            if (buttonSprite.getGlobalBounds().contains(
    ↪ gameEvent.mouseButton.x, gameEvent.mouseButton.y))
            {
                // They clicked it!

                // Play a click sound
                clickSound.play();

                // If the game is currently playing
                if (playing == true)
                {
                    // Add to the score

```

```

        score = score + 1;
    }
    else
    {
        // We aren't playing, so we should start!
        playing = true;
        // Re-initialise the game
        score = 0;
        timeRemaining = timeLimit;
        promptText.setString("Click the button as
↪ fast as you can!");
    }
}

// Did the player try to close the window?
if (gameEvent.type == sf::Event::Closed)
{
    // If so, call the close function on the window.
    gameWindow.close();
}

// End event polling loop

// -----
// Update Section
// -----
// Get the time passed since the last frame and restart
↪ our game clock
sf::Time frameTime = gameClock.restart();
// Are we currently playing?
if (playing == true)
{
    // Update our time remaining based on how much time
↪ passed last frame
    timeRemaining = timeRemaining - frameTime;
    // Are we out of time?
    if (timeRemaining.asSeconds() <= 0)
    {
        // Set playing state to false
        playing = false;
        // Update the prompt
        promptText.setString("Your final score was: "+std::
↪ to_string((int)timeRemaining.asSeconds()) +". Click the
↪ button to start a new game!");
        // Play jingle
        gameOverSound.play();
    }
}

```

```

    // Update our time display based on our time remaining
    timerText.setString("Time Remaining: " + std::to_string((
↪ int)timeRemaining.asSeconds()));
    // Update our score display text based on our current
↪ numerical score
    scoreText.setString("Score: " + std::to_string(score));

    // -----
    // Draw Section
    // -----
    // Clear the window to a single colour
    gameWindow.clear(sf::Color::Black);

    // Draw everything to the window
    gameWindow.draw(buttonSprite);
    gameWindow.draw(titleText);
    gameWindow.draw(authorText);
    gameWindow.draw(scoreText);
    gameWindow.draw(timerText);
    gameWindow.draw(promptText);

    // Display the window contents on the screen
    gameWindow.display();

}
// End of Game Loop
return 0;
}
// End of main() Function

```

Full screen
video mode
with no title
bar or borders

1.6 Improvements

- **Non-Button Click Sound:** Add a different sound effect when clicking outside of the button.
- **Start Game Sound:** Add a sound effect when starting the game, similar to the one when the game ends.
- **Restart Dead Zone:** Make a short dead period of time after the game ends before a new game can be started

1.7 Review

This was our first graphics based project, and we learned a lot of basic SFML concepts that we'll need for all our future SFML projects.

- **Flowchart:** Flow charts are very useful for conceptualizing how game logic works. We used them to visualize the game loop and the overall logic of the Button Masher game.
- **Game Loop:** The game loop is a series of actions that are repeated every frame, including input, update, and draw actions. We used this in Button Masher at a basic level, and we will be building on this concept for future projects.
- **Window:** We created an SFML render window and learned some of the ways the window can be customized. We also learned to handle events such as the window being closed, so our program can quit when this happens.
- **Sprites:** We learned how to load and display sprites to the screen. We only positioned the sprite in this project - in the future, we'll learn to rotate, scale, and tint sprites as well.
- **Audio:** We learned to load and use both music and sound effects in our game. We haven't changed settings on these sounds yet - we will learn to change volumes, pitches, and looping later.
- **Text Display:** We learned to load fonts and create text. We also learned a variety of settings for the displayed text, such as size, position, colour, and style. We also learned to set the text based on numerical values in our game, such as score or time.
- **Mouse Click:** We learned how to respond to mouse events from our SFML window, and how to detect if a mouse click was over a particular sprite.
- **Timer:** We learned how to use SFML timers, and how to create a game timer to track time elapsed between each frame.

The next few games will be very similar to Button Masher - they will all be "clicker" style games, so you'll be able to practice the skills we've learned so

far without adding too much new material. This means we'll have a lot more Challenge sections where you will try things out on your own - so make sure you understand what we've covered so far!

