Drawing 2D Graphics With SFML and Rust

Introduction

This tutorial details how to draw a simple circle shape to the screen. It is assumed that you:

- Have a good understanding of the Rust programming language
- Know how to compile and run a Rust program
- Have an IDE for Rust
- Familiar with what the SFML library is
- Have the SFML library path set up on your device

All the code for this tutorial will be in one file, named main.rs and can be done in whatever IDE you use.

Step 1: Import Necessary Libraries

For every SFML project in Rust, you will need to use `sfml::SfResult` so you can handle errors that occur when using SFML libraries.

For creating 2D graphics, import the following:

- `sfml::graphics::Color` for SFML's shapes and the displayed window.
- `sfml::graphics::RenderWindow` for creating the window where everything will be drawn
- `sfml::graphics::RenderTarget` for rendering objects on the screen
- `sfml::graphics::Transformable` for rotating and resizing shapes
- `sfml::system::Vector2f` for determining a shape's position, size, and other numerical quantities
- `sfml::window::Event` for handling window events such as closing the window
- `sfml::window::Key` for handling keyboard events
- `sfml::window::Style` for configuring the window

For shapes, SFML provides libraries for circles, rectangles, or any other custom shape you wish. You can find the list of shapes and other graphic libraries on the documentation website. As said before, this tutorial will focus on rendering a circle shape to a window. So, to do that, import `sfml::graphics::CircleShape`. If you wanted to import a different shape or even a sprite, you would import sfml::graphics:: and then whatever the name of the library you want to import to draw the desired object.

The code should appear as so when importing the necessary libraries:

```
use sfml::{
    SfResult,
    graphics::{
        CircleShape, Color, RenderTarget, RenderWindow, Transformable,
    },
    system::{Vector2f},
    window::{Event, Key, Style},
};

```

You may have more or different imported graphic libraries depending on the object you want to draw. As long as you have the essentials, `RenderTarget`, `RenderWindow`, and `Color`, you can follow along the tutorial.

Step 2: Create the Window

1) create the main function as shown below.

```
fn main() -> SfResult<()>{

    Ok(())
}
```

Note: Since this is a simple example, everything will be done in the main function. However, it is good practice to separate your code into different functions, where each one has its purpose. For example, there could be a function for specifically creating and configuring the window, another function for configuring your graphic, and so on.

2) Inside the main function, you will create three immutable (aka unchangeable) variables. (Remember, Rust has immutable variables by default):

- `width`: an unsigned integer value that represents the width of the screen to be displayed.
- `height`: an unsigned integer value that represents the height of the screen to be displayed.
- `title`: a string value that will show up as the title bar.

You can set any value to these variables, although you would have to make sure that, for width and height, the values are positive. If you set either the width or height equal to a

negative value, say -600, because these integers are unsigned, you would end up with the positive valued width/height that you may not be expecting.

Here is what the code would look like. Your variable names and values may appear different.

```
10 fn main() -> SfResult<()>{
11     let height = 600;
12     let width = 800;
13     let title = "2D Graphics with SFML and Rust";
14 
15     Ok(())
16 }
```

3) Create another variable, `window`. This variable will be changeable unlike the first three. Set this variable equal to a new `RenderWindow` structure. This structure can have the following arguments: `VideoMode` (which has `width`, `height`, and pixels per bit), `Title`, `Style`, and `ContextSettings`. The only necessary components the window needs are `width`, `height`, `title`, and `Style`. For simplicity, `Style` will be set to its default value. Also, to avoid the hassle of importing and using the struct `VideoMode`, `height` and `width` will be stored in an array (you can also store them in a tuple) and use that array (or tuple) when creating a new window.

```
15     let mut window = RenderWindow::new(
16         [width, height],
17         title,
18         Style::DEFAULT,
19     )?;
```

As shown above, the first arguments are the width and height in an array, the second argument is the title, and the last argument is the style for the window. For more information on the `Style` struct, look at documentation for Style. In the case of default, the window will have a titlebar containing the title, buttons to resize the window, and a button to close the window. The resulting code should appear similar to this:

```rust
use sfml::{
    SfResult,
    graphics::{
        CircleShape, Color, RenderTarget, RenderWindow, Transformable,
    },
    system::{Vector2f},
    window::{Event, Key, Style},
};

fn main() -> SfResult<()>{
    let height = 600;
    let width = 800;
    let title = "2D Graphics with SFML and Rust";
    let mut window = RenderWindow::new(
        [width, height],
        title,
        Style::DEFAULT,
    )?;
}
```

Step 3: Create and Render Object on the Screen

Creating the object itself is simple. Create a new mutable structure for the desired object and send in the necessary arguments. For this tutorial, creating a `CircleShape` requires two arguments: `radius` and `point_count`. Both have default values so if you choose to exclude one or both, the value for `radius` will be set to zero and the `point_count` will be set thirty. In this demonstration, the `radius` will be set to 100, as seen below.

```rust
21      let mut obj = CircleShape::new(100.0);
```

The next lines of code are essential when you are making any Rust project with SFML.

```rust
22      'mainloop: loop{
23          while let Some(event) = window.poll_event(){
24              match event{
25                  Event::Closed | Event::KeyPressed{
26                      code: Key::Escape, ..
27                  } => break 'mainloop,
28                  _ => {}
29              }
30          }
31
32      }
```

The `mainloop` (line 22) is an infinite loop that will break when a condition is met. The `while` loop (line 23) inside of the `mainloop` is a listener for events, such as keypresses, mousepresses, and clicked buttons. Once an event occurs, it pauses until another iteration of the mainloop begins. If either the close button is clicked or the escape key is pressed (line 25), the `mainloop` will end and the window will close (line 27).

After listening for an event, an object can be drawn to the screen as shown below. First (line 32), clear the window of anything has on it and reset the color to anything you want. Next (line 33), draw the image, sending its address as a parameter. Then lastly (line 34), display the window with its new content.

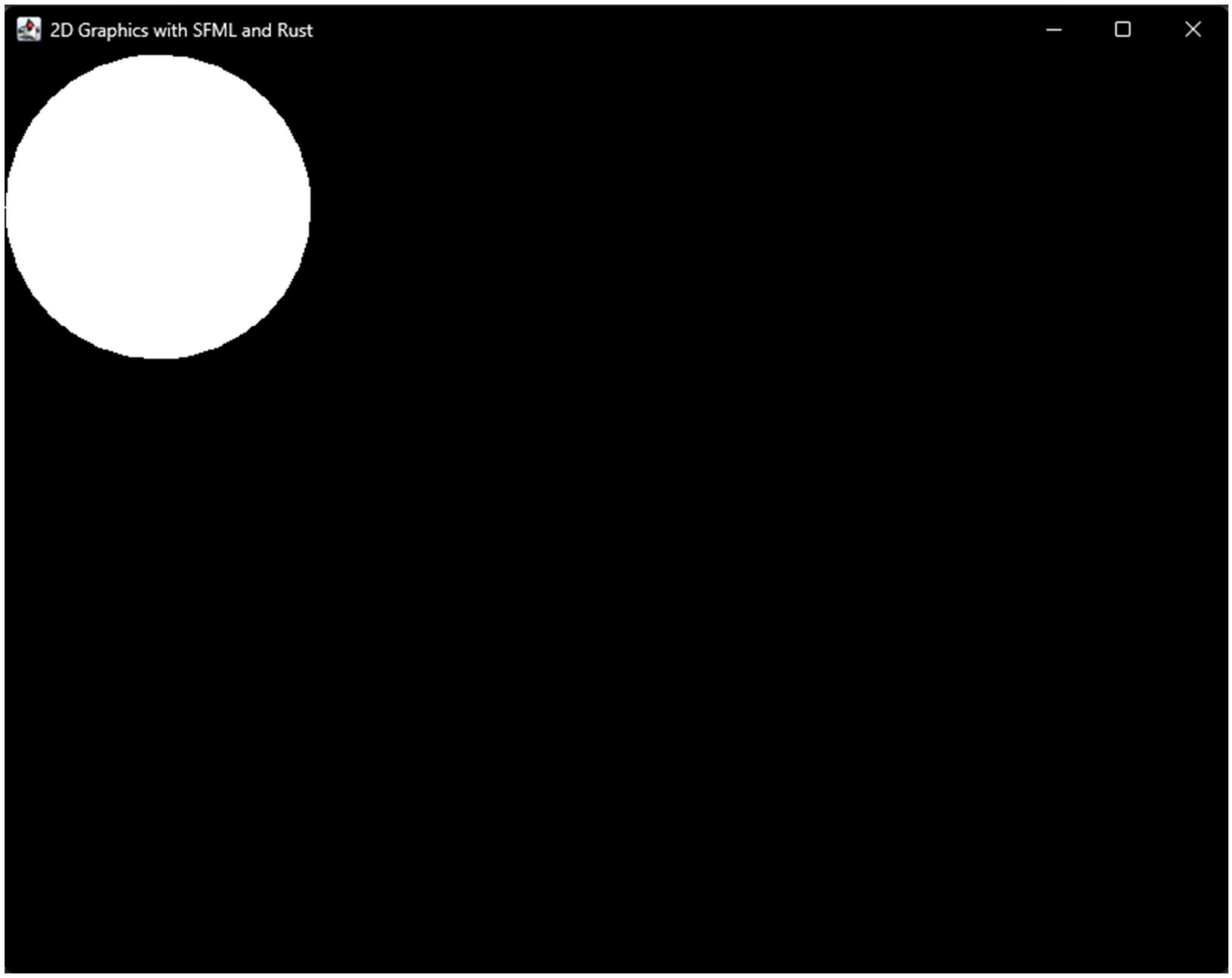
```
32          window.clear(Color::BLACK);
33          window.draw(&obj);
34          window.display();
```

Your code should now appear as:

```
use sfml::{
    SfResult,
    graphics::{
        CircleShape, Color, RenderTarget, RenderWindow, Transformable,
    },
    system::{Vector2f},
    window::{Event, Key, Style},
};

fn main() -> SfResult<()>{
    let height = 600;
    let width = 800;
    let title = "2D Graphics with SFML and Rust";
    let mut window = RenderWindow::new(
        [width, height],
        title,
        Style::DEFAULT,
    )?;

    let mut obj = CircleShape::new(100.0);

    'mainloop: loop{
        while let Some(event) = window.poll_event(){
            match event{
                Event::Closed | Event::KeyPressed{
                    code: Key::Escape, ..
                } => break 'mainloop,
                _ => {}
            }
        }

        window.clear(Color::BLACK);
        window.draw(&obj);
        window.display();
    }
    Ok(())
}
```

Last Step: Run Program

The final result, when the code is runned, should appear similar as the below image:



If you do not see something similar:

- Check for misspellings and missed pieces of code
- Check the SFML Library path
- Check the command you use for running the program