SFML Tutorials and Practice

maksimdan

Published with GitBook

Table of Contents

SFML

Personal notes for practical SFML learning!

Getting Started

Video Notes: https://www.youtube.com/watch?v=FLpD54gx_5w

What is SFML?

Simple and Fast Multi-Media Library.

1. Simple API
2. Light weight
3. Mobile support
4. Open Source
5. Hides complexity
6. Works with OpenGL
7.

Getting Started with Your First Window

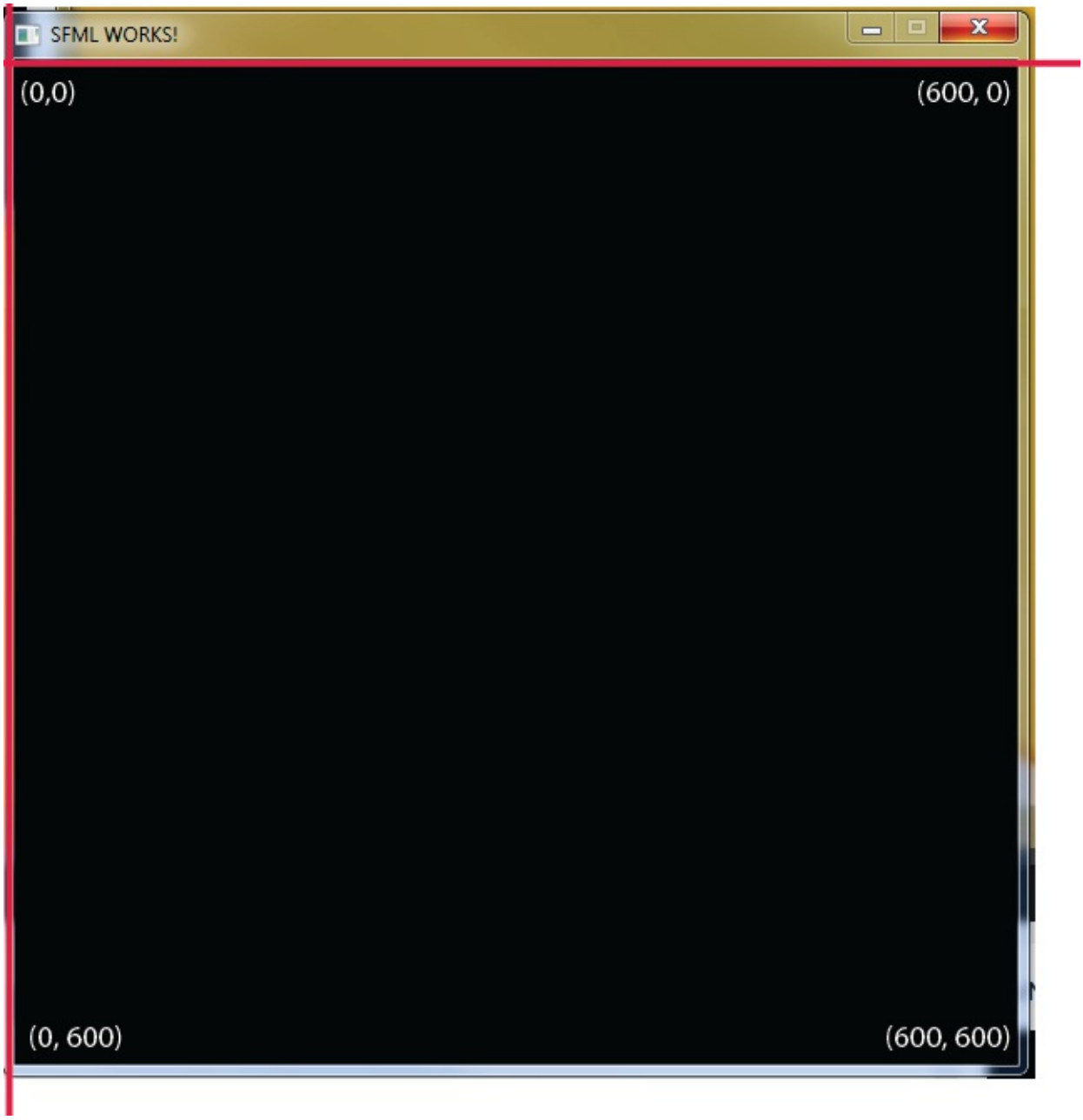
```
#include "SFML/Graphics.hpp"

int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        window.clear();

        window.display();
    }
}
```

This is our first rendered window. Here are some things we've specified:

- A rendered window with a specific size display, and a title.
- An event loop that displays information between clear() and display(). *

We can also notice that the exit handler does not function. This is because it has to be implemented seperately.

Note that the SFML coordinate system works as follows:


SFML WORKS!
(0,0)
(600, 0)
(0, 600)
(600, 600)

Introduction to Events

Incorporates event handling in:

1. Math
2. Keyboard
3. Joystick
4. Text

In handling window events, we create an sd event object

```
sf::Event event;
```

For which we can then use to handle specific events or occurances in the the window. These cases are handled one at a time. For example, we can finally handle both window movement, and exit.

```
// This loop will run everytime an event occurs
        while (window.pollEvent(event))
        {
            // Check for specific events
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();

                break;
            }
        }
```

Key Handling

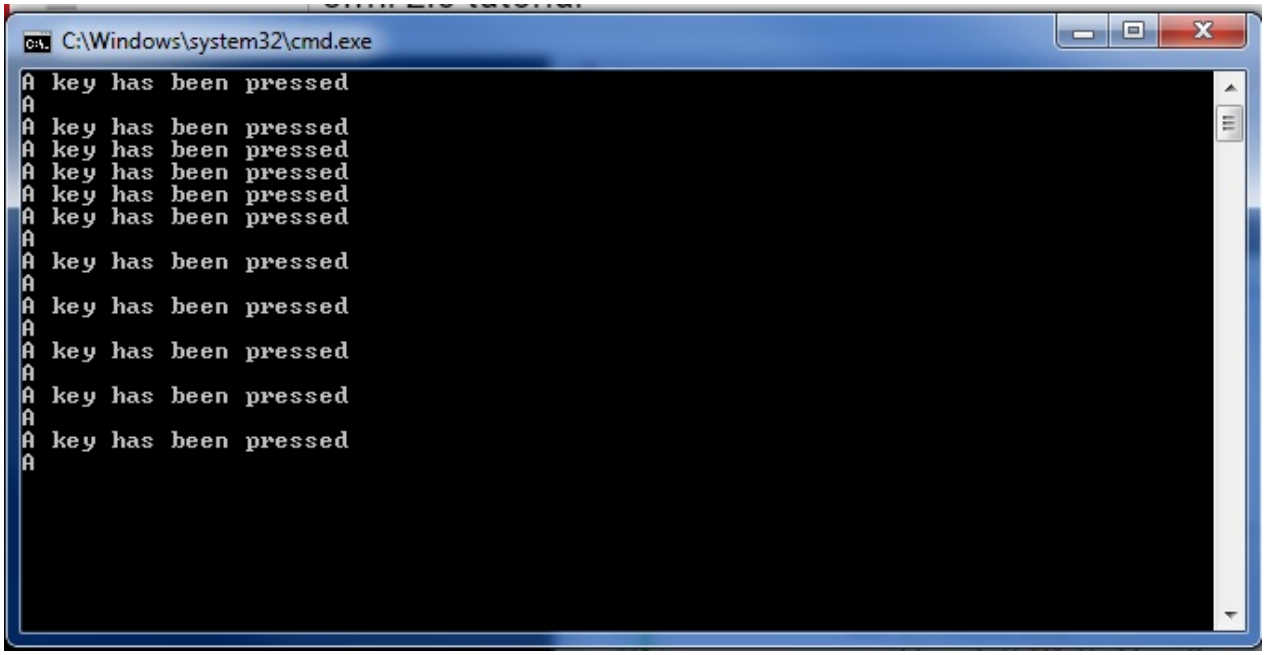
We can also introduce the key handling. In this case, specific to a keyboard. We can do this by denoting an another event case.

```
            case sf::Event::KeyPressed:
                cout << "A key has been pressed" << endl;
```

Although this is quite broad, and can specified further with more particular keyboard input through another a key type:

```
switch (event.key.code)
                {
                case sf::Keyboard::A:
                    cout << "A" << endl;
                    break;
                }
                break;
```

And as expected, our output is as follows.



We can handle key releases as well.

```
            case::sf::Event::KeyReleased:
                cout << "A key has been released" << endl;

                switch (event.key.code)
                {
                case::sf::Keyboard::A:
                    cout << "A released" << endl;
                    break;
                }
                break;
```

So here are some things we can notice so far:

1. There are different event types, and where each event is linked to a particular control on the window.
2. There are generic event types like keypressed and keyreleased, but also subcatagorical situations that are built to handle specific subinstances within those events.

Mouse Handling

A new 'generic' event:

```
            case sf::Event::MouseButtonPressed:
                cout << "Mouse has been pressed" << endl;
                break;
```

and it's apparent subcase:

```
                switch (event.key.code)
                {
                case sf::Mouse::Left:
                    cout << "Left Mouse key has been pressed" << endl;
                    break;
                }
```

Likewise with mouse key releases

```
            case sf::Event::MouseButtonReleased:
                cout << "Mouse button has been released" << endl;
                switch (event.key.code)
                {
                case sf::Mouse::Right:
                    cout << "Right key has been released" << endl;
                    break;
                }
                break;
            }
```

An interesting case is mouse scrolling. In particular, it is possible to capture the speed on the scroll itself, as well as it's direction (-1 = down, +1 = up). Naturally, abs(the larger the number) the greater the speed.

```
            case sf::Event::MouseWheelMoved:
                cout << "Mouse wheel has been scrolled" << endl;

                cout << event.mouseWheel.delta << endl;
                break;
```

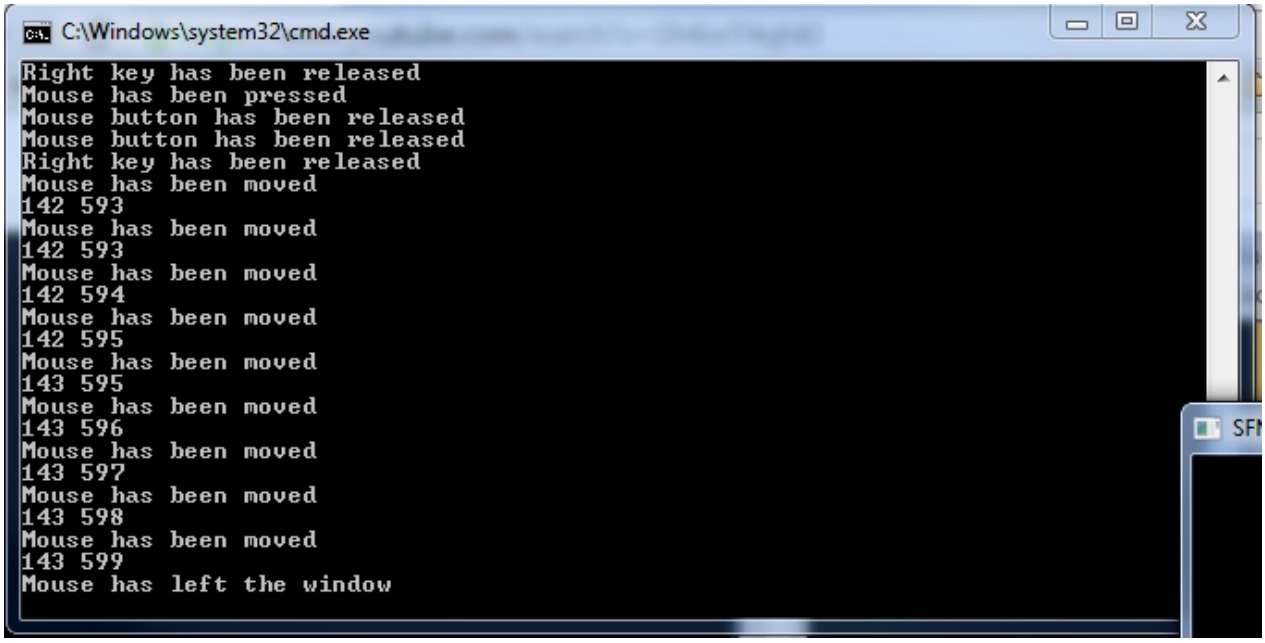
This next one is a fun one. Mouse movement that can capture the specific locality of the mouse position relative to the window. I was surprised to discover that the window is still able to capture the positions of the mouse, regardless on whether or not it is active. Minimized, of course, deactivates it.

```
            case::sf::Event::MouseMoved:
                cout << "Mouse has been moved" << endl;
                cout << event.mouseMove.y << " " << event.mouseMove.x << endl;
                break;
```

And with our final mouse demonstration: Noting when a mouse has both entered and left the field of the window.

```
            case::sf::Event::MouseEntered:
                cout << "Mouse has entered the window" << endl;
                break;

            case::sf::Event::MouseLeft:
                cout << "Mouse has left the window" << endl;
                break;
```

//Output Summary:



For reference, here is the manifested code:

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

using namespace std;

int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        sf::Event event;

        // This loop will run everytime an event occurs
        while (window.pollEvent(event))
        {
            // Check for specific events
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;

            case sf::Event::KeyPressed:
                cout << "A key has been pressed" << endl;

                switch (event.key.code)
                {
                case sf::Keyboard::A:
                    cout << "A" << endl;
                    break;
                }
                break;

            case::sf::Event::KeyReleased:
                cout << "A key has been released" << endl;

                switch (event.key.code)
                {
                case::sf::Keyboard::A:
                    cout << "A released" << endl;
                    break;
                }
                break;

            case sf::Event::MouseButtonPressed:
                cout << "Mouse has been pressed" << endl;
```

```
                switch (event.key.code)
                {
                case sf::Mouse::Left:
                    cout << "Left Mouse key has been pressed" << endl;
                    break;
                }
                break;

            case sf::Event::MouseButtonReleased:
                cout << "Mouse button has been released" << endl;

                switch (event.key.code)
                {
                case sf::Mouse::Right:
                    cout << "Right key has been released" << endl;
                    break;
                }
                break;

            case sf::Event::MouseWheelMoved:
                cout << "Mouse wheel has been scrolled" << endl;

                cout << event.mouseWheel.delta << endl;
                break;

            case::sf::Event::MouseMoved:
                cout << "Mouse has been moved" << endl;
                cout << event.mouseMove.y << " " << event.mouseMove.x << endl;
                break;

            case::sf::Event::MouseEntered:
                cout << "Mouse has entered the window" << endl;
                break;

            case::sf::Event::MouseLeft:
                cout << "Mouse has left the window" << endl;
                break;
            }

        }
        window.clear();

        window.display();
    }
}
```

As well as a cleaned up version:

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

using namespace std;

int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        sf::Event event;

        // This loop will run everytime an event occurs
        while (window.pollEvent(event))
        {
            // Check for specific events
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;

            case sf::Event::KeyPressed:
```

```
                switch (event.key.code)
                {
                case sf::Keyboard::A:
                    cout << "A" << endl;
                    break;
                }
                break;

            case::sf::Event::KeyReleased:

                switch (event.key.code)
                {
                case::sf::Keyboard::A:
                    cout << "A released" << endl;
                    break;
                }
                break;

            case sf::Event::MouseButtonPressed:

                switch (event.key.code)
                {
                case sf::Mouse::Left:
                    cout << "Left Mouse key has been pressed" << endl;
                    break;
                }
                break;

            case sf::Event::MouseButtonReleased:

                switch (event.key.code)
                {
                case sf::Mouse::Right:
                    cout << "Right key has been released" << endl;
                    break;
                }
                break;

            case sf::Event::MouseWheelMoved:
                cout << "Mouse wheel has been scrolled" << endl;

                cout << event.mouseWheel.delta << endl;
                break;

            case::sf::Event::MouseMoved:
                cout << event.mouseMove.y << " " << event.mouseMove.x << endl;
                break;

            case::sf::Event::MouseEntered:
                cout << "Mouse has entered the window" << endl;
                break;

            case::sf::Event::MouseLeft:
                cout << "Mouse has left the window" << endl;
                break;
            }

        }
        window.clear();

        window.display();
    }
}
```

Window Events

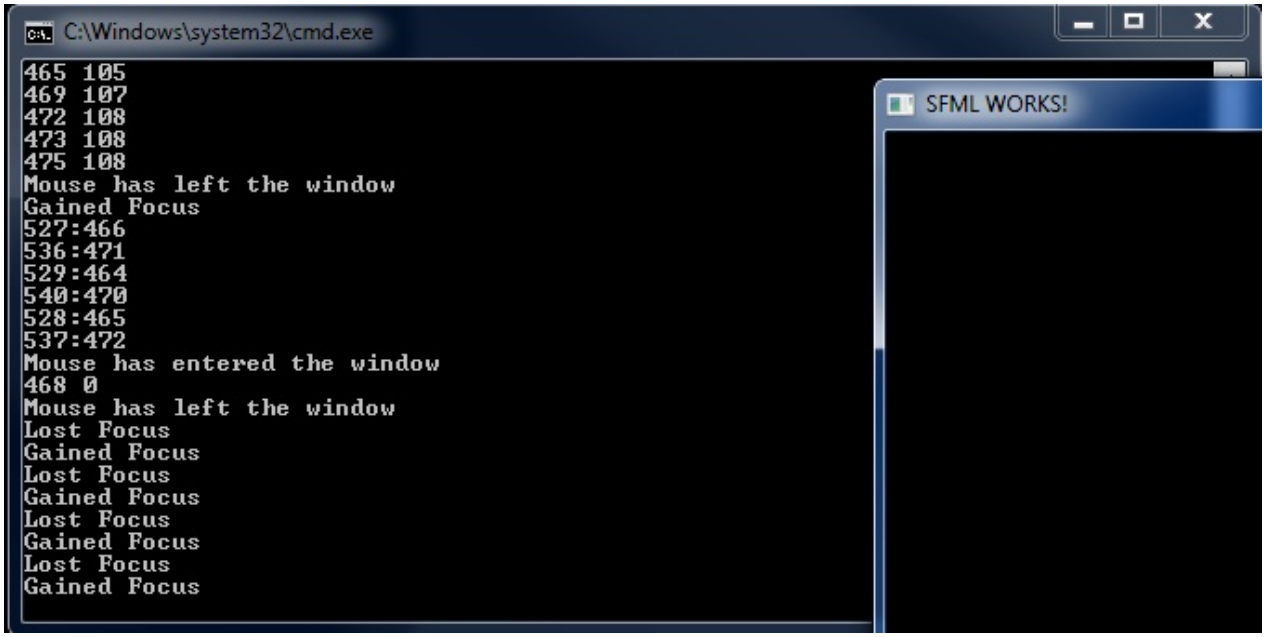
Window resizing. An important thing to notice here is that the when rendering the window in video mode, the window size is initialized. Resizing the window is based, and therefore changes based on these initial parameters.

```
case sf::Event::Resized:
                cout << event.size.width << ":" << event.size.height << endl;
                break;
```

This next case is largely similiar to that of AHK's winActive. SFML calls it focus.

```
            case sf::Event::LostFocus:
                cout << "Lost Focus" << endl;
                break;

            case sf::Event::GainedFocus:
                cout << "Gained Focus" << endl;
                break;
```

//Demonstration Output



Text Events

Just like with the 'keypressed' event, we can take a more specific approach determine any unicode determinate. Take capitals for example and its unicode equivalent.

```
            case sf::Event::TextEntered:
                cout << "Text entered" << endl;

                if (event.text.unicode == 65)
                {
                    cout << "Capital A has been pressed." << endl;
                }
                break;
```

# #

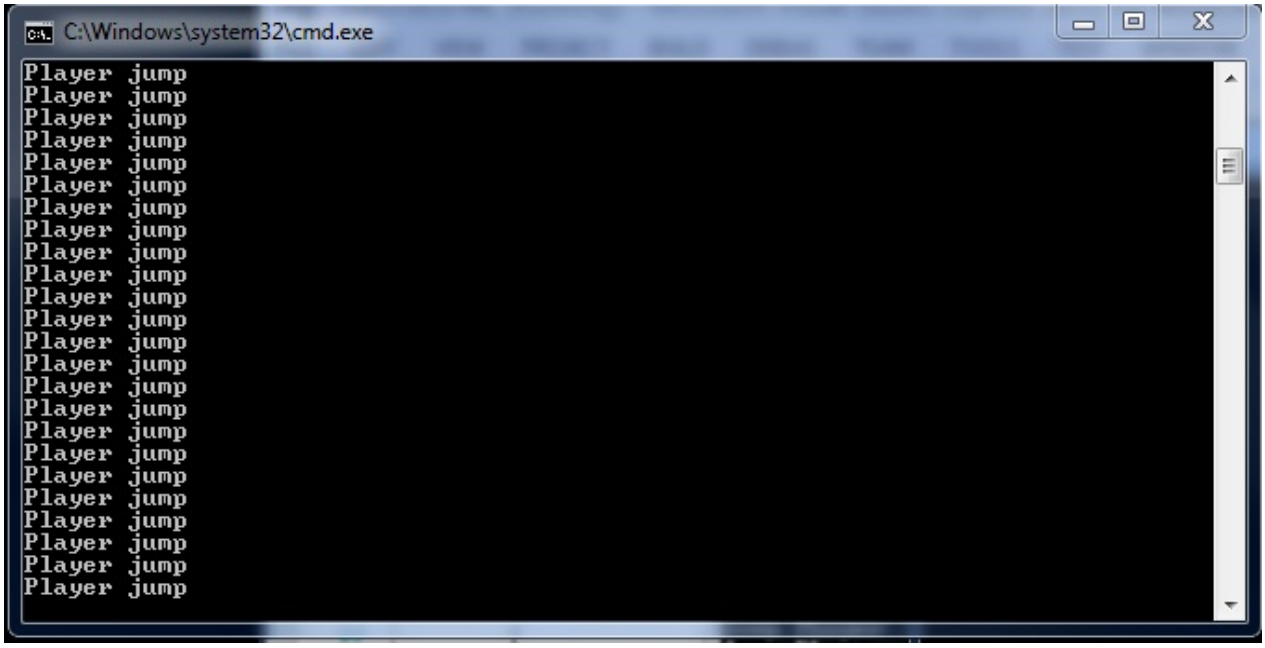
Live KeyBoard Input

Beginning with a fresh slate:

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

using namespace std;

int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORK!");

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();

                break;
            }
        }

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Space))
        {
            cout << "Player jump" << endl;
        }

        window.clear();

        window.display();
    }
}
```

Live events, as seen here, are not explicitly 'events', so they do not belong within the while loop and do not have to be instantiated. What makes live events useful is that, unlike poll events that are much slower, live events are process with great speed so it makes it really great with things like animation.

Poll events are more generic, while live events are more specific to your application.



Live Mouse Input

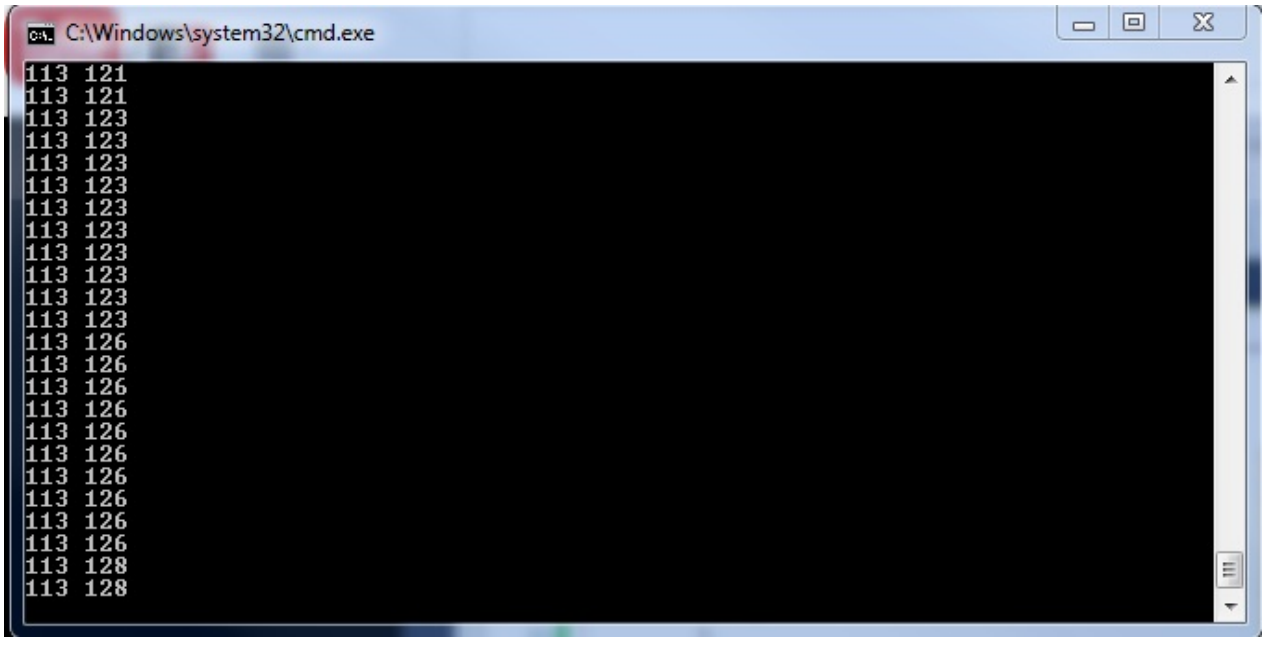
Very similarly, we can detect live mouse input:

```
        if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
        {
            cout << "Fire" << endl;
        }
```

Like before, the output is triggered immediately for the purpose of each individual frame, and smooth gameplay.

Refering backing to an earlier event that allowed us to capture a reletive mouse frame, we can do a very similiar thing with live input.

```
cout << sf::Mouse::getPosition().x << " " << sf::Mouse::getPosition().y << endl;
```

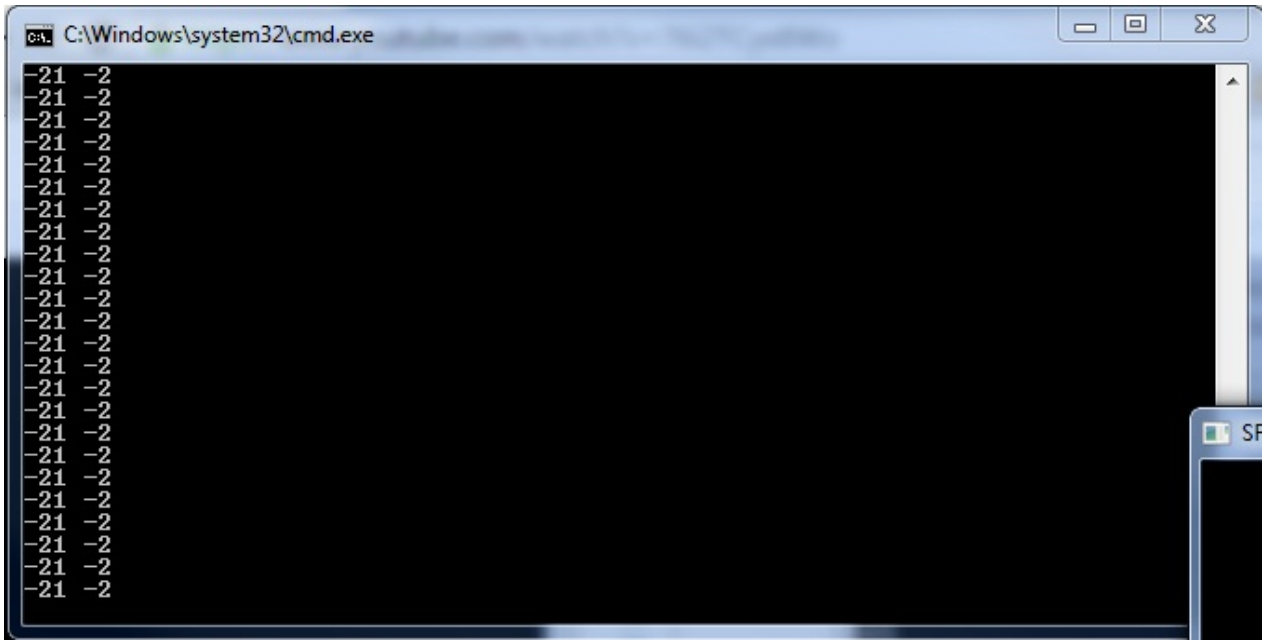


It is important to note that like with any live event, this is updated very quickly, and it is reletive to your monitors as a whole (combining dual displays), rather than the graphics window like before.

However, we can modify this so that its reletivity IS captured by the graphics window:

```
cout << sf::Mouse::getPosition(window).x << " " << sf::Mouse::getPosition(window).y << endl;
```

Still, the positions are still captured outside the window, as shown here:



If the window was the 4th quadrant with its origin on the upper left region, then negative values would be displayed among all other quadrants.

- Left of window = -x
- Above window = -y
- Left and above = -x, -y

Unlike before, shifting the window will change depending on the mouse position.

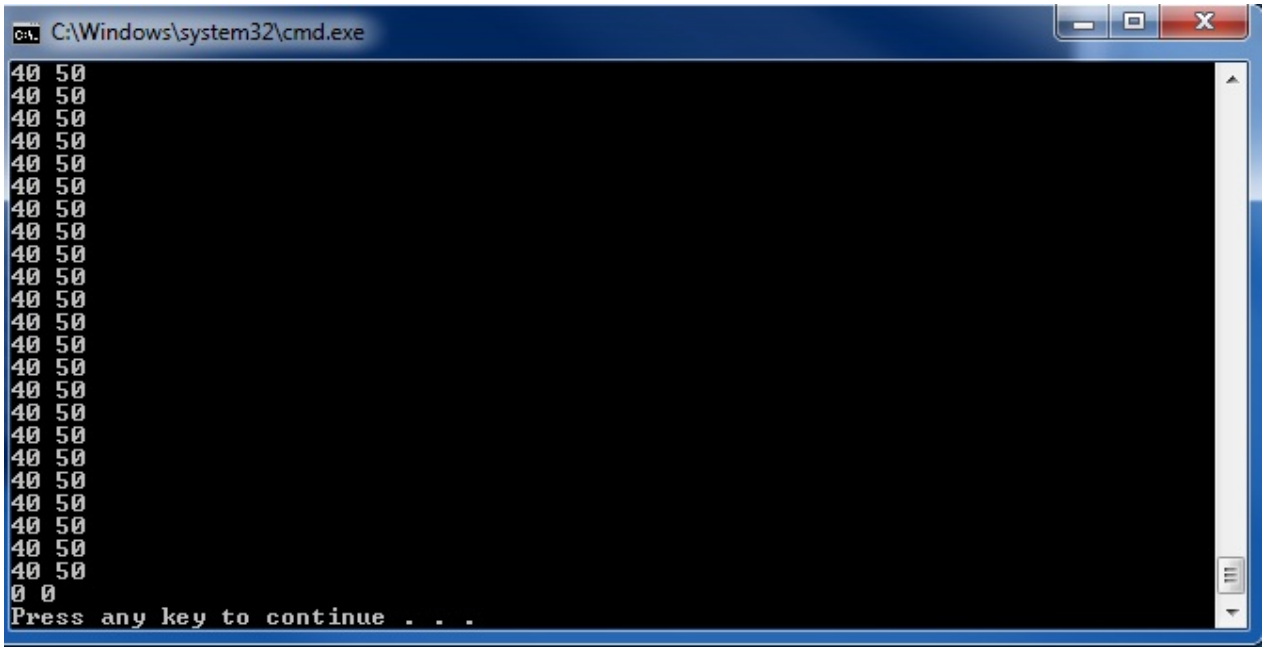
If we want to maintain a position, or a least guide to a particular position on the screen, it can also be done.

```
sf::Mouse::setPosition(sf::Vector2i(40, 50));
```

Now, regardless on where I move my mouse, the position will ALWAYS be set, live, to the position I've specified within the parameters. This, particularily, is reletive to your monitor, but can be specified to change, as such:

```
sf::Mouse::setPosition(sf::Vector2i(40, 50), window);
```

By merely setting this into int main(), this will always be executed quickly, frame by frame. I can attempt to move my mouse, and doing so fast enough, I might notice a flicker between each mouse frame. However, its position will always be quickly reset to the position specified reletive to the window consistantly.


C:\Windows\system32\cmd.exe
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
40 50
0 0
Press any key to continue . . .

Adding a Sprite

A sprite is a computer graphic that can manipulated as a single entity on a screen.

Reseting back to default for demonstration purposes:

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

using namespace std;

int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORK!");

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();

                break;
            }
        }
        window.clear();

        window.display();
    }
}
```

Lets take a moment to explain what window.clear() does. Its functionality is meant to update the screen from frame to frame. For example, if a 2 dimensional square were to be moving from left to right in the window, without the clear() functionality, the previous position, along with the current position of the square would both be displayed. So rather than viewing the intended movement of the square, a single growing rectangle would be shown.

The reason that it is displayed below the pollEvents, is because we want to handle everything the user does before we clear(), and redisplay() [update] the window once again.

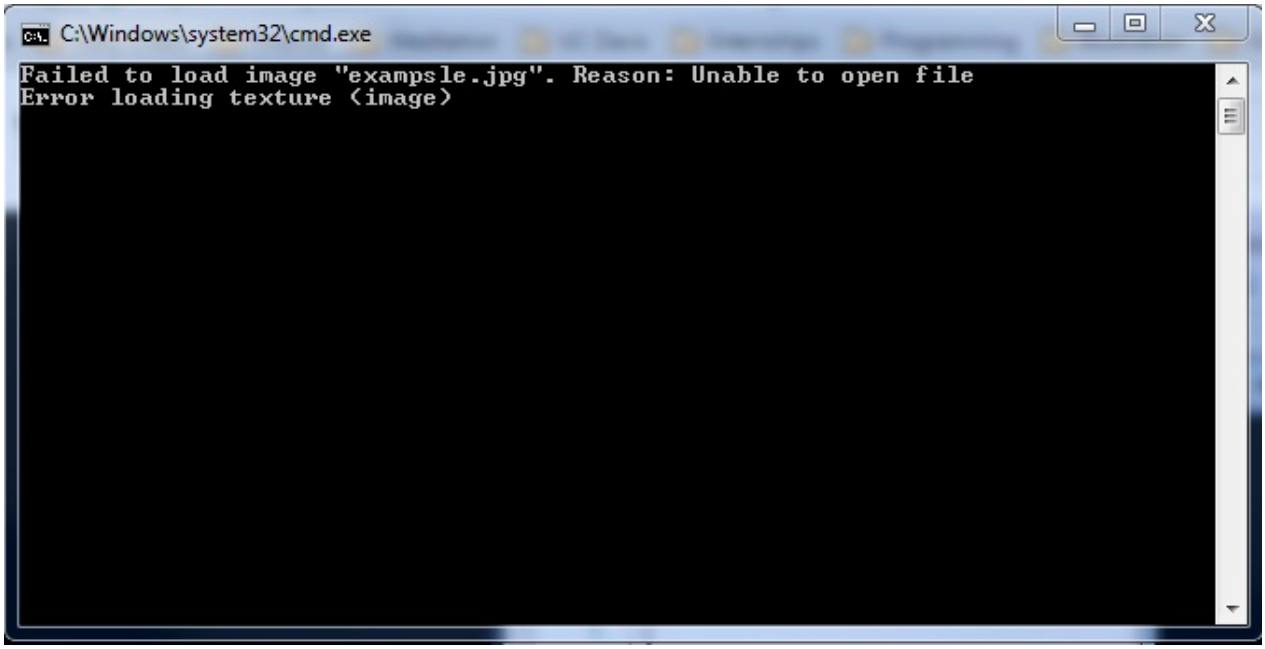
Lets work with our first image:

The first thing we want to do is load the image (texture), and determine whether or not the load was successful.

// after rendering the window

```
sf::Texture texture;
//alternative instantiation:
//sprite.setTexture(texture);

if (!texture.loadFromFile("example.jpg"))
{
    cout << "Error loading texture (image)" << endl;
}
```

Failure in loading the image, may look something like this:

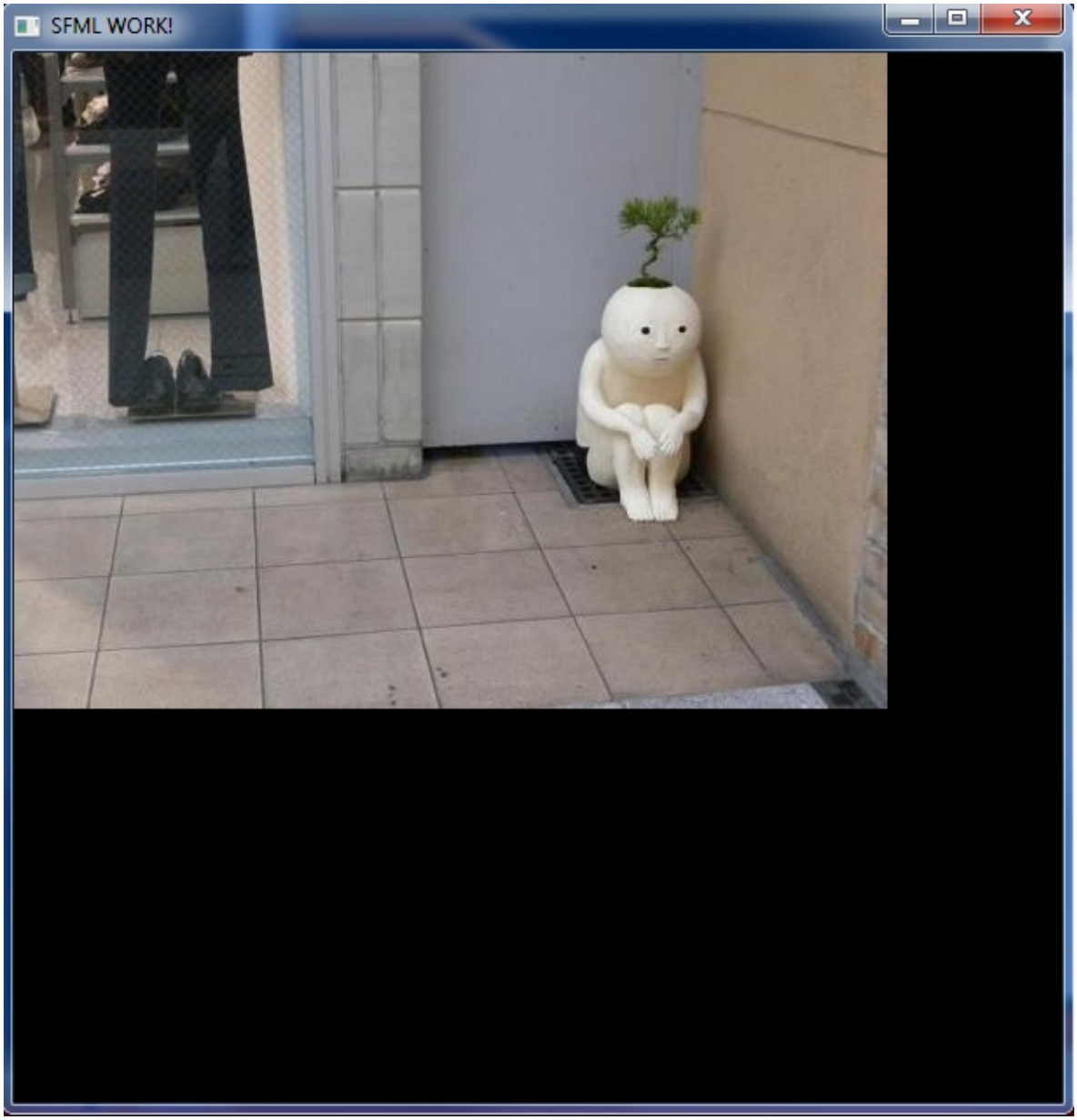


Next, we after a successful load, we instantiate the image as a sprite:

```
sf::Sprite sprite(texture);
```

And now "draw" the image onto our display.

```
        window.clear();

        window.draw(sprite);

        window.display();
```



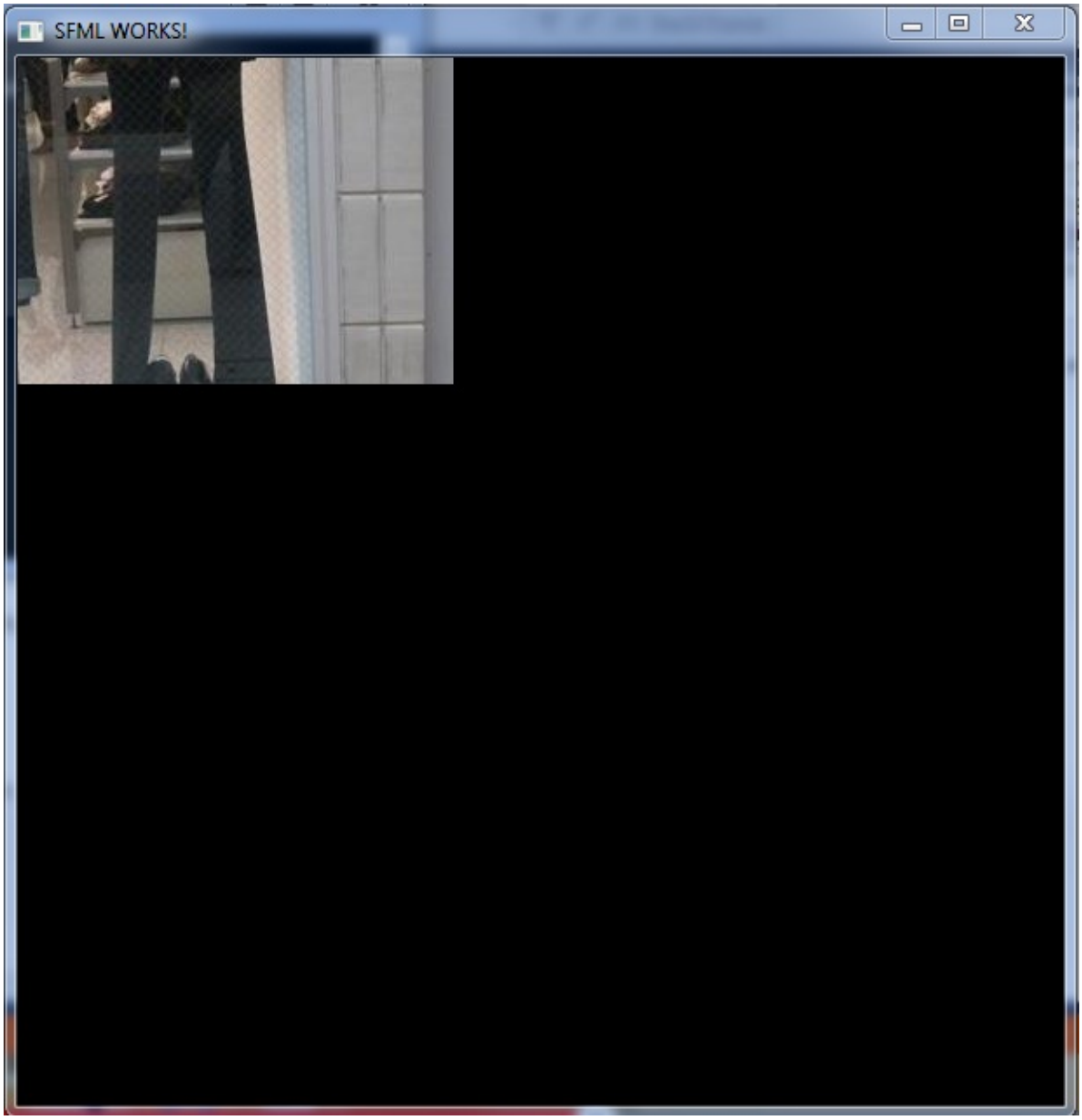
Setting Spite Texture Rect Size

By rect size, we are refering to its 'rectangular size', and what kinds of features we can manipulate here.

```
sprite.setTextureRect(sf::IntRect(0, 0, 250, 187));
```

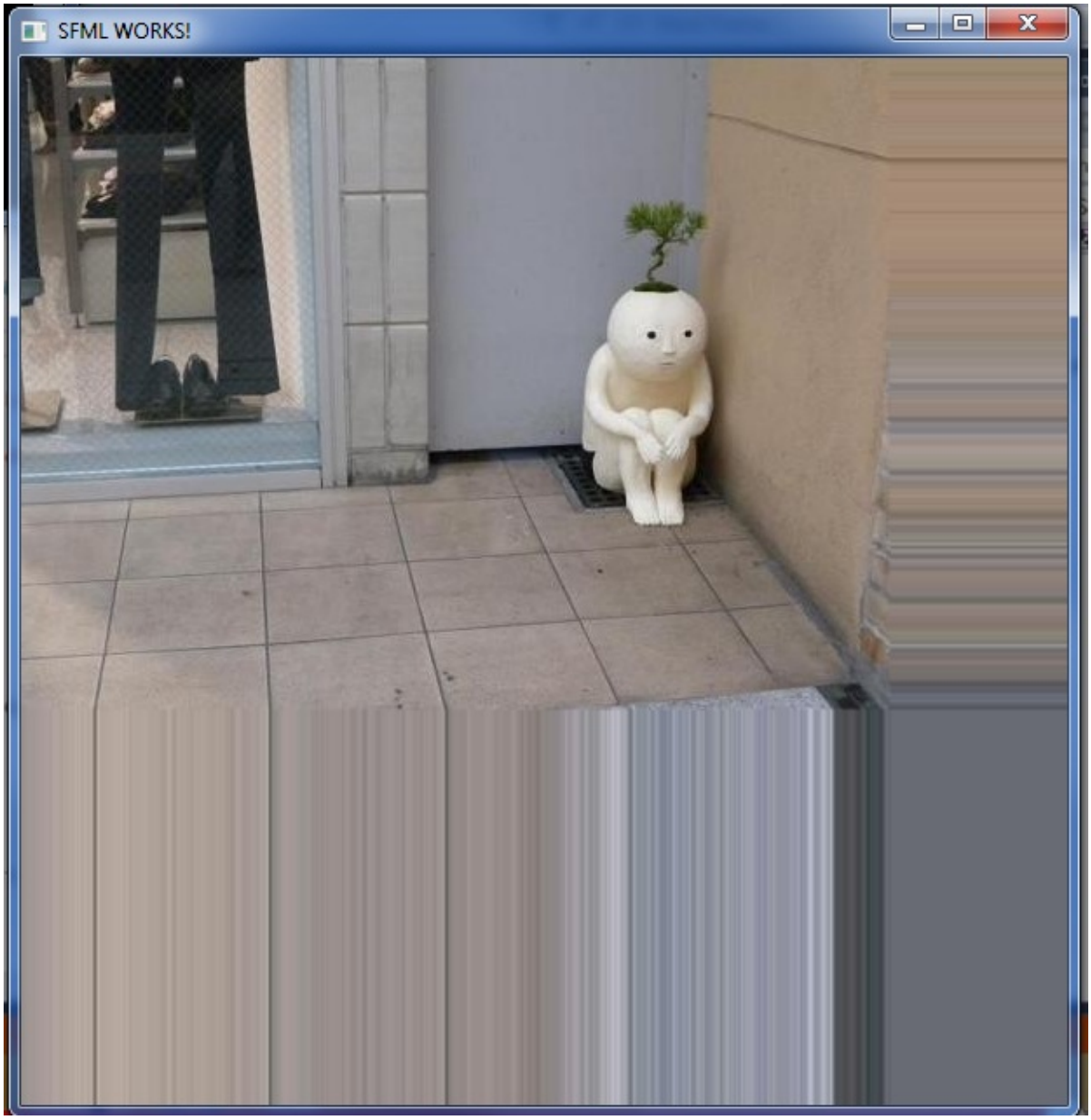
Looking at our example.jpg image, I reference its proporties, and determine its size to be 500x375 (width x height). IntRec will take (upper x, upper y, lower x, lower y), and constrain the image to the specified pixels.

So now, I've successfully cut my image into a quarter.



Now if I were to oversize the image, I would notice an interesting affect.

```
    sprite.setTextureRect(sf::IntRect(0, 0, 600, 600));
```



Here I've essentially resized the image to the size of the screen. What we see here is the last pixel that exceeds the resize becomes copied, and dragged to the position we specified last. None relevent pixels, (e.g. the bottom right) that are not part of the x and y position of the image are set to default or null color (grey).

However, SFML has an interesting feature that is particularly useless with backgrounds, and repreated forms of texture. We can enable this setting as such:

```
sf::Texture texture;
texture.setRepeated(true);
```

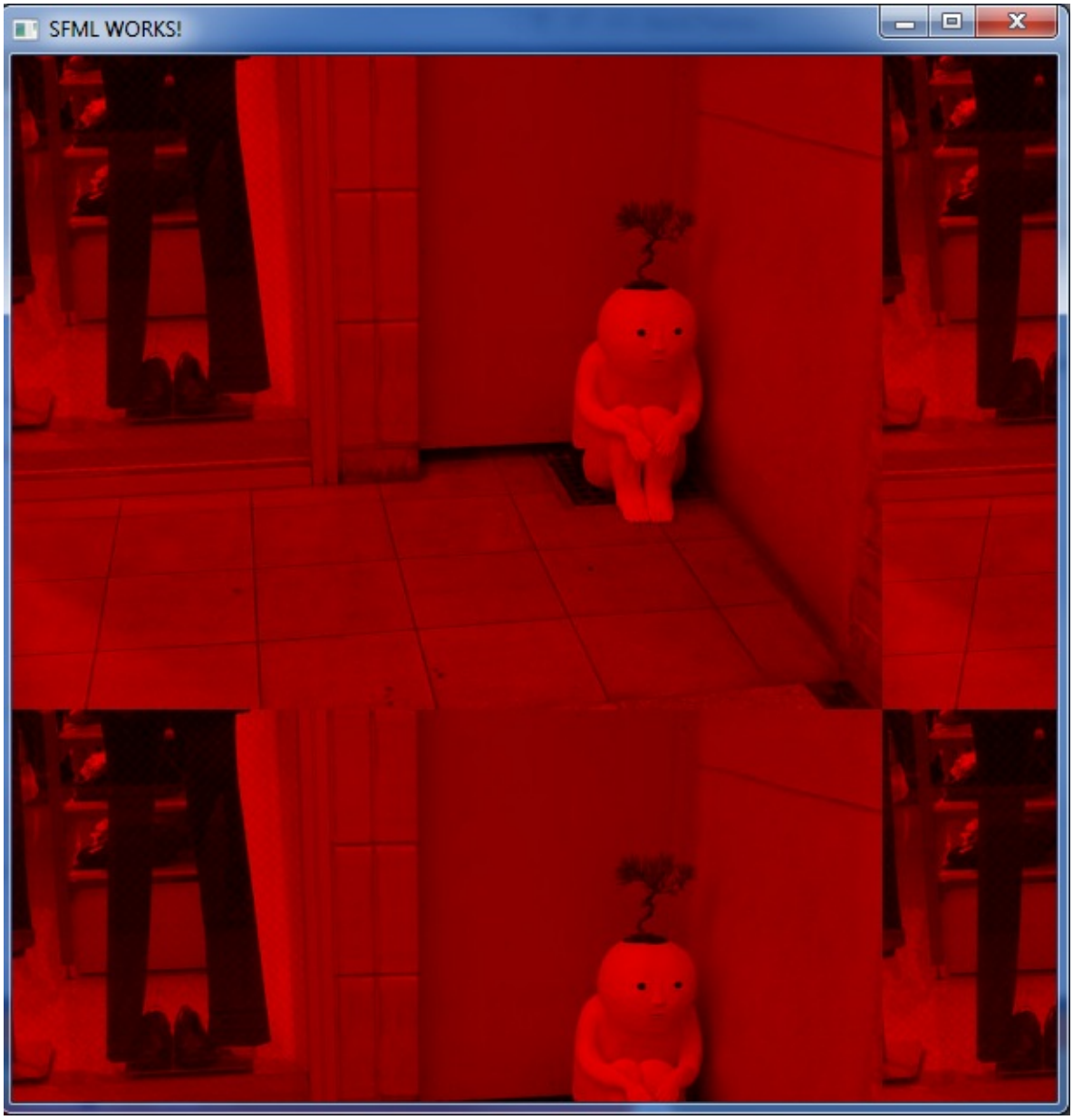
How this example isn't specifically helpful, but given a condition a background repetition can be repeated (e.g. grass), it can help save memory.

Set Sprite Color

It is common practice in most games to flash red when your health bar get affected, in order to communicate this information to the player. This takes advantage of sprite color control.

```
sprite.setColor(sf::Color(255, 0, 0));
```

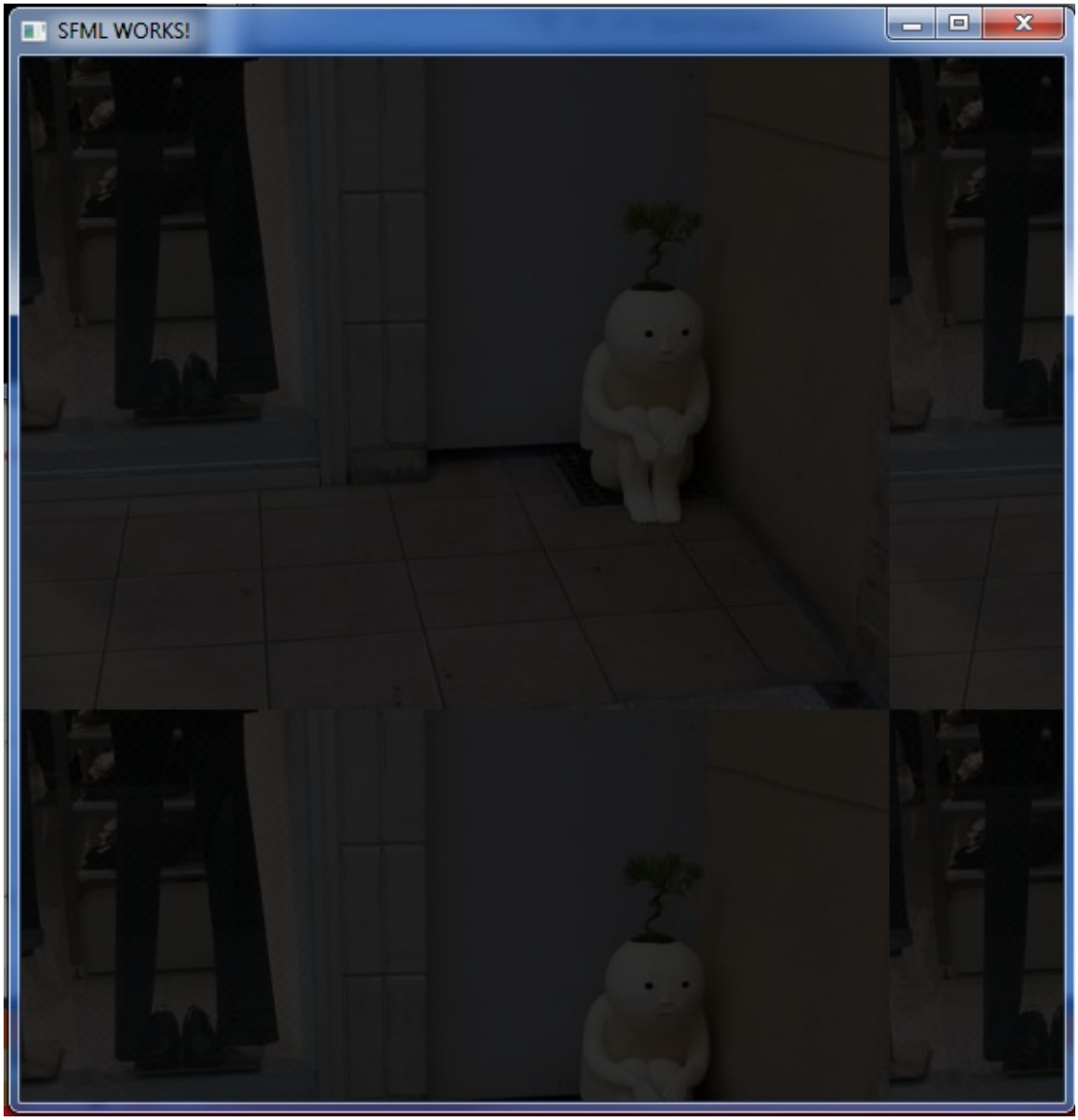
The parameters are specified in RGB (red, green, blue). Here we are only manipulating red, as blue and green are turned off. Note that the color intensity is set from [0-255].



Like with openGL, setColor has a fourth parameter that determines the opacity/transparancy of our sprite. Here is a demonstration:

```
sprite.setColor(sf::Color(255, 255, 255, 50));
```

Notice that I have so set RGB to its maximum state, otherwise known as its default state.

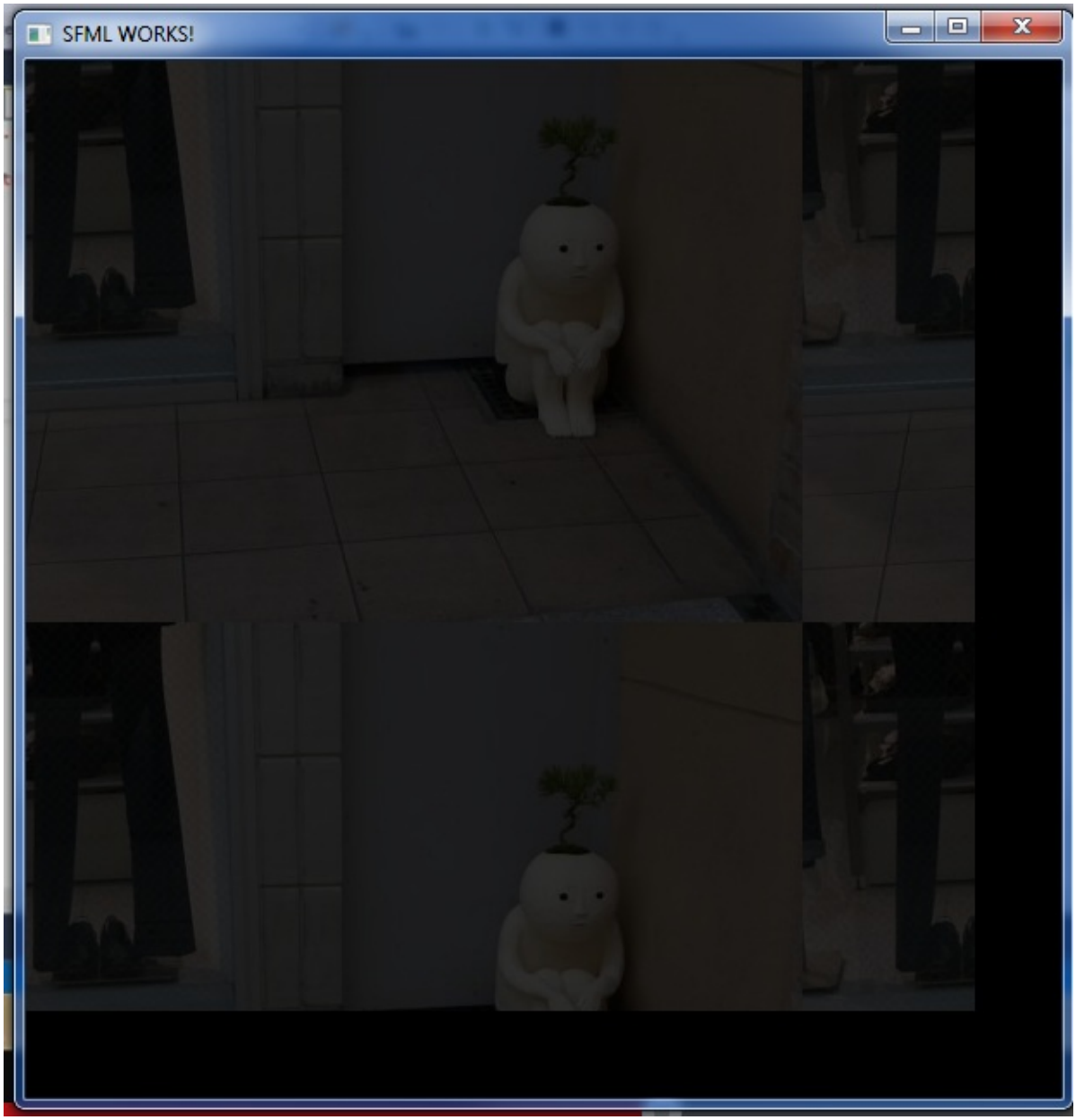


This can prove useful, if for example, we wanted to fade an object after it got collected, or killed, etc.

Setting Sprite Origin

In the previous examples, we've noticed that the sprite origin is always displayed at the origin (0,0), or aka anchor point.

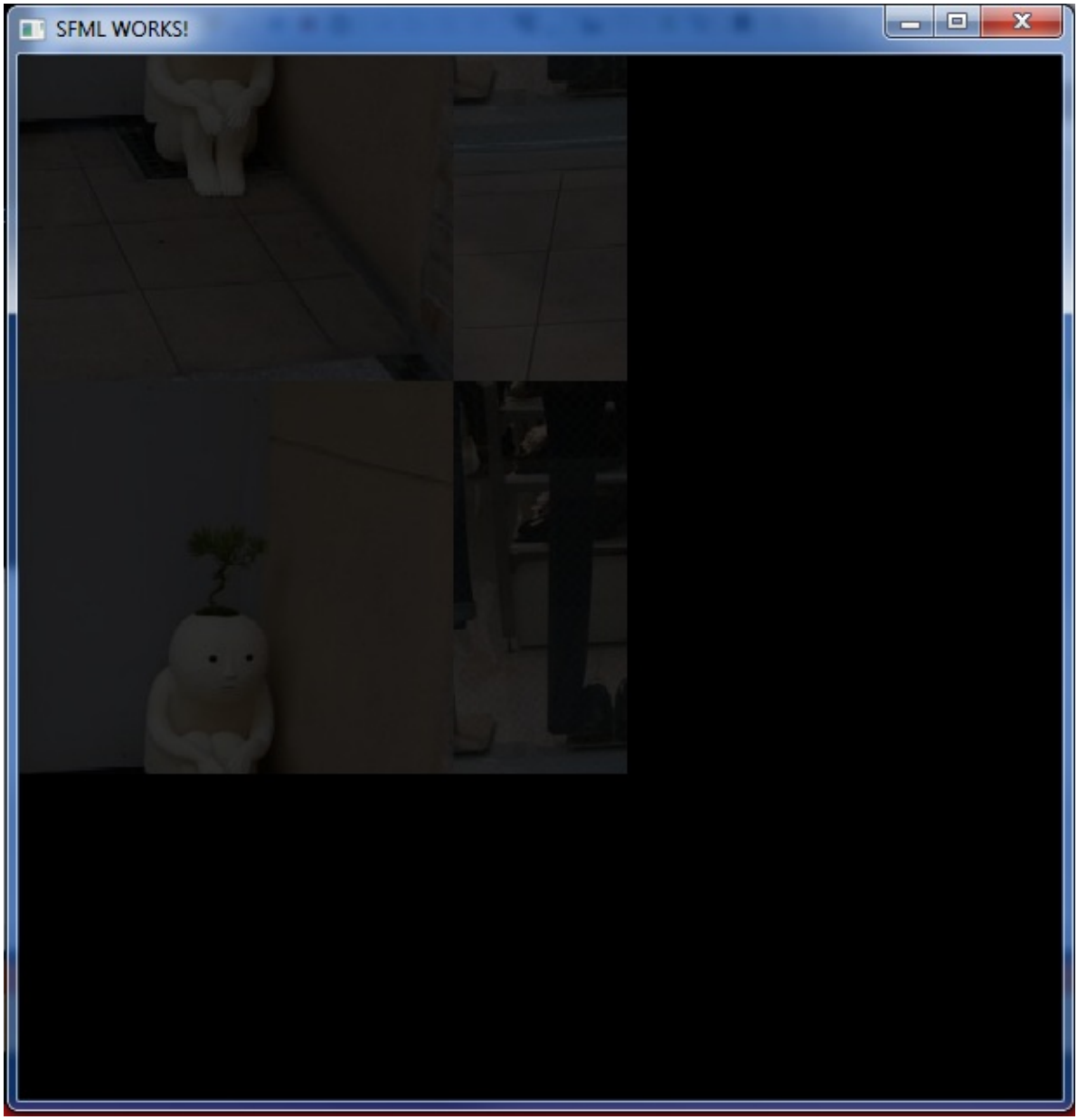
```
sprite.setOrigin(sf::Vector2f(50, 50));
```



As shown the position is moved reletive to the bottom right corner. Also, the difference between vector2f and 2i, etc are the template types, that can change freely from integers (i) and floats (f).

Handling pixels can prove to be a little tedious, so instead what we can do is manipulate the origin reletive to the size of the texture/sprite, and take the percentage of the pixel width, in our case, 50%.

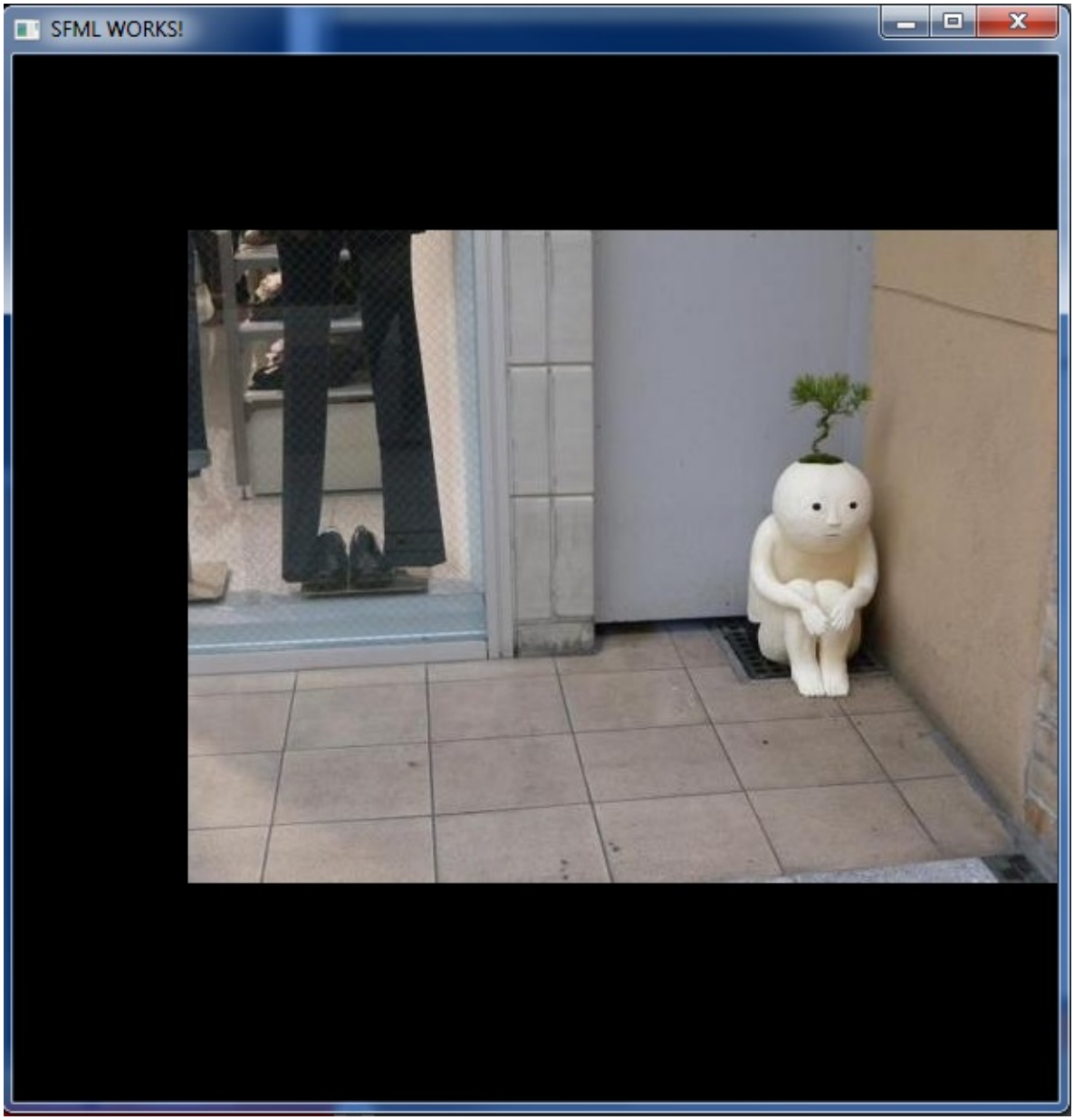
```
sprite.setOrigin(sf::Vector2f(sprite.getTexture()->getSize().x * 0.5, sprite.getTexture()->getSize().y * 0.5));
```



Setting Sprite Position

Personally, I find the functionality of setPosition more useful than set origin. It does simply what you might expect it to do. In addition, it is more literate, as it follows the positioning rules of the original window display.

```
sprite.setPosition(sf::Vector2f(100, 100));
```



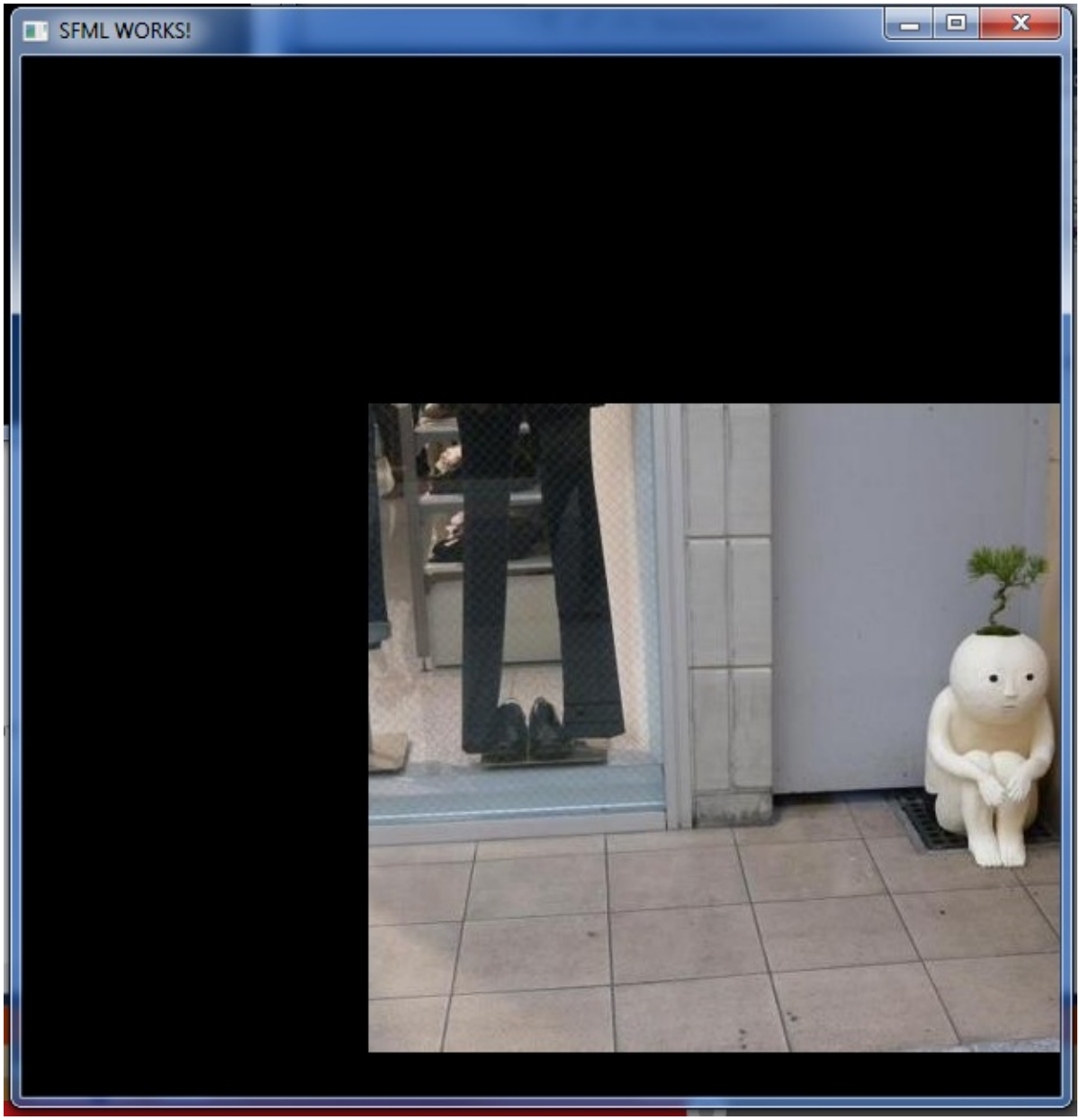
Furthermore, we can take this information (returns a vector 2f), and use it to any means.

```
cout << sprite.getPosition().x << " " << sprite.getPosition().y;
```

Moving a Sprite

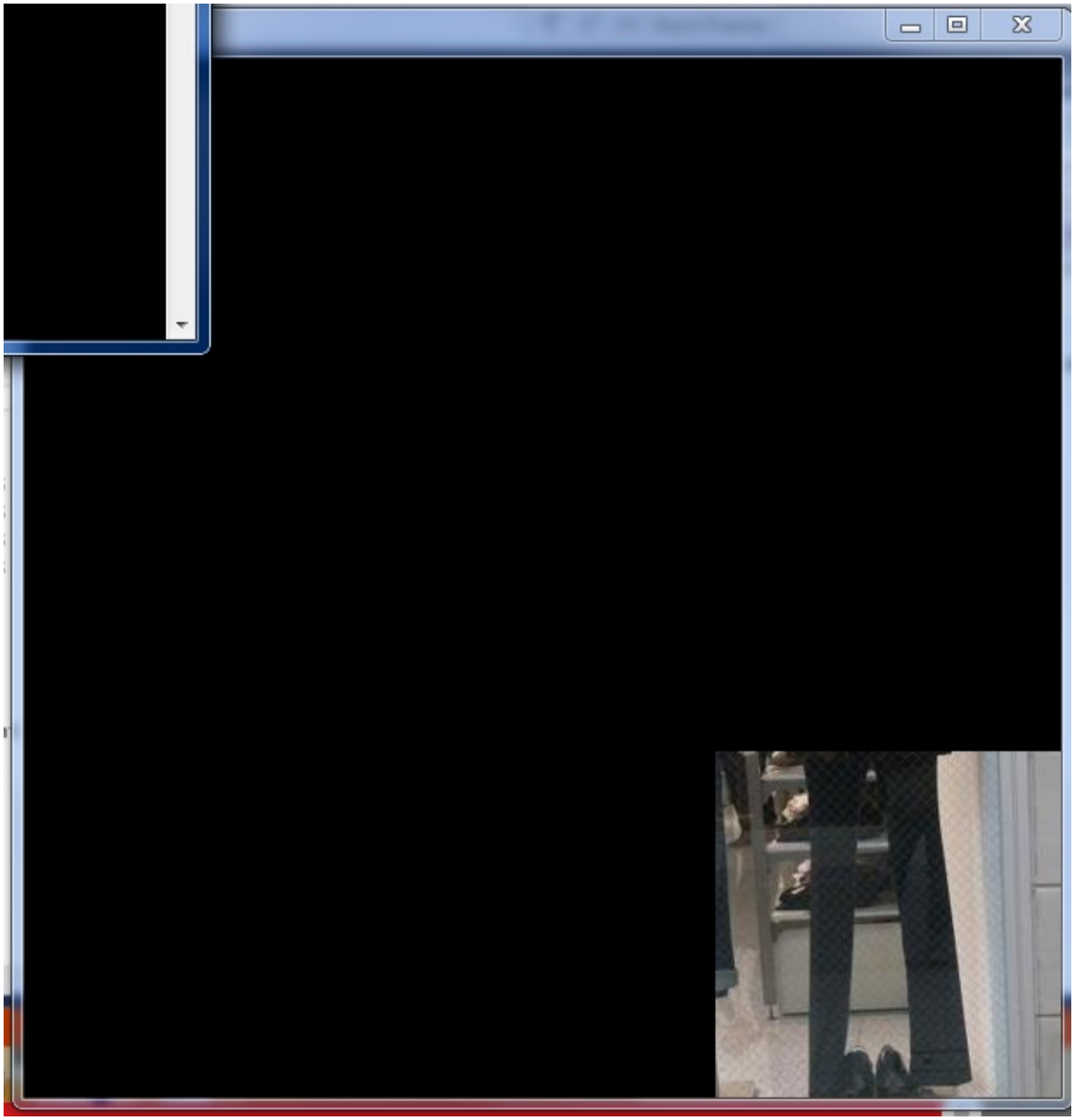
Originally, we have originating the initial position of the image. Given that we wanted to move it, reletive to the setPosition(), (and NOT the graphics window!).

```
sprite.move(sf::Vector2f(200, 200));
```



Because of this property, stacking the image will consistantly move the image. It can therefore be used as very helpful method with animation.

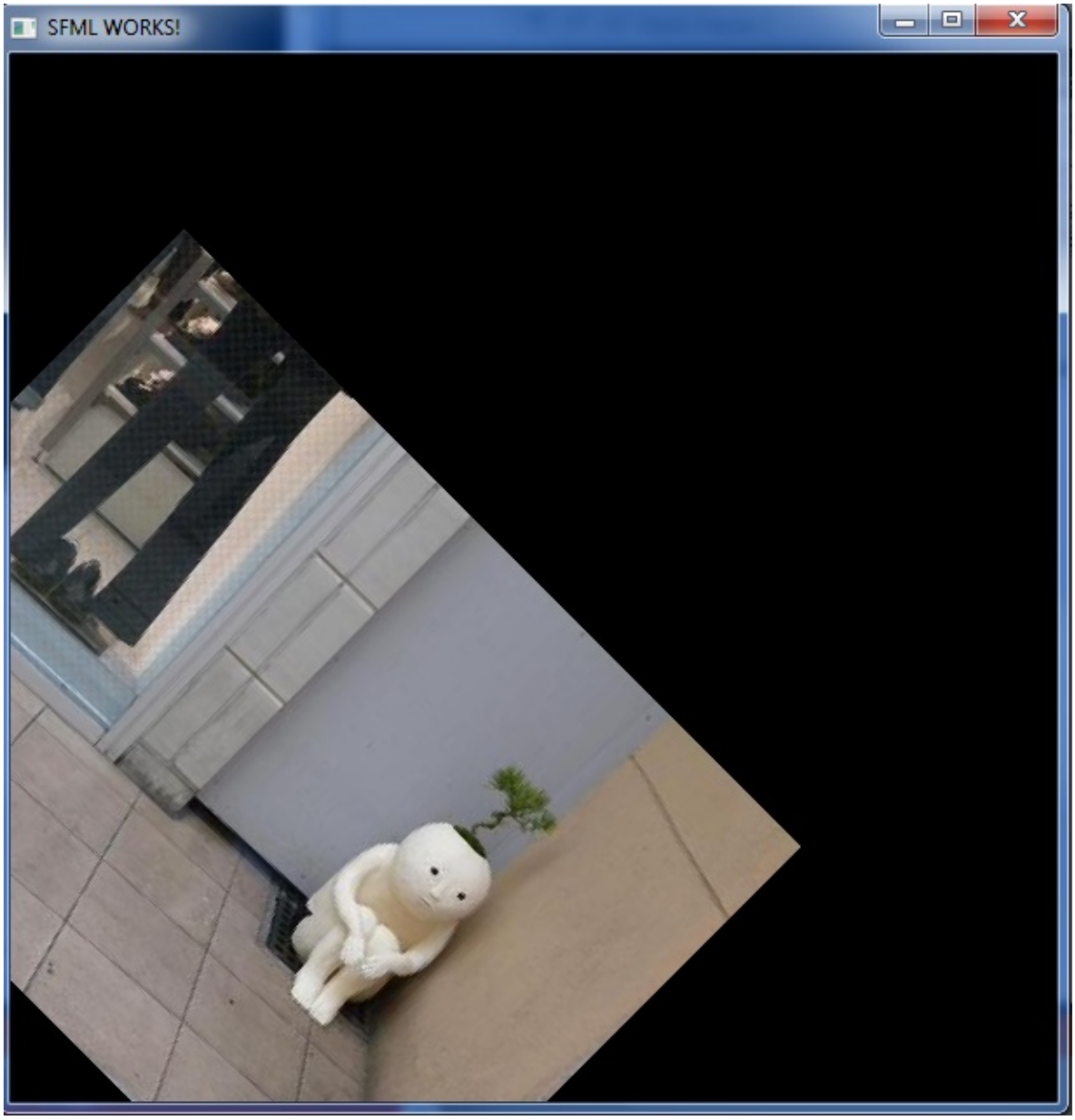
```
sprite.move(sf::Vector2f(100, 100));
sprite.move(sf::Vector2f(100, 100));
sprite.move(sf::Vector2f(100, 100));
```



Sprite Rotation

Quite simply:

```
sprite.setRotation(45);
```

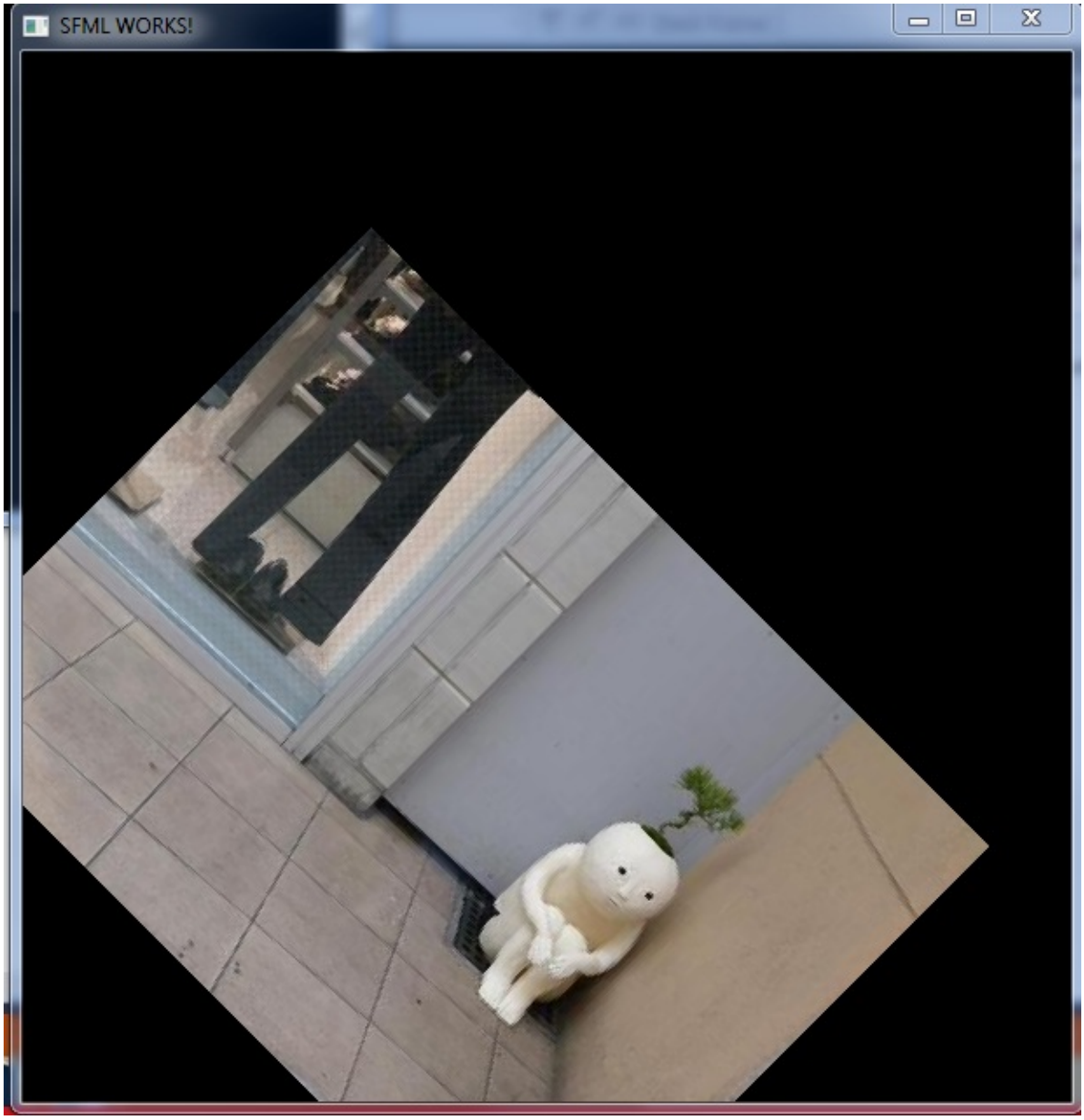


This is unlike the unit circle I have learned in high school. Notice the counter clockwise rotation. Can you determine why the rotation appears to be slightly off the origin point (0,0)? We are achieving this effect because of:

```
    sprite.setPosition(sf::Vector2f(100, 100));
```

the rotation is reletive to the position we've set our texture to be. To demonstate this:

```
    sprite.setPosition(sf::Vector2f(200, 100));
```



However, unlike with .move(), this does not stack, and fixed indefinitely reletive to degree 0.

// Redundent:

```
sprite.setRotation(45);
sprite.setRotation(45);
sprite.setRotation(45);
```

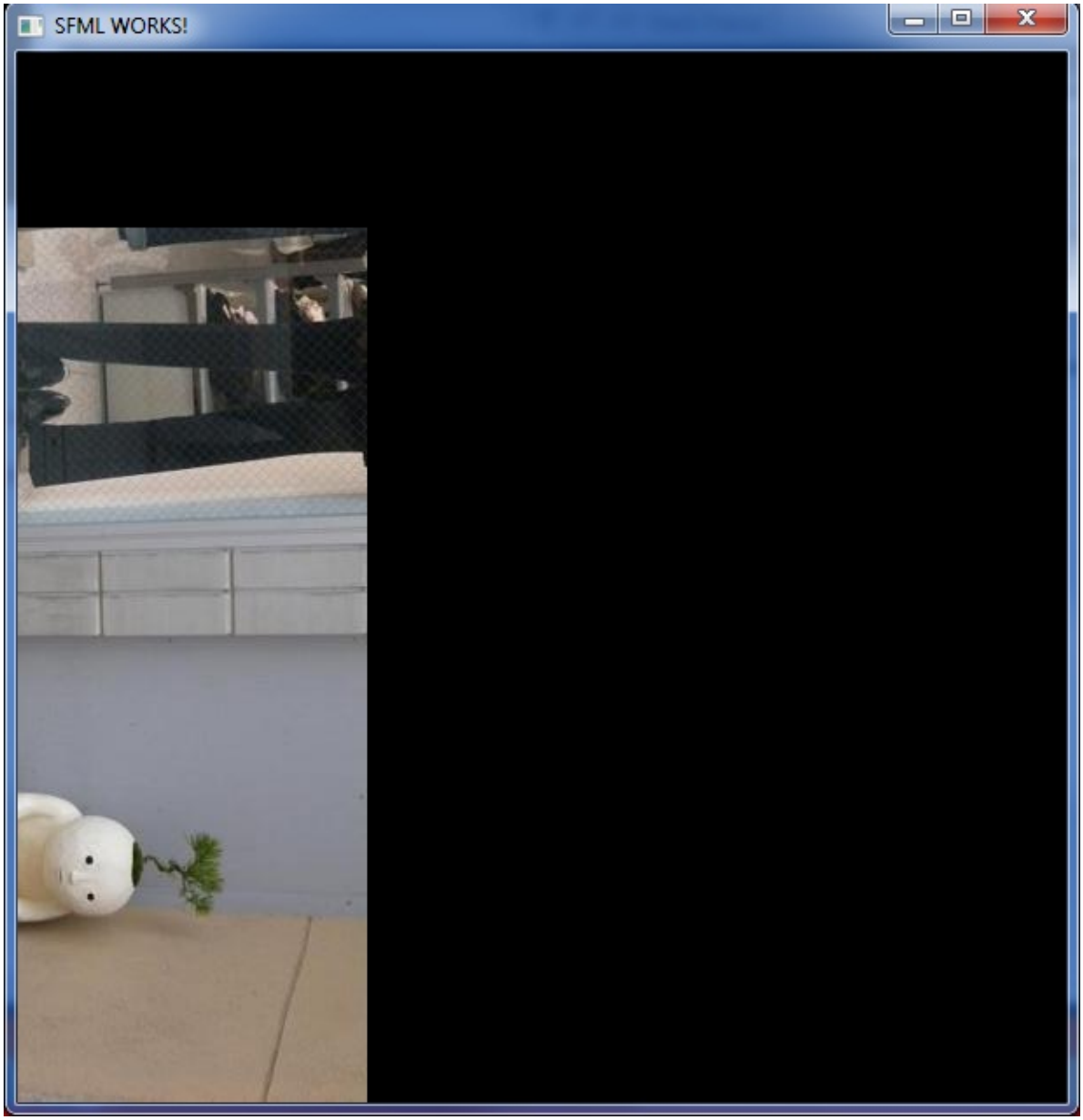
We can obtain the current rotation:

```
cout << sprite.getRotation() << endl;
```

Just like with move(), and setPosition(), there is a feature to stack rotations.

```
sprite.rotate(15);
sprite.rotate(15);
sprite.rotate(15);
```

// Initially set at 45, rotated 15 x 3 = total of 90, therefore vertical.

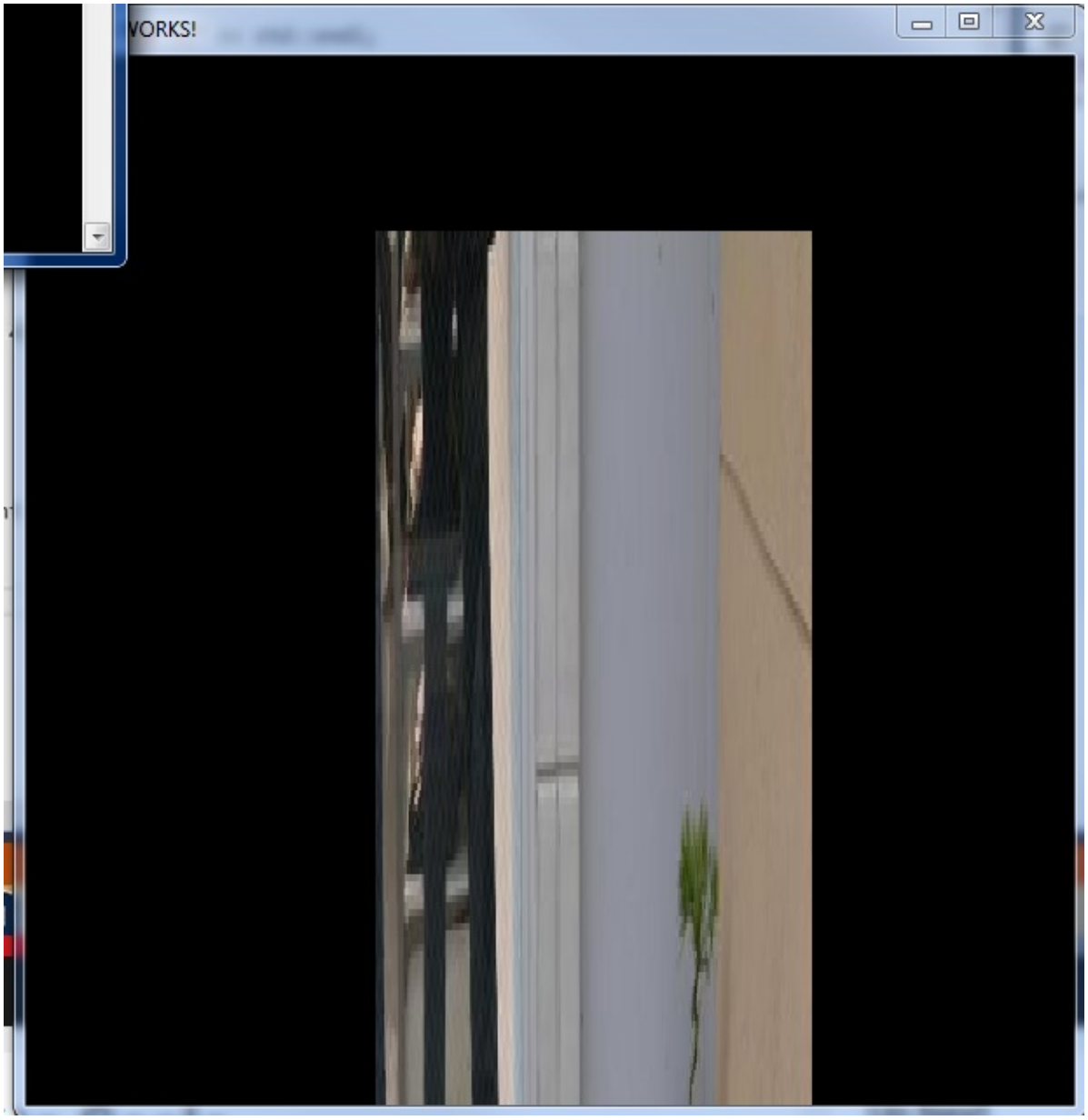


Sprite Scale

Aka, resizing your texture without proportion constraints. It is, however possible to have propotional constraints by making sure that both x and y parameters are the same.

Like with **setPosition** and **setRotation**, this property is does not stack - but can be through other functionale.

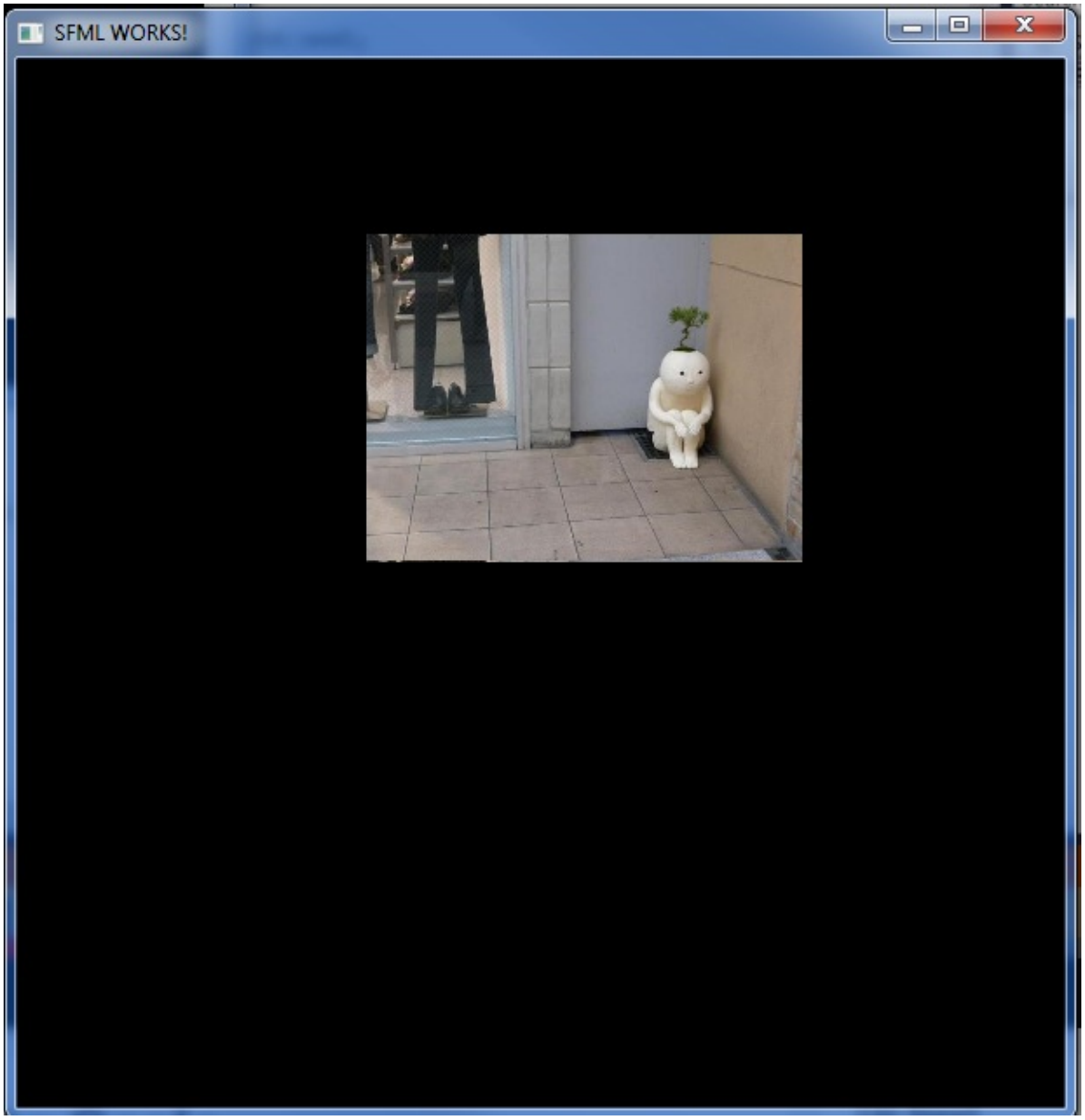
```
sprite.setScale(sf::Vector2f(0.5, 4));
```



Here, sprite.getTexture()->getSize().x is halfed, while sprite.getTexture()->getSize().y is scaled by 4.

Here is a demonstration of proportional scale.

```
sprite.setScale(sf::Vector2f(0.5, 0.5));
```

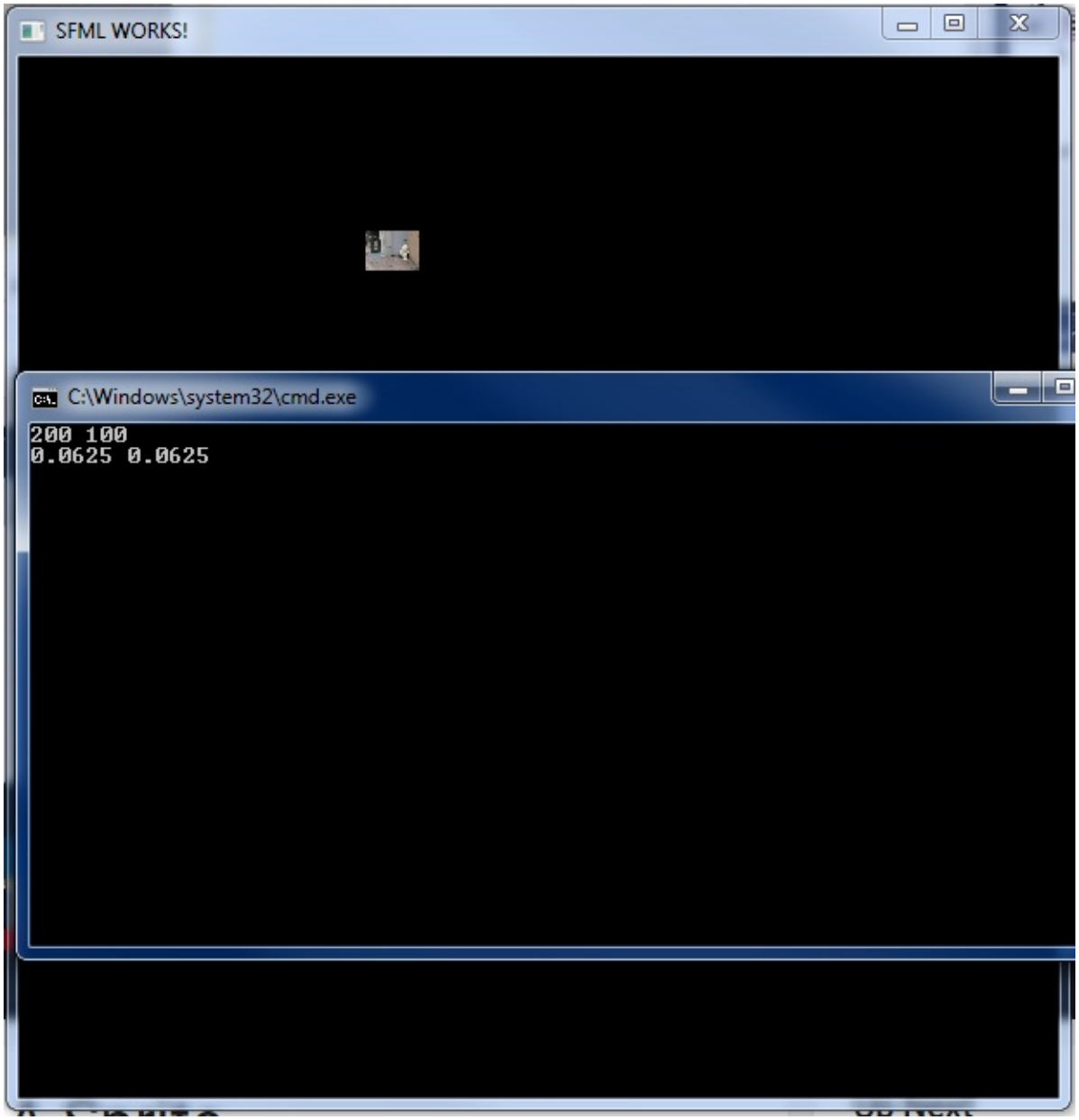


Naturally, we can retrieve the information.

```
cout << sprite.getScale().x << " " << sprite.getScale().y << endl;
```

Like before, lets demonstrate a scale stack:

```
sprite.scale(sf::Vector2f(0.5, 0.5));
sprite.scale(sf::Vector2f(0.5, 0.5));
sprite.scale(sf::Vector2f(0.5, 0.5));
```



In addition to the first set scale, the stack is applied 4 times. Thus achieving $$0.625 = 5^4 $$

Other Options

- Use sf::image if you want to:
  - load from RAM (as opposed to disk with sf::Textures)
  - manipulate individual pixels.
  - load a large image from data (sf::texture has a limited capacity of sf::Texture::getMaximumSize(). This can serve useful for large static background image, for example.
  -

Basic Shapes

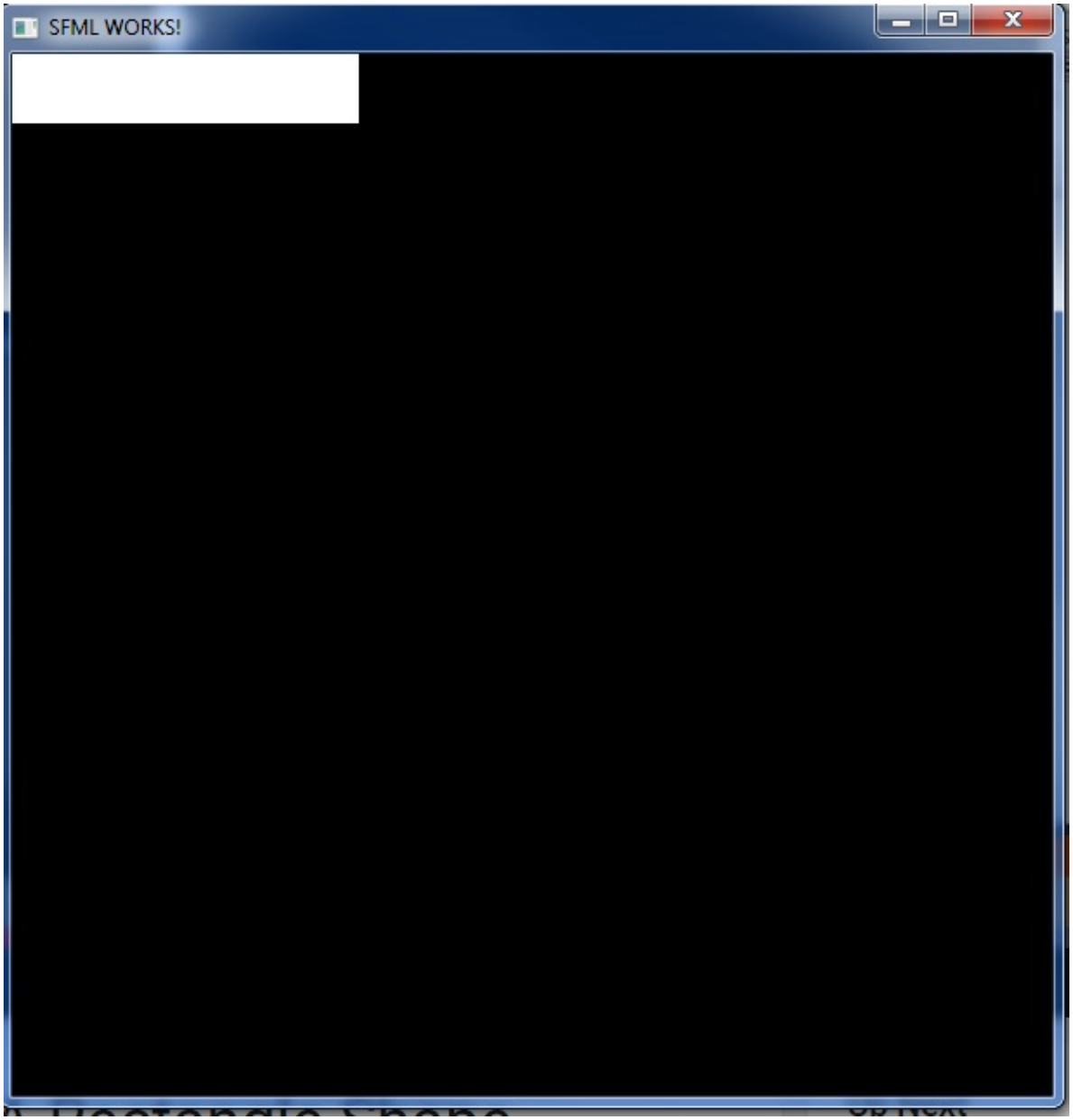
Rectangles

Lets work from:

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

using namespace std;


int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();

                break;
            }
        }
        window.clear();

        window.display();
    }
}
```

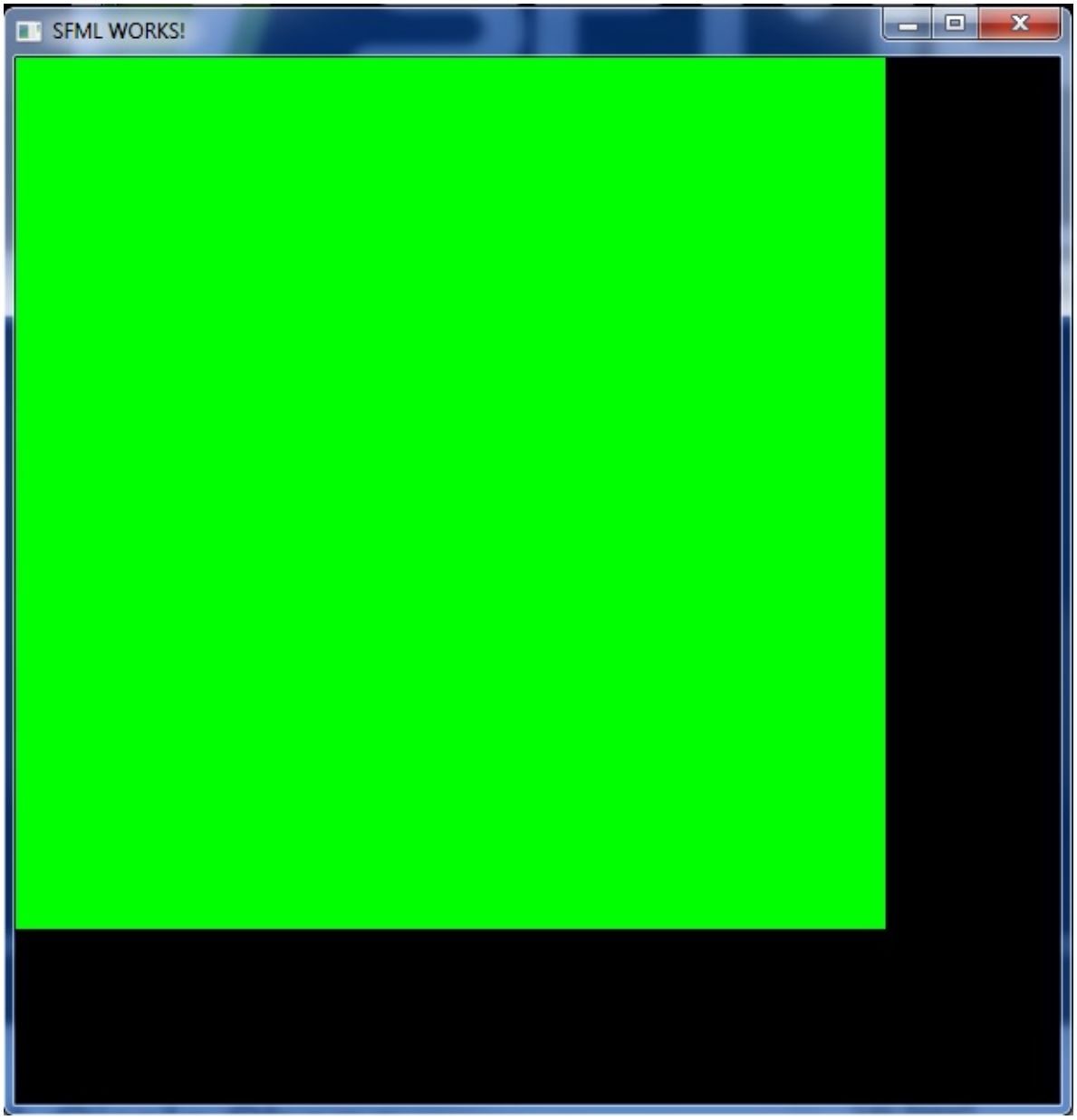
SFML provides utilaties for basic shapes, and like with sprites, there definitions are declared, and then drawn in the window.

```
    sf::RectangleShape rectangle(sf::Vector2f(200, 40));

    ...

    window.draw(rectangle);
```



Noticably, its default color is white. We can add properties to our rectangle to customize our object. These can include functions like scale, rotation, and or color.

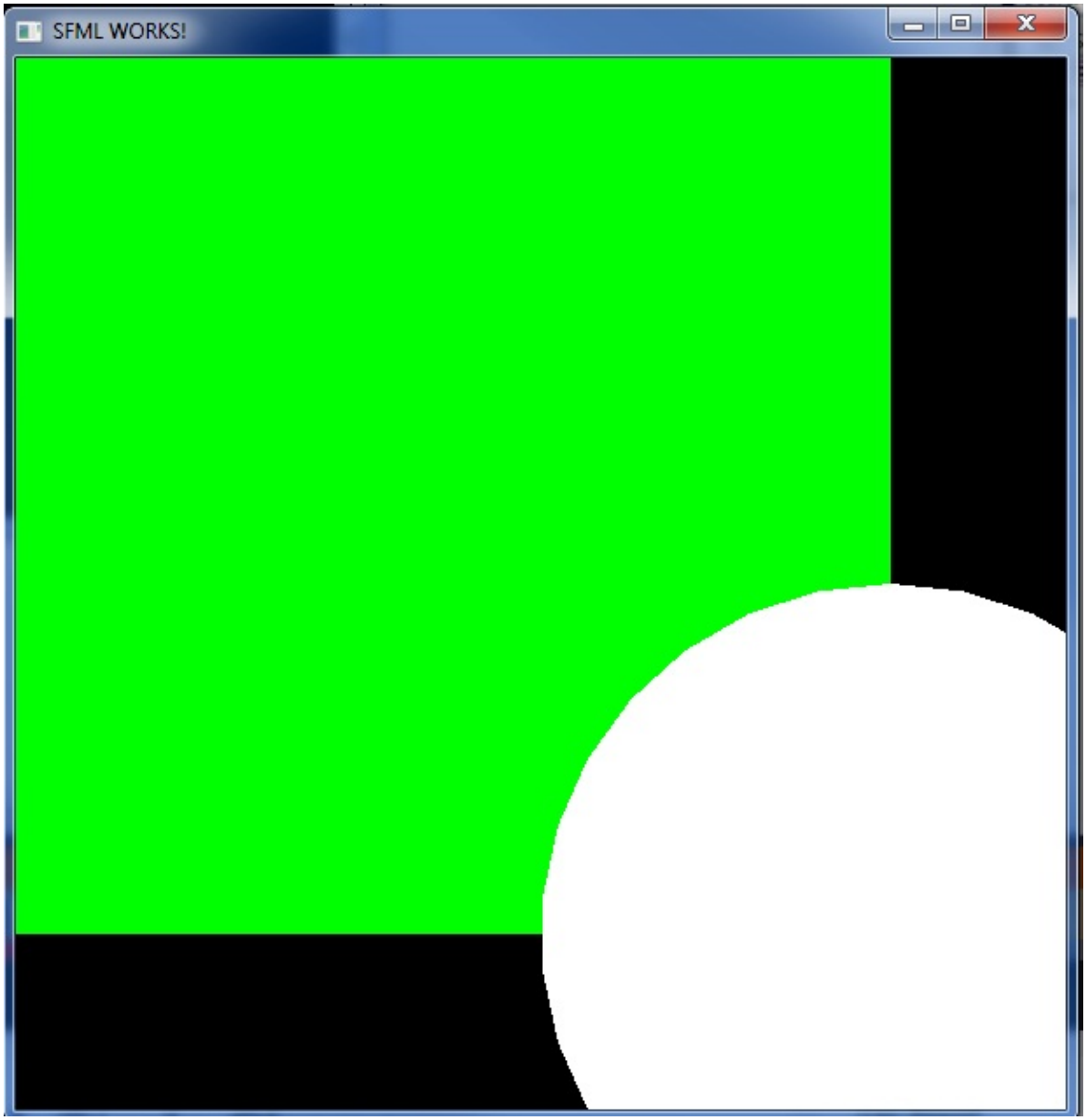
```
rectangle.setSize(sf::Vector2f(500, 500));
// Use RGB method in establishing color (with an additional 4th parameter to specify transparency)
rectangle.setFillColor(sf::Color(0, 255, 0));

// Another method in setting color:
//rectangle.setFillColor(sf::Color::Cyan);
```



Circles

Lets set some properties:

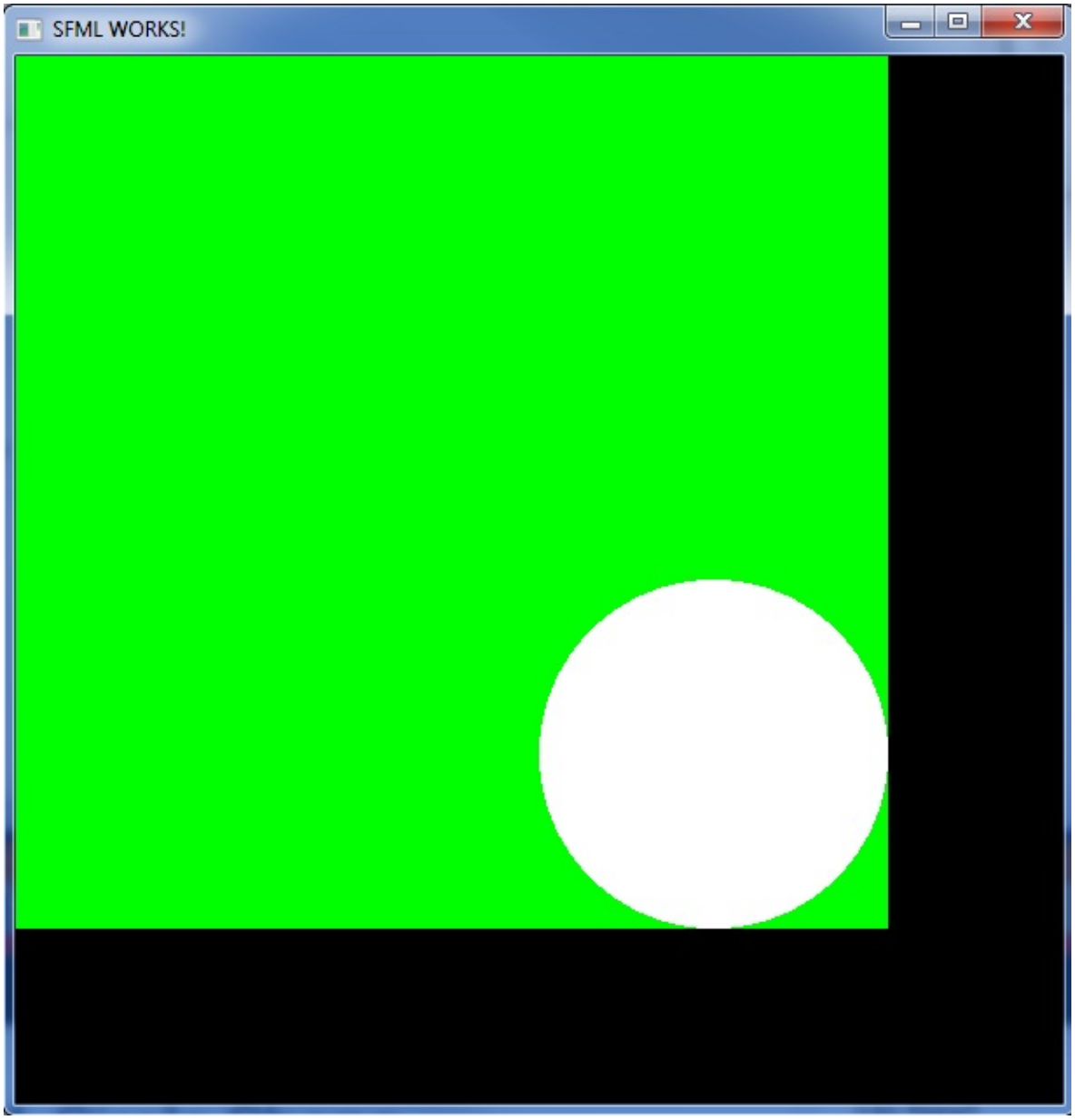
```
sf::CircleShape circle(200);
circle.setPosition(sf::Vector2f(300, 300));

...

window.draw(circle);
```



We can notice that the circle itself is roughly jagged. This is because a circle is made up with lines and points. The amount we see here is enough to make a noticable difference.

Luckly, this property can be controlled by setting the point count:

```
    circle.setRadius(100); // demonstrating radius adjustment
    circle.setPointCount(500);
```



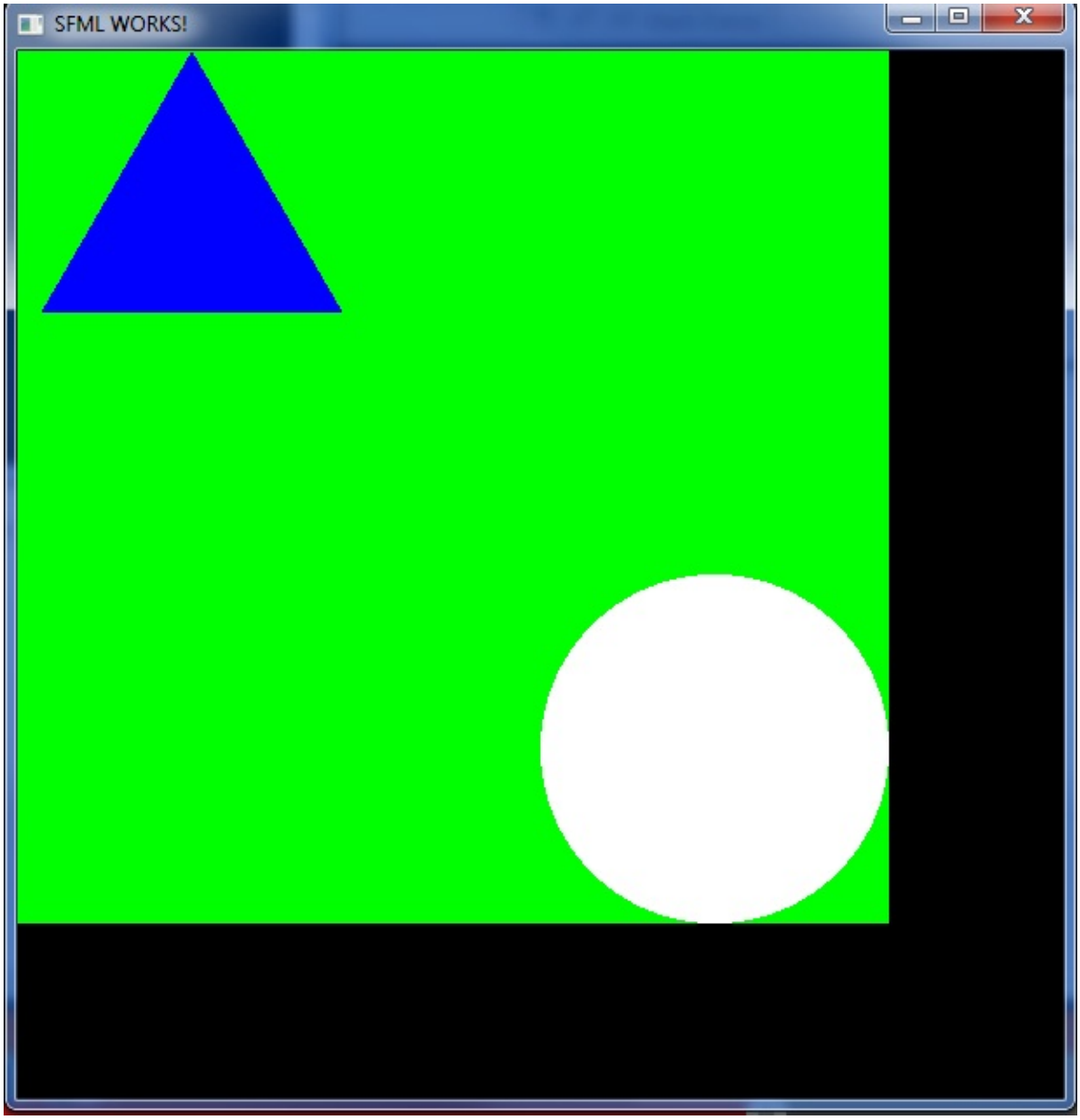
Notice those dense pixels. mmmhm.

Polygons

Interestingly polygons are generated from CircleShape. This is because we can define a circle to be a polygon with $$ lim x->+inf $$ number of sides. This is what defines pi.

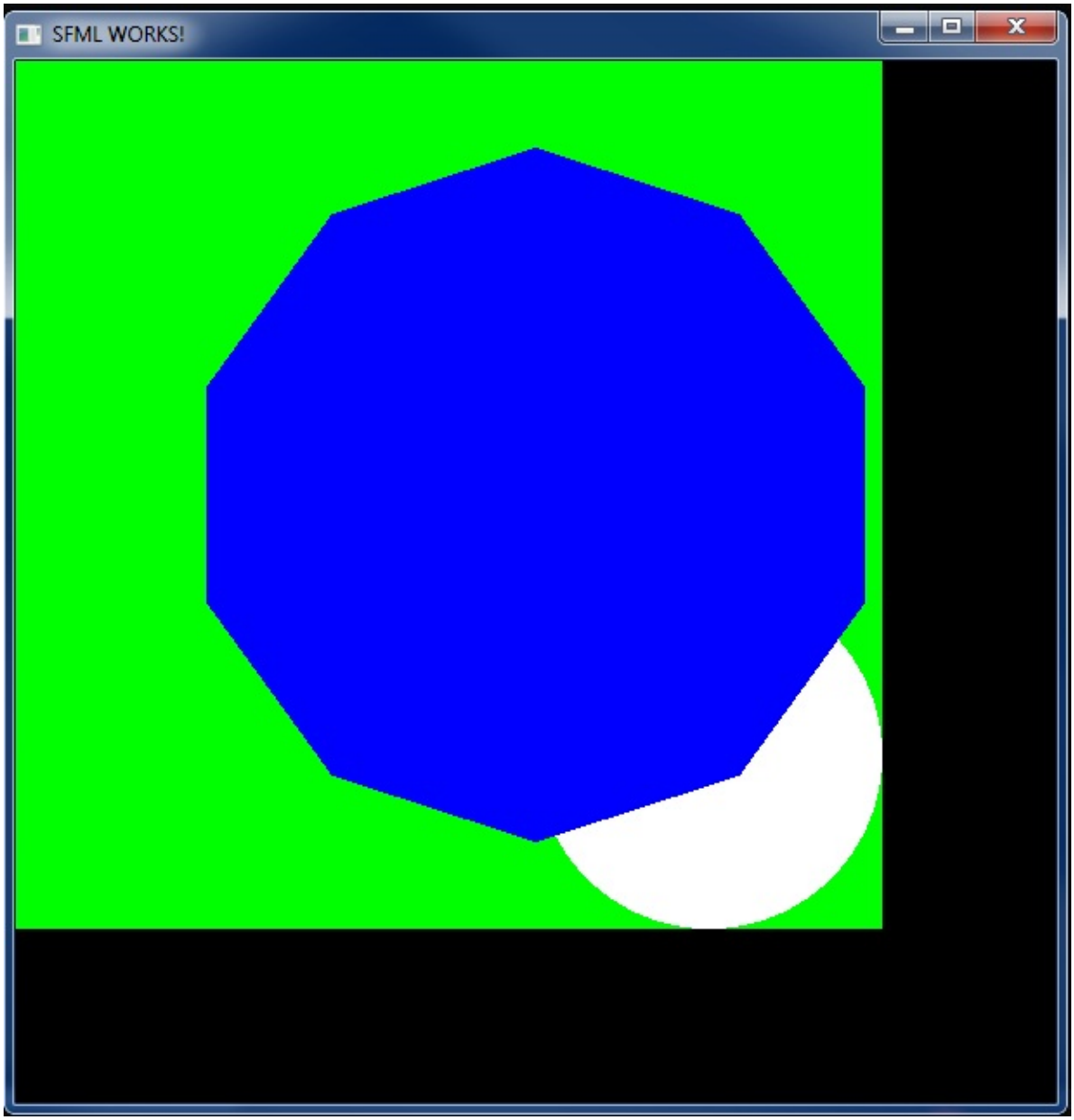
So.

```
sf::CircleShape polygon(100, 3);
polygon.setFillColor(sf::Color(0, 0, 255));

...

window.draw(polygon);
```



More and more we can notice the gradual formation for a circle.

```
sf::CircleShape polygon(200, 10);
polygon.setFillColor(sf::Color(0, 0, 255));
polygon.setPosition(sf::Vector2f(100, 50));
```



Convex Shapes

aka, abnormal, or custom shapes. To do this, we declare an instance of the convex shape, and then specify the number of points.

```
sf::ConvexShape shape(6);
//shape.setPointCount(6); ALTERNATIVE OPTION
shape.setPointCount(6);
```

Next, we can customize the coordinates of each shape:

```
shape.setPoint(0, sf::Vector2f(40, 70));
shape.setPoint(1, sf::Vector2f(70, 100));
shape.setPoint(2, sf::Vector2f(100, 150));
shape.setPoint(3, sf::Vector2f(150, 300));
shape.setPoint(4, sf::Vector2f(500, 550));
shape.setPoint(5, sf::Vector2f(575, 580));
```

Lines

While SFML does have built in functionality with lines, we can simulate it's effect using rectangles.

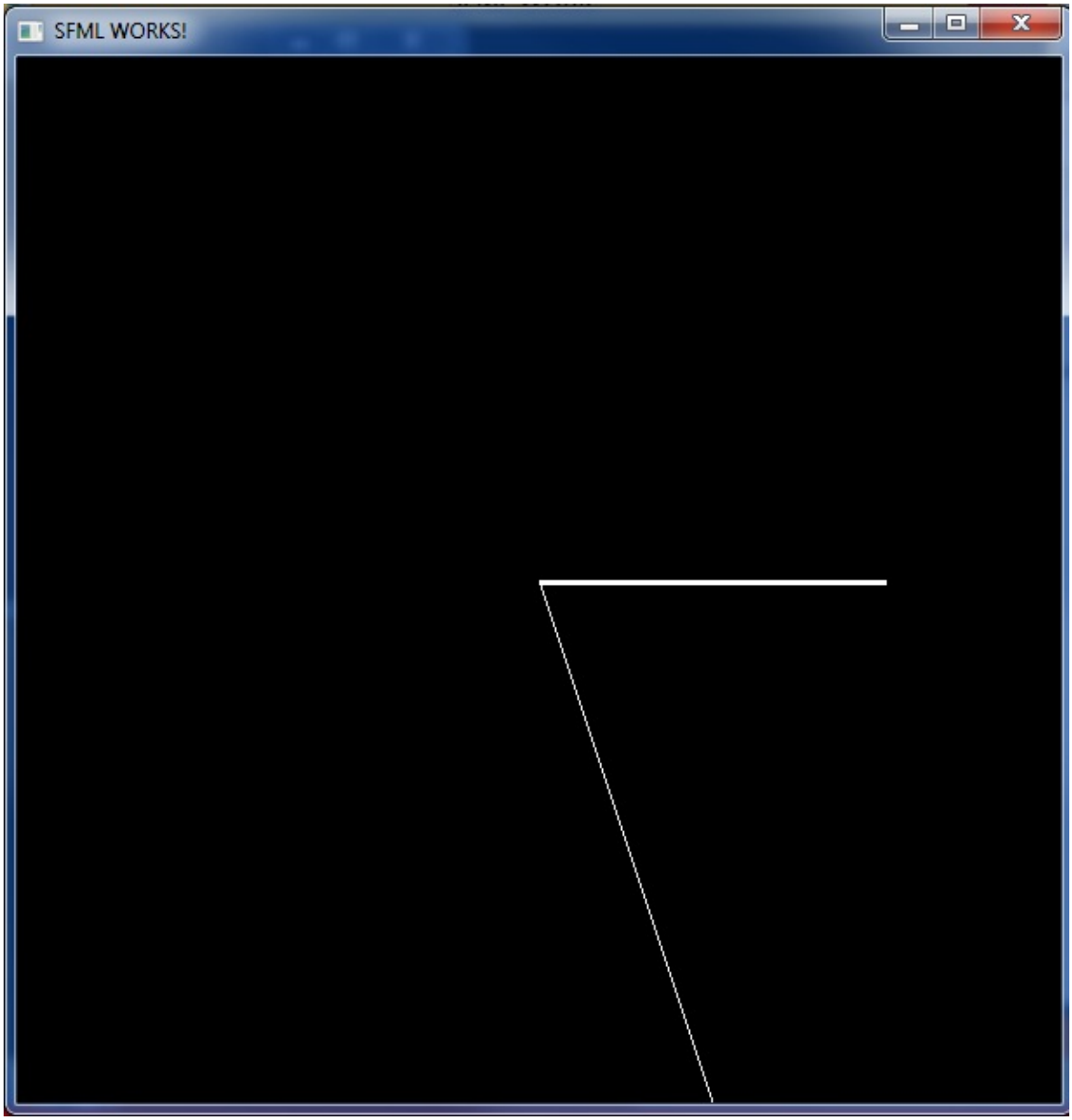
```
sf::RectangleShape line;
line.setSize(sf::Vector2f(200, 3));
line.setPosition(sf::Vector2f(300, 300));

...

window.draw(line);
```

The second way to similiar a line is to use a vertex array.

```
sf::Vertex line2[] =
{
    sf::Vertex(sf::Vector2f(300,300)),
    sf::Vertex(sf::Vector2f(400, 600)),
};
```

Here we've defined two vertex points, that defines the start and end point of the line. Later, we define the line attribute to this vertex, in order to let the computer know these two points must be connected.

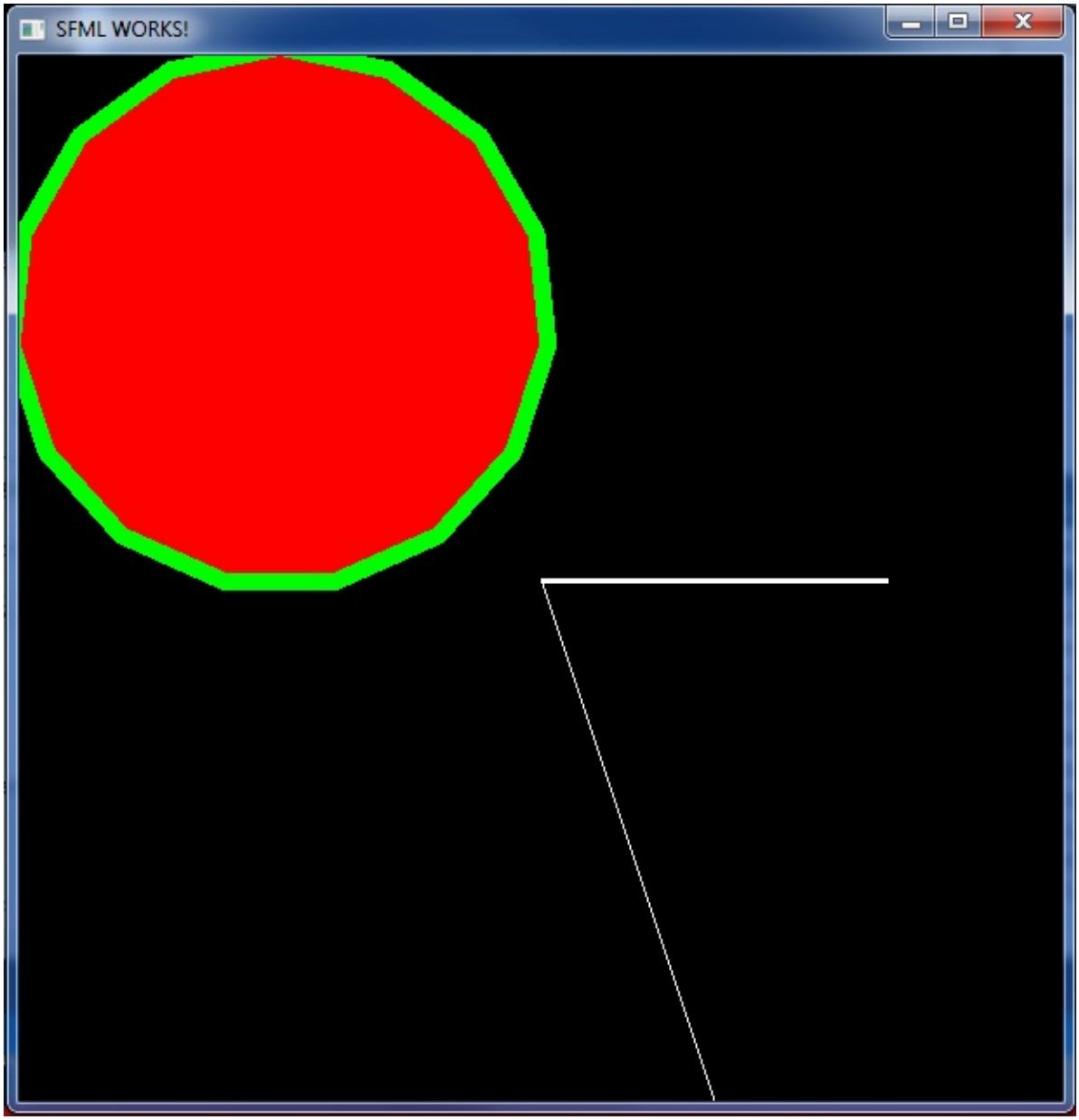
```
    window.draw(line2, 2, sf::Lines); // array, size, type
```



Color and Stroke

In the previous examples, we've learned about color fill. With color strokes, it first nessessary to specify a stroke thinkness, because by default, it doesnt exist, or is 0. Afterwards, we can define the color.

```
sf::CircleShape polygon(150, 15);
polygon.setFillColor(sf::Color::Red);
polygon.setOutlineThickness(10);
polygon.setOutlineColor(sf::Color::Green);
```

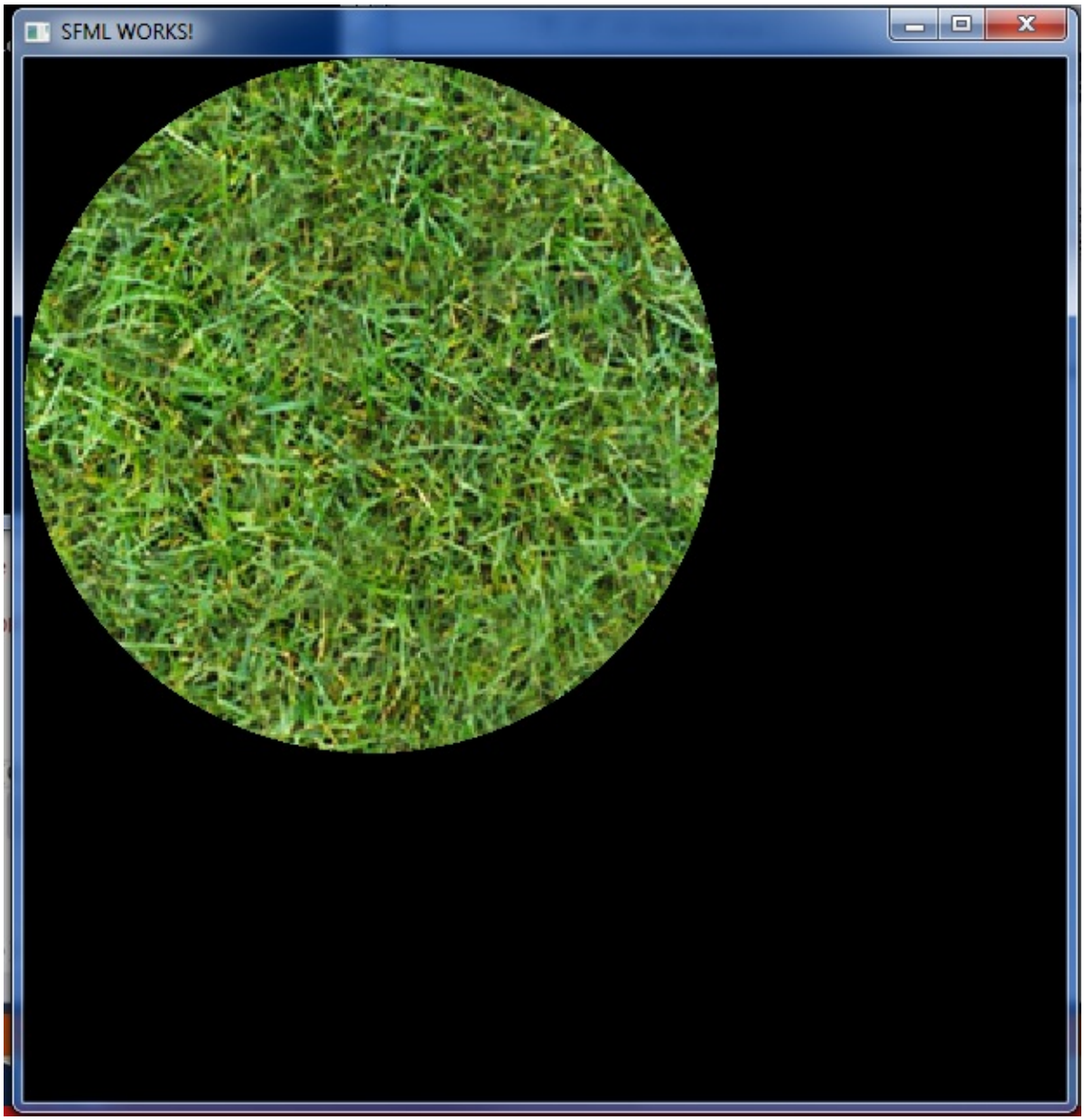


Notice how the stroke fits outwards from the default edge of the fill. This is because the outlineThinkness parameter is positive. A negative value would set the stroke inwards.

Combining Sprites and Shapes (Shape Textures)

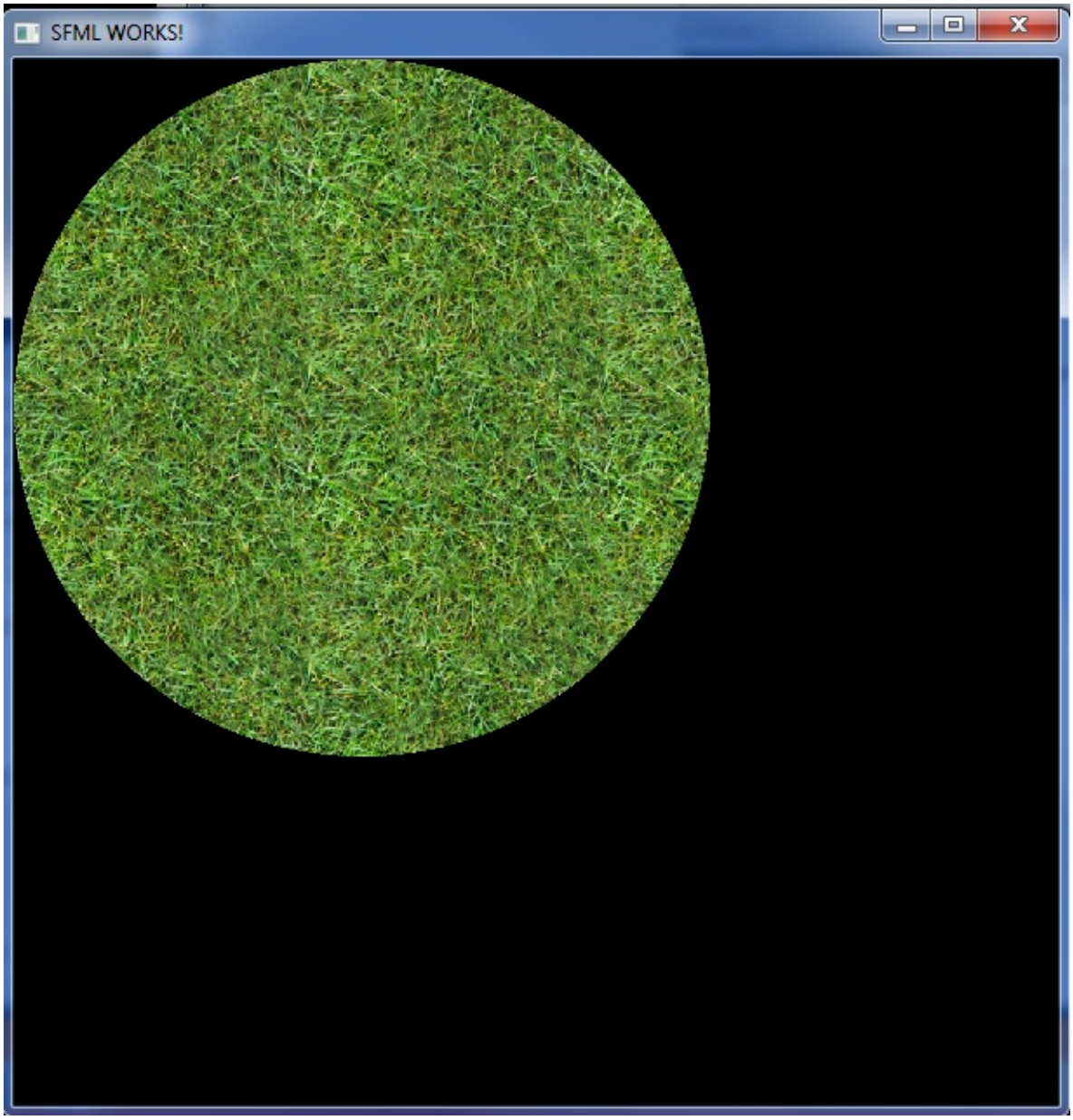
Using any SFML provided arbitrary shape, we can crop, or cut an image essentailly as a color fill with a provided image. Using a circle, for example, we can define its fill or 'texture' with an image.

```
sf::CircleShape circle(200);
circle.setPointCount(300);

sf::Texture texture;

if (!texture.loadFromFile("grasstex.jpg"))
{
    cout << "File not successfully loaded" << endl;
}

circle.setTexture(&texture);

...

window.draw(circle);
```



Like before, we still have all the available properties to manipulate this graphics. To demonstrate one, lets size the original image to the length of the window, and repeat the texture. We will notice a 'greater resolution image' that is in it's more primative form, just the repeation of a single texture.

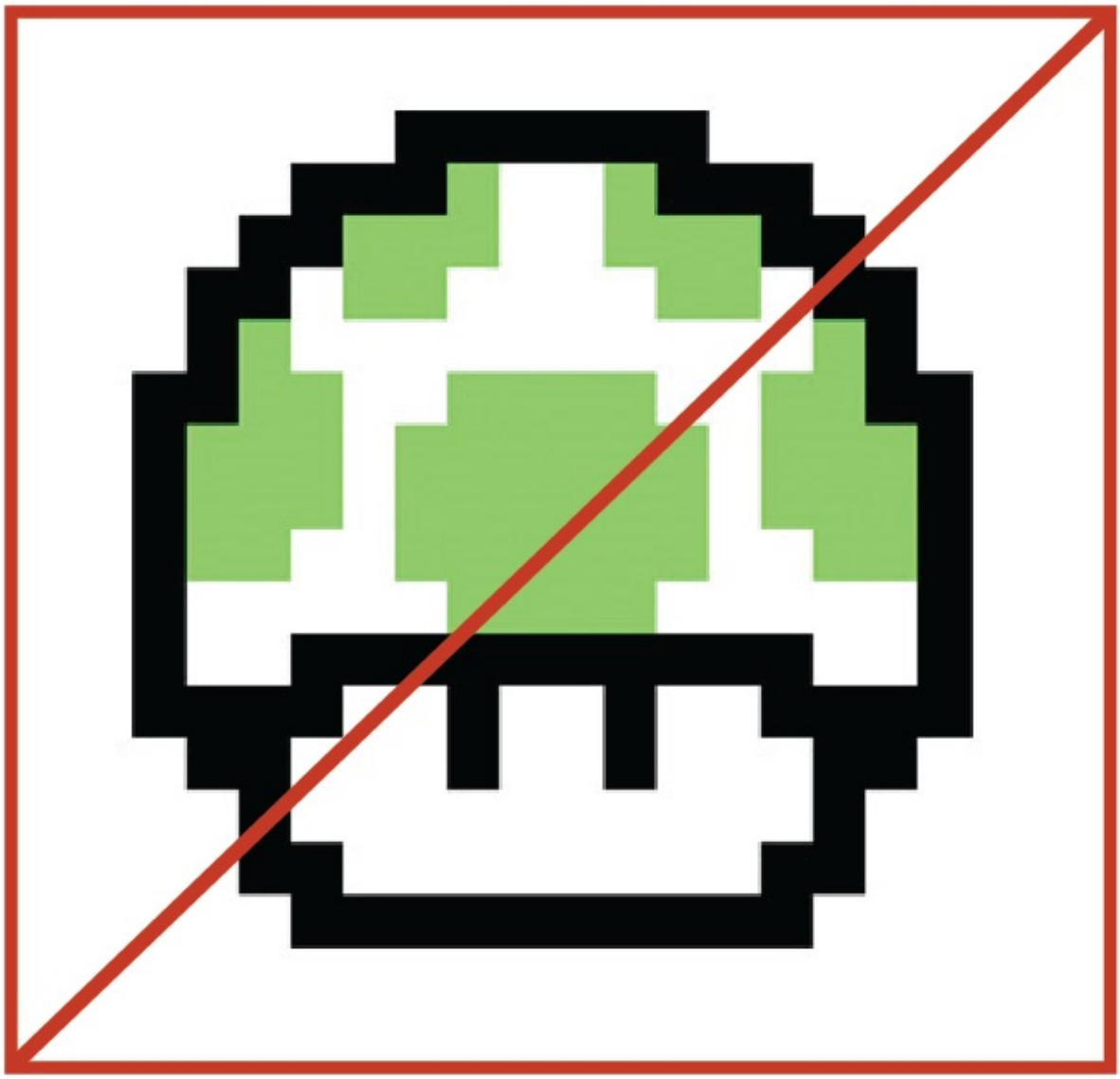
```
sf::Texture texture;
texture.setRepeated(true);

if (!texture.loadFromFile("grasstex.jpg"))
{
    cout << "File not successfully loaded" << endl;
}

circle.setTexture(&texture);
circle.setTextureRect(sf::IntRect(0, 0, 600, 600));
```


SFML WORKS!

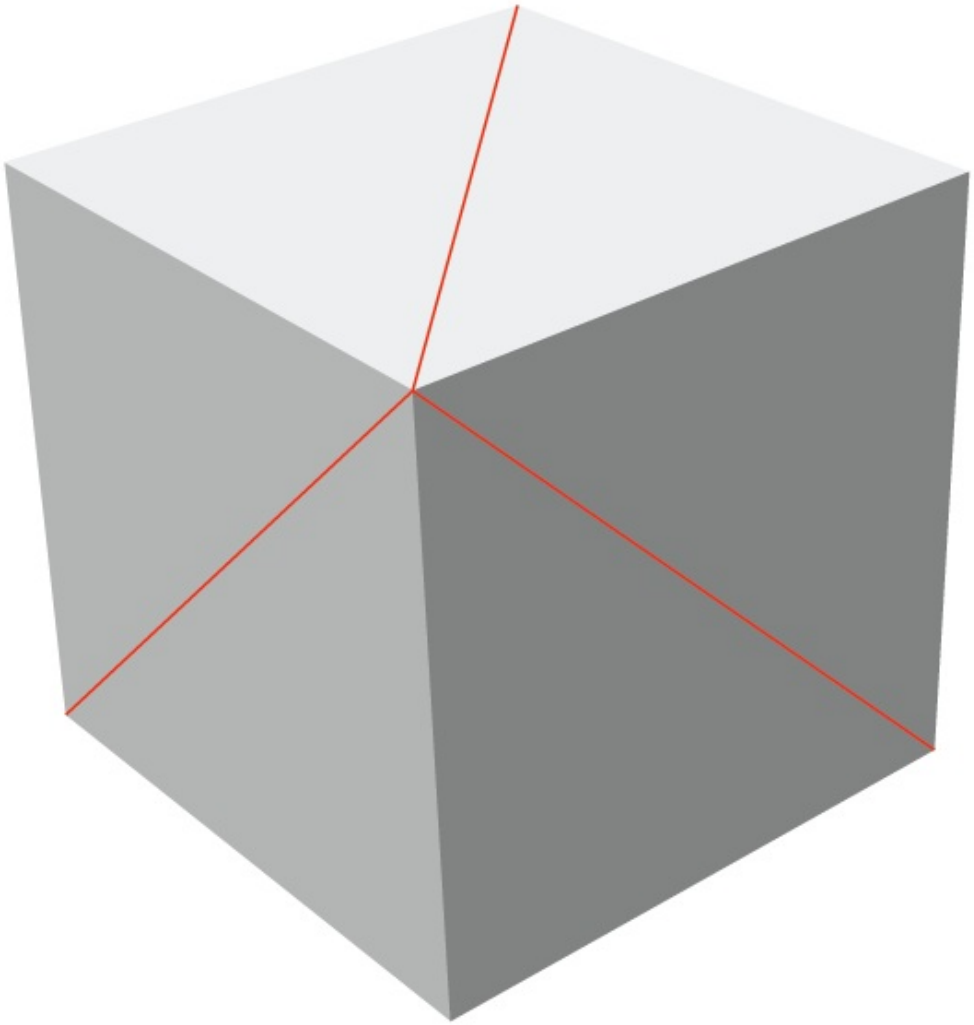
Vertex Arrays

Lines

- All sprites are drawn using triangles. Take for example the following image.



This also applies to 3d dimensional images:



We can store information using vertex arrays. There are 4 types:

1. Point (1 vertex)
2. Line (2 vertices)
3. Triangle (3 verticies)
4. Quad (4 verticies)

Vertex's also have a color attribute, which can be used for gradients.

Starting for scratch:

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

using namespace std;


int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
```

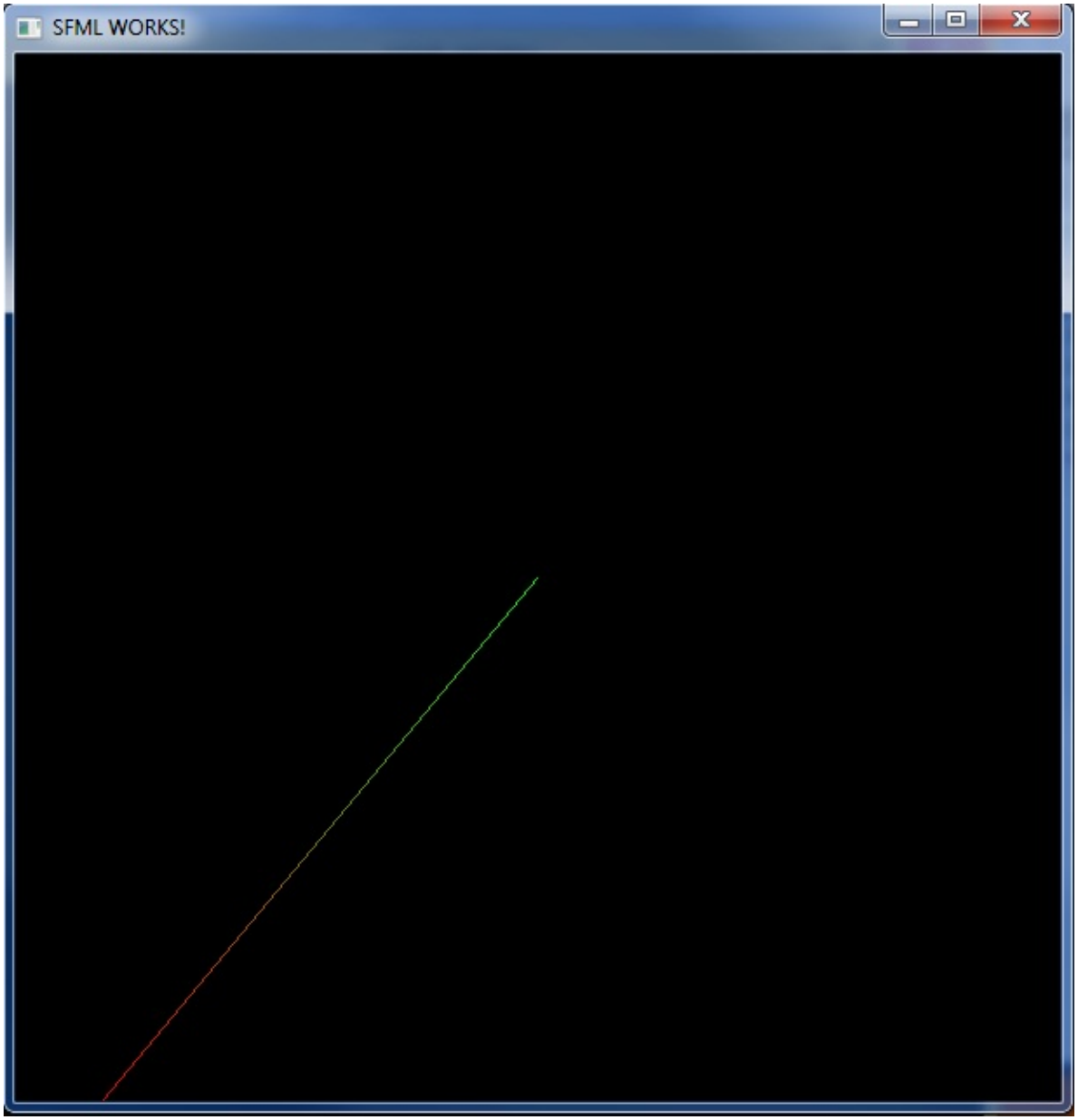
```
            {
            case sf::Event::Closed:
                window.close();

                break;
            }
        }
        window.clear();

        window.draw(circle);

        window.display();
    }
}
```

Earlier, within basic shapes, we've learned to draw lines using a vertex array:

```
    sf::Vertex line2[] =
    {
        sf::Vertex(sf::Vector2f(300,300)),
        sf::Vertex(sf::Vector2f(400, 600)),
    };
```

The method below is no different, just a little less clear. Here we specify an array, and then fill it.

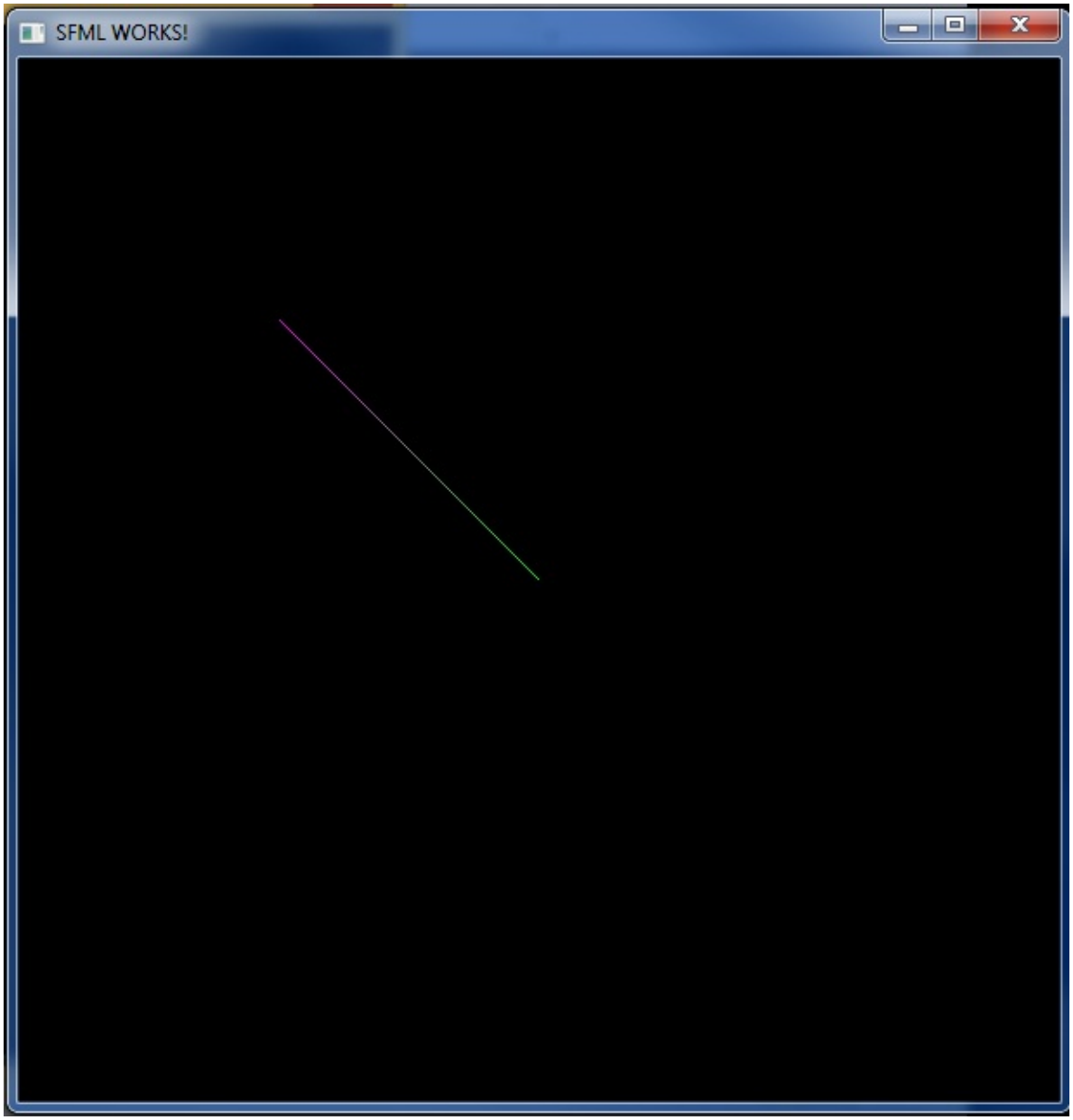
```
    sf::VertexArray line(sf::Lines, 2);

    line[0].position = sf::Vector2f(300, 300);
    line[0].color = sf::Color::Green;
    line[1].position = sf::Vector2f(50, 600);
    line[1].color = sf::Color::Red;

    ...

    window.draw(line);
```



The result here has an interesting effect called color interpolation. The starting point is 100% green, and the ending point is 100% red. Everything in between is the gradual formation of these two colors, where the greater the distance, the lower the percentage.

Notice that we've specified two vertex attributes: Position and color. This attribute was applied with the 2nd vertex array we've mentioned earlier. If these same attributes can be applied with a point as well, couldn't we use a point to specify a line? Certainly.

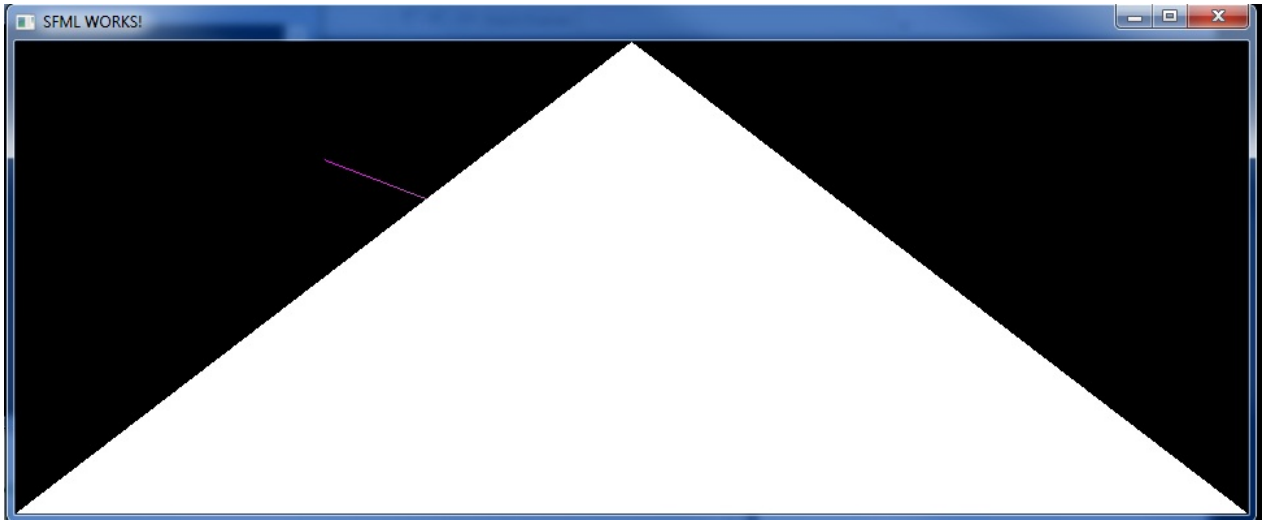
```
sf::Vertex point;
point.position = sf::Vector2f(150, 150);
point.color = sf::Color::Magenta;

//line[1].position = sf::Vector2f(50, 600);
//line[1].color = sf::Color::Red;

line[1] = point;
```



Triangles

Using similiar syntax, we can represent our third vertex attribute, triangles.

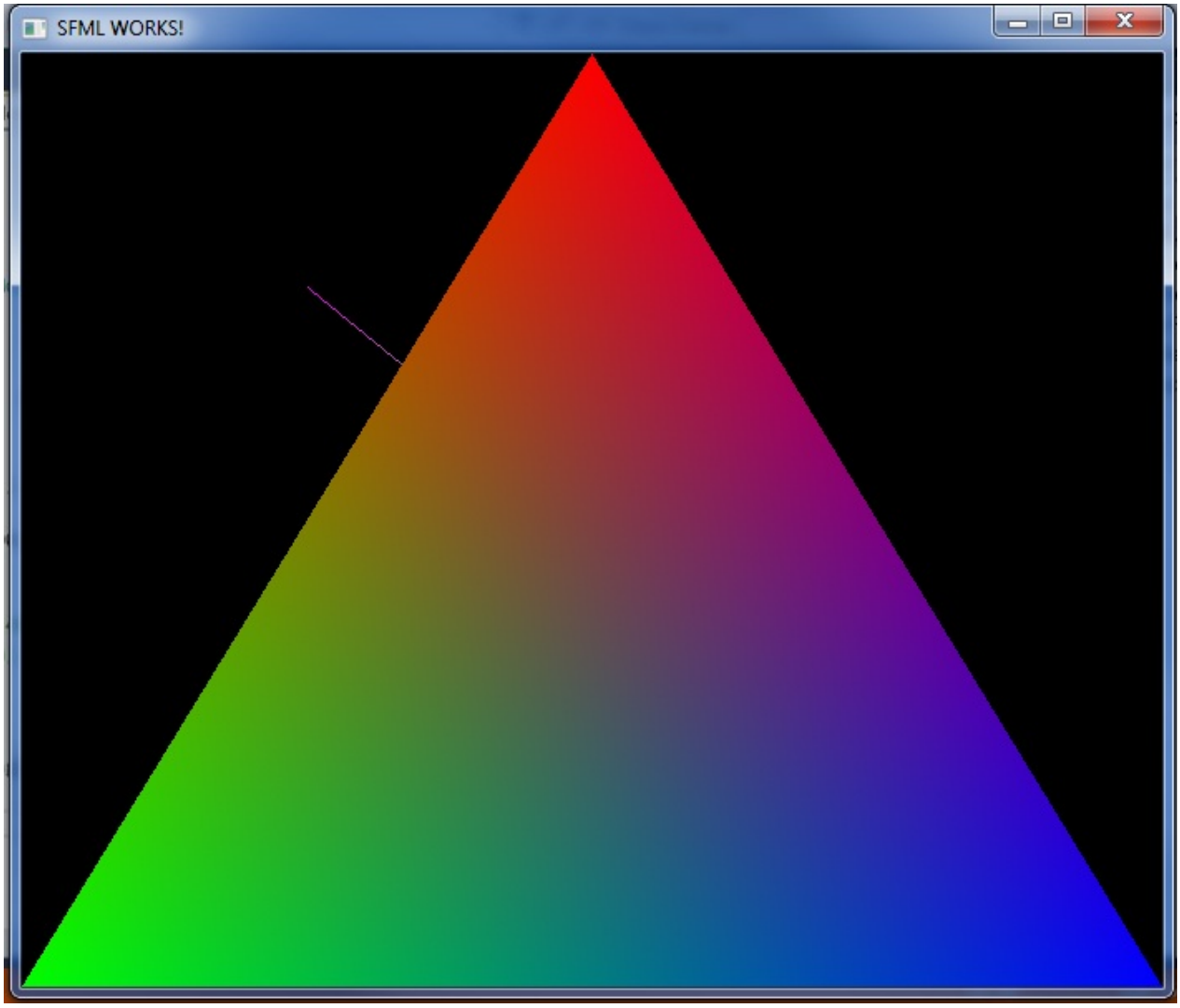
```
sf::VertexArray triangle(sf::Triangles, 3);
triangle[0].position = sf::Vector2f(300, 0);
triangle[1].position = sf::Vector2f(0, 600);
triangle[2].position = sf::Vector2f(600, 600);

...

window.draw(triangle);
```



Three things here. First notice the default color. The color attribute was not specified. Second, I have resized that window and the properties have still remained proportional to the resized property of the window. Third, the triangle has overlapped the line that was drawn earlier. We will get into this later.

```
    triangle[0].color = sf::Color::Red;
    triangle[1].color = sf::Color::Green;
    triangle[2].color = sf::Color::Blue;
```

Check out this beauty

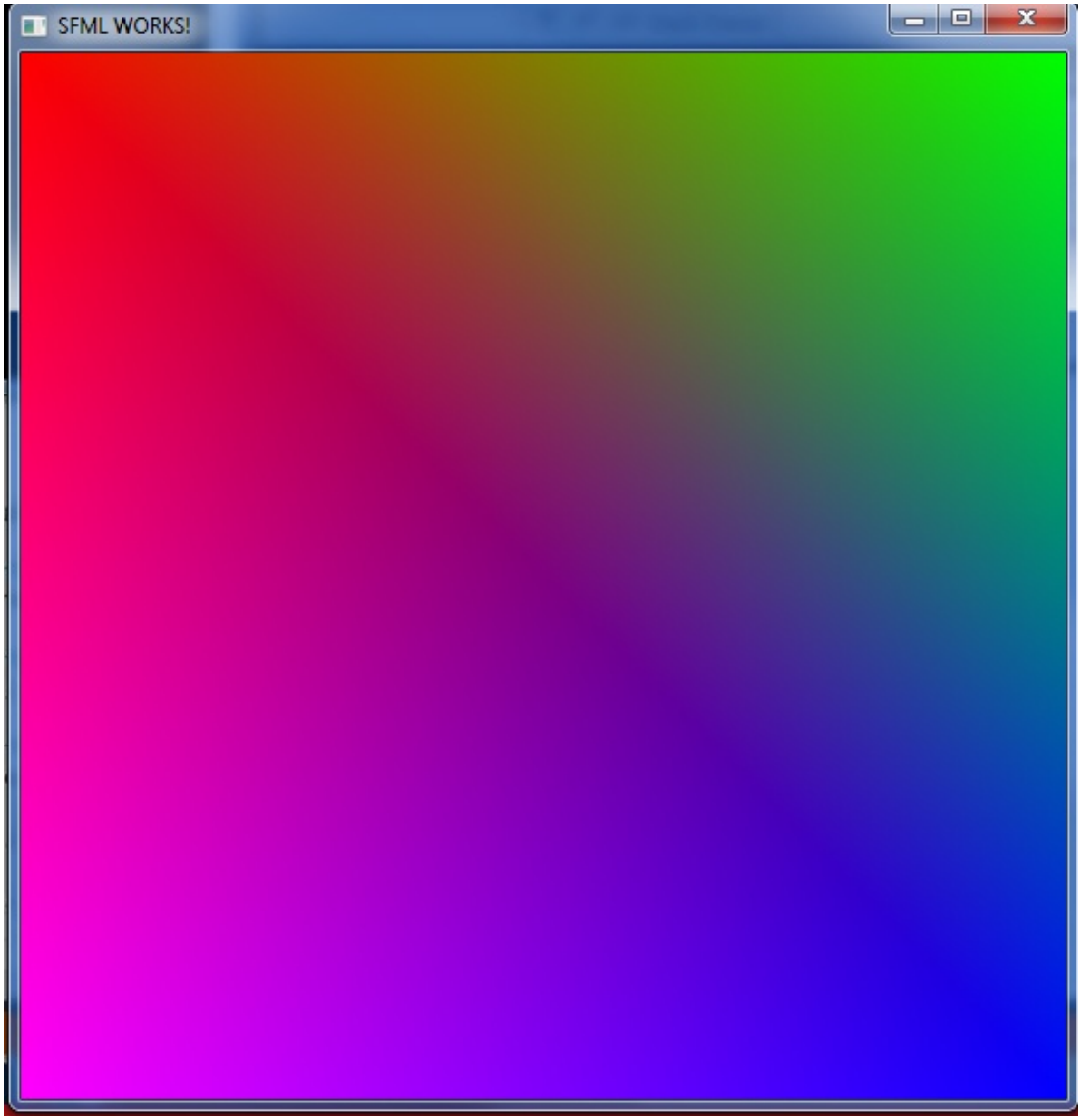


Here we can notice the full effects of color interpolation, because in total, I've specified the vertices to have red, green, and blue respectfully. We can notice in the center, the null state, grey, where red, green, and blue are turned on with equal intensities.

Quads

```
sf::VertexArray quad(sf::Quads, 4);
quad[0].position = sf::Vector2f(0, 0);
quad[1].position = sf::Vector2f(600, 0);
quad[2].position = sf::Vector2f(600, 600);
quad[3].position = sf::Vector2f(0, 600);

quad[0].color = sf::Color::Red;
quad[1].color = sf::Color::Green;
quad[2].color = sf::Color::Blue;
quad[3].color = sf::Color::Magenta;
```

One thing to note here is that the ordering of the position vector arrays matter. Think of drawing the quad without lifting your pencil from the paper. An otherwise non-chronological order would disorient your final image.

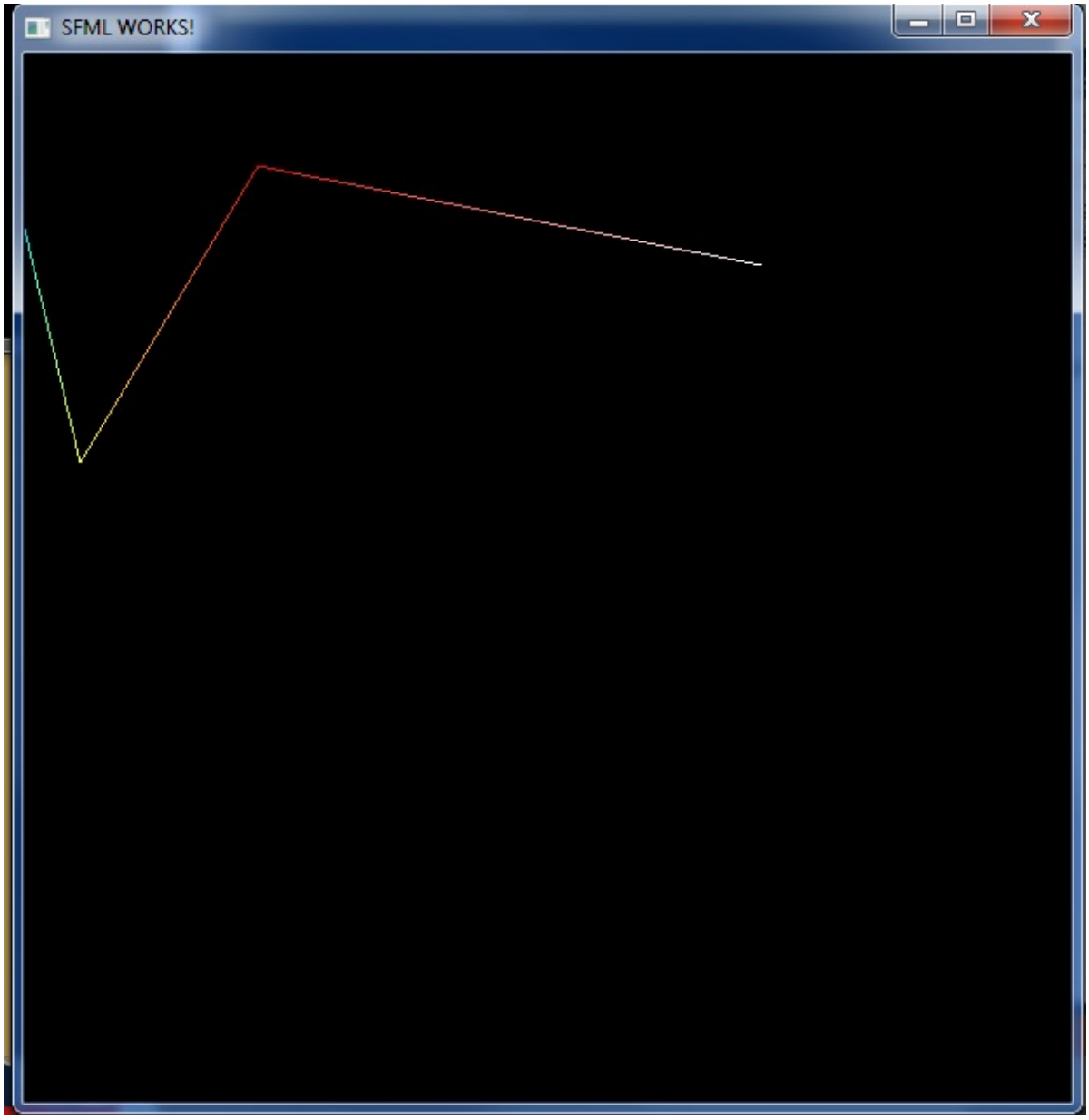


We've also notice that our method in implementing this was quite inefficient, disorganized and redundent. In the next part, we will focus on total that can help with more complicated objects.

Line Strips

Line strips is a helpful tool with drawing multiple, connected line segments.

```
sf::VertexArray lines(sf::LinesStrip, 4);

lines[0].position = sf::Vector2f(0, 100);
lines[1].position = sf::Vector2f(32, 234);
lines[2].position = sf::Vector2f(134, 64);
lines[3].position = sf::Vector2f(423, 121);

lines[0].color = sf::Color::Cyan;
lines[1].color = sf::Color::Yellow;
lines[2].color = sf::Color::Red;
lines[3].color = sf::Color::White;

...

window.draw(lines);
```

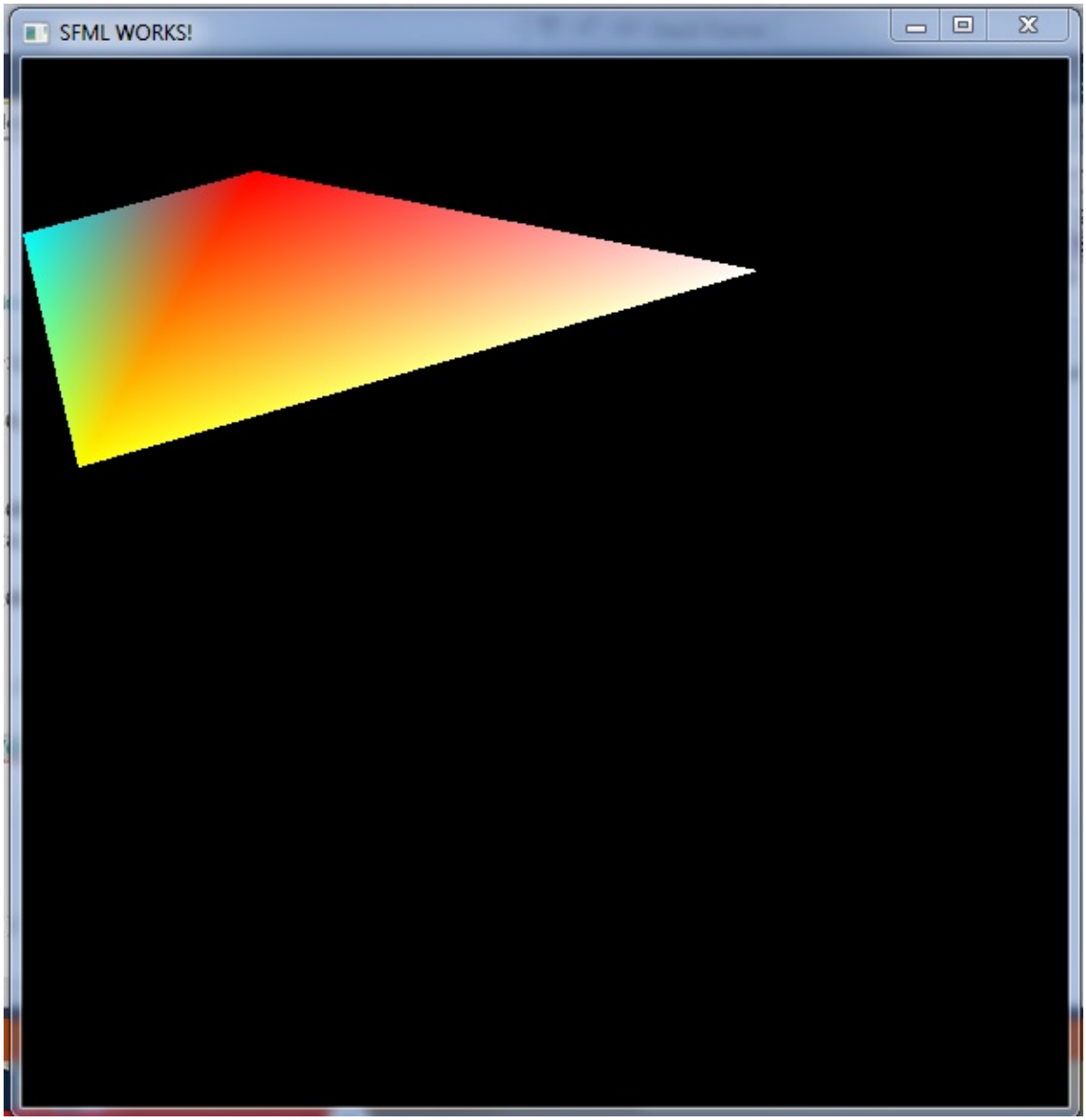


4 Vertex point were specified, each of which were provided and position and color attribute.

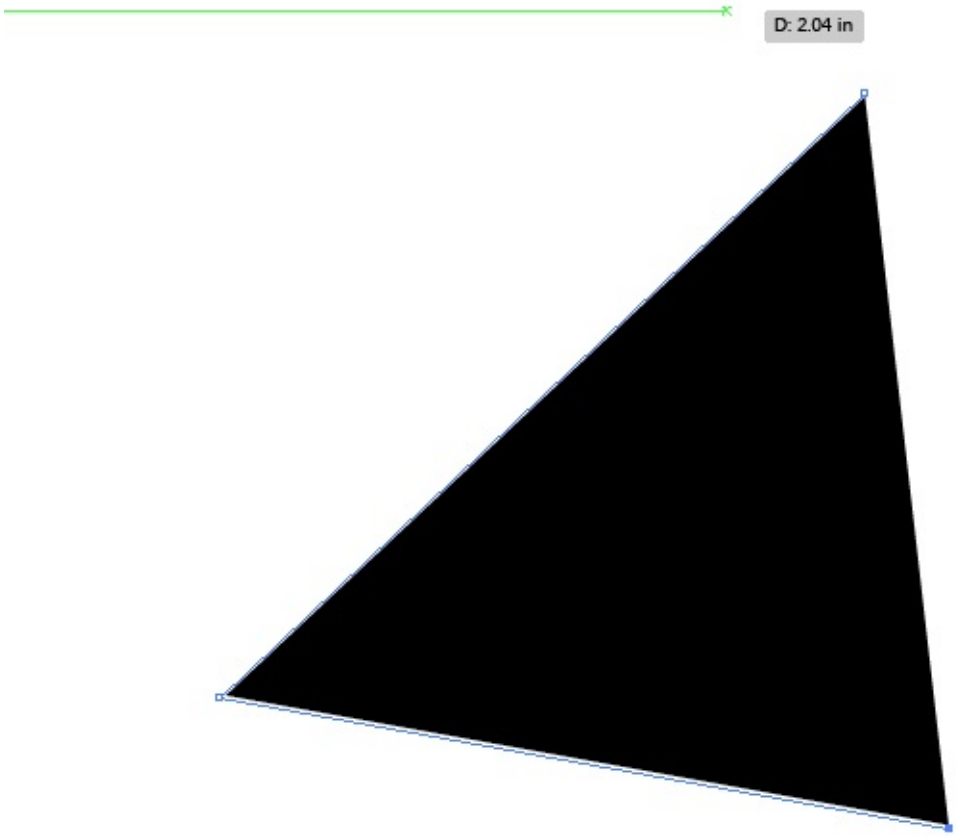
Triangle Strips

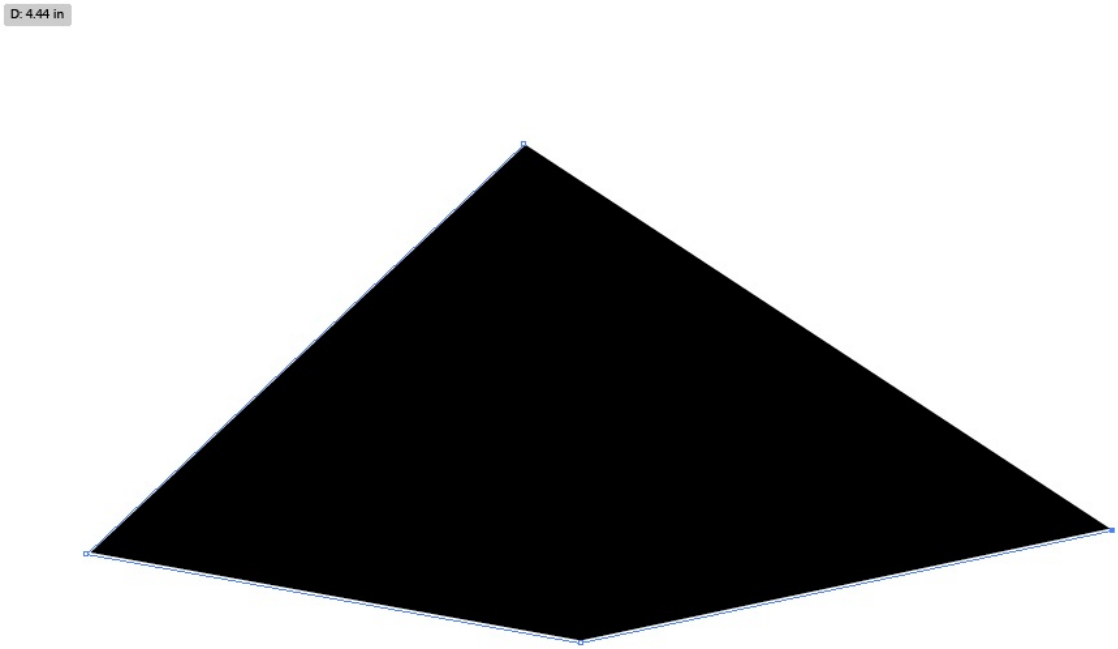
Using the previous image (the same positions are colors are specified), we can determine how the strip feature works.

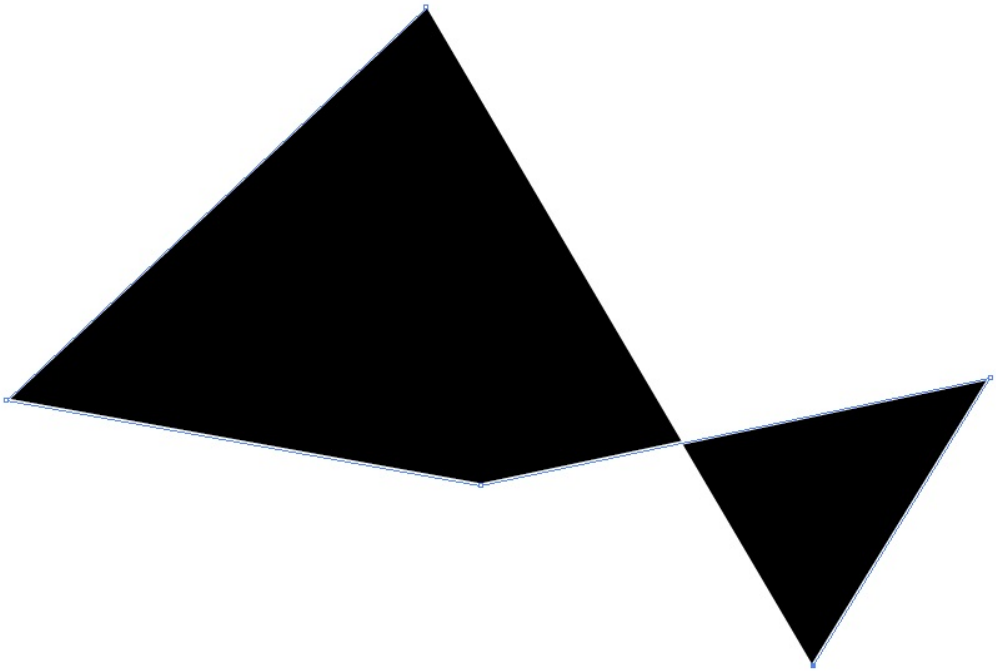
```
sf::VertexArray triangleStrip(sf::TrianglesStrip, 4);

triangleStrip[0].position = sf::Vector2f(0, 100);
triangleStrip[1].position = sf::Vector2f(32, 234);
triangleStrip[2].position = sf::Vector2f(134, 64);
triangleStrip[3].position = sf::Vector2f(423, 121);

triangleStrip[0].color = sf::Color::Cyan;
triangleStrip[1].color = sf::Color::Yellow;
triangleStrip[2].color = sf::Color::Red;
triangleStrip[3].color = sf::Color::White;

...

//window.draw(lines);
window.draw(triangleStrip);
```



The triangleStrip is the nearly the same tool as the 'pen tool' featured with adobe products:


D: 2.04 in


D: 4.44 in

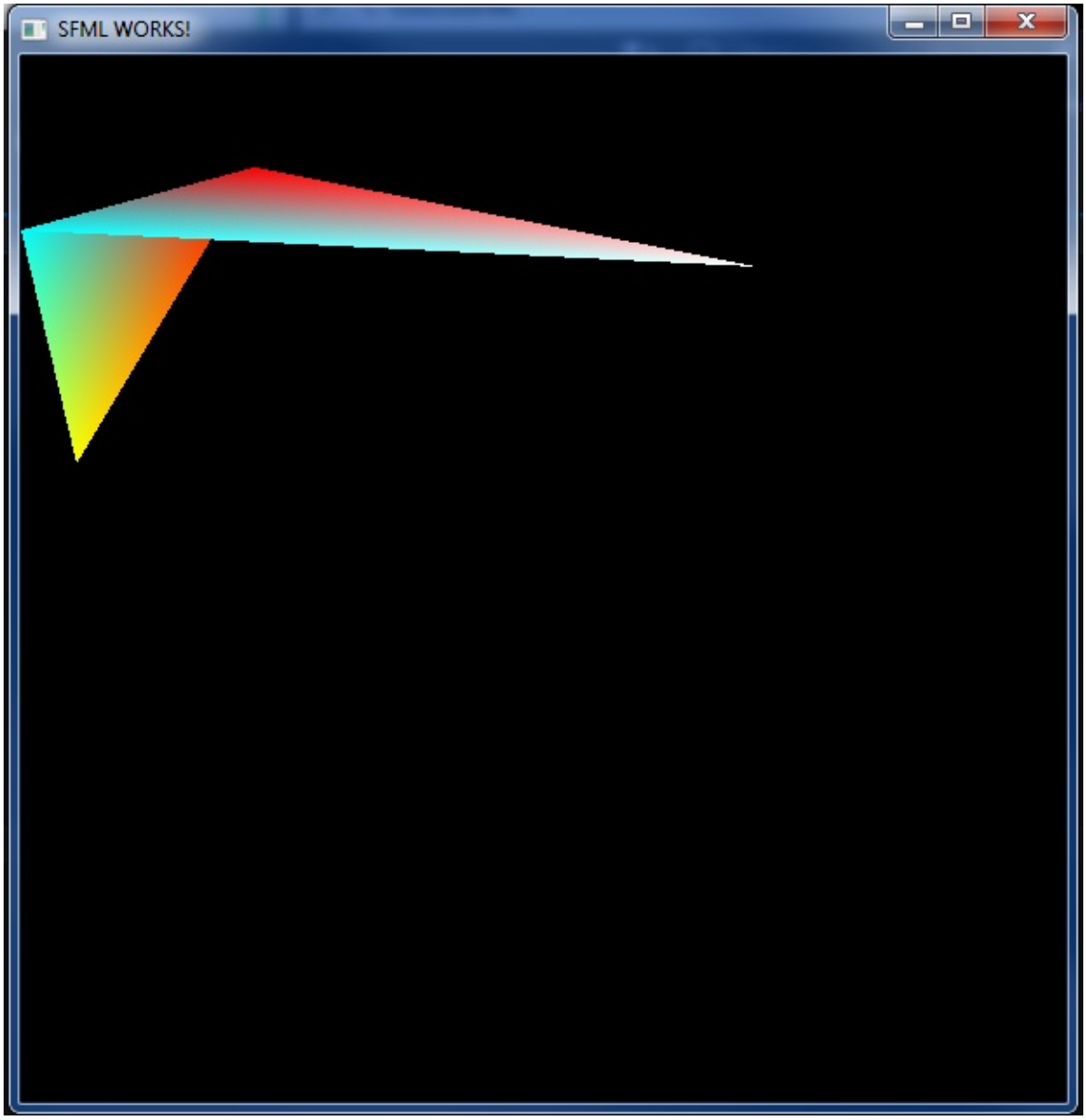


The difference is that fills are next subtracted, but are added from each vertex point to the next.

Triangle Fans

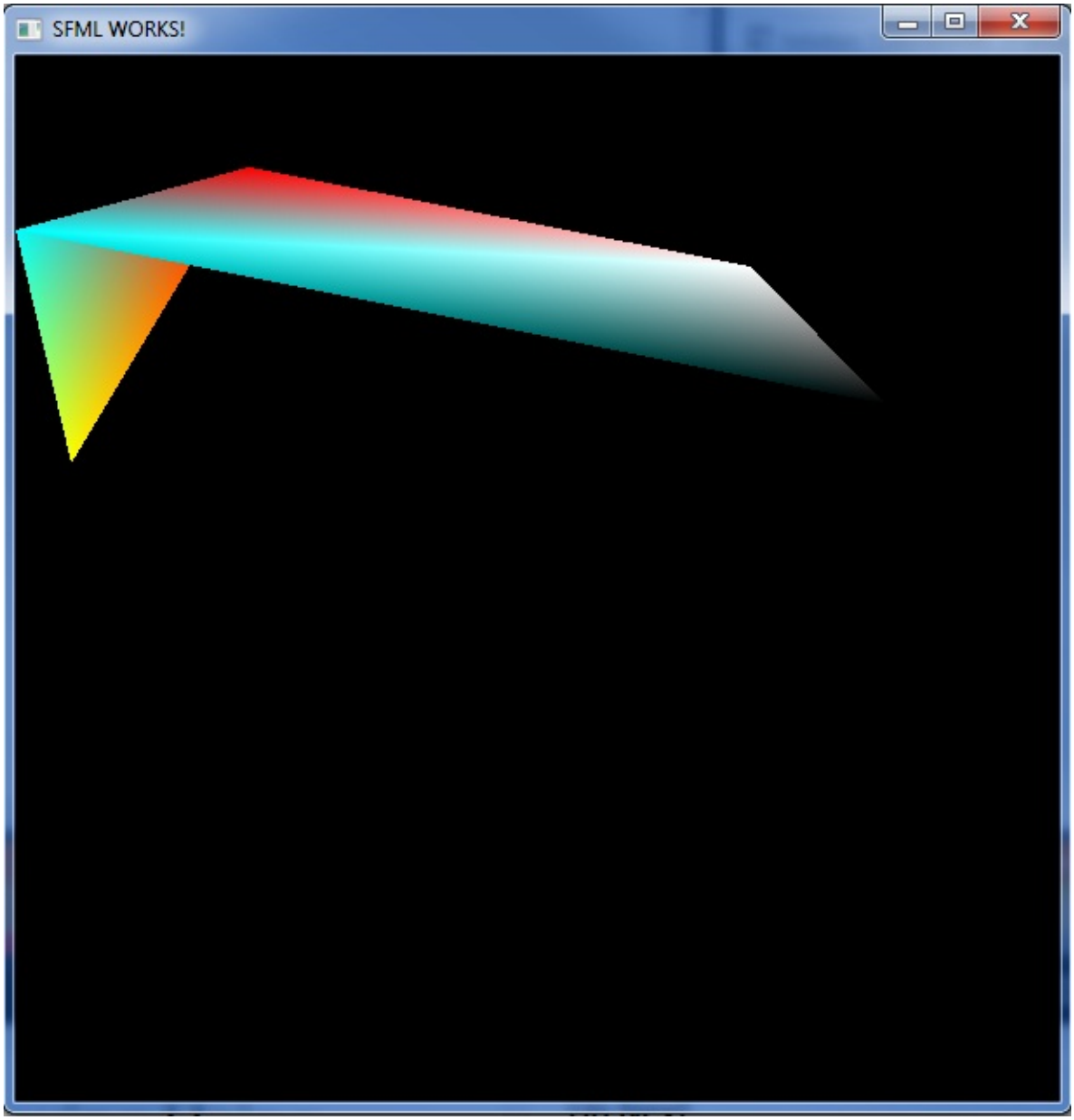
With triangle fans, all points will originate from the starting vertex, in this case, (0,100).

```
sf::VertexArray triangleFan(sf::TrianglesFan);

triangleFan[0].position = sf::Vector2f(0, 100);
triangleFan[1].position = sf::Vector2f(32, 234);
triangleFan[2].position = sf::Vector2f(134, 64);
triangleFan[3].position = sf::Vector2f(423, 121);

triangleFan[0].color = sf::Color::Cyan;
triangleFan[1].color = sf::Color::Yellow;
triangleFan[2].color = sf::Color::Red;
triangleFan[3].color = sf::Color::White;

...

window.draw(triangleFan);
```



Every additional vertex will 'fan out' from the origin point. To demonstrate this, lets add an additional vertex.

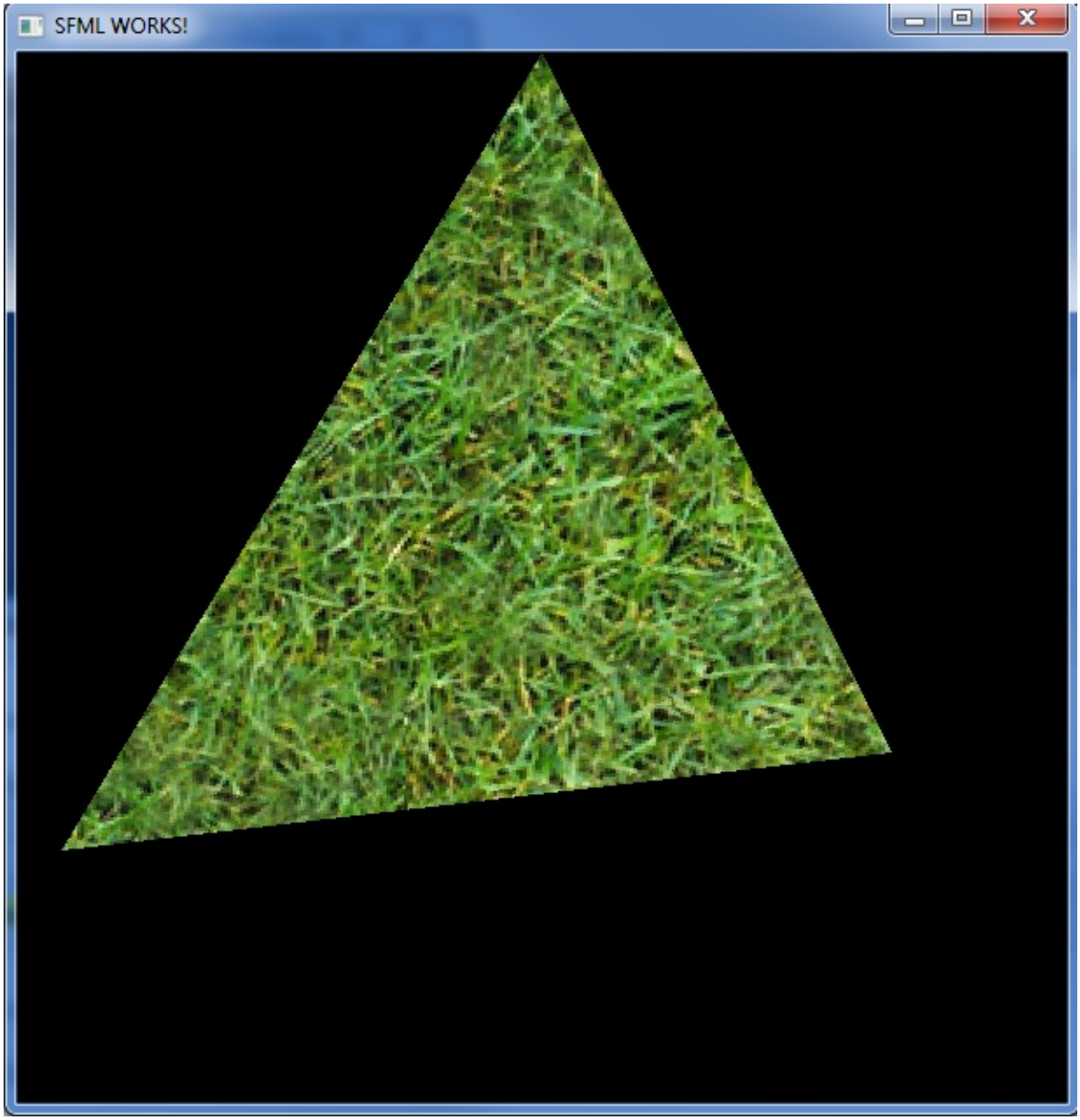
```
sf::VertexArray triangleFan(sf::TrianglesFan, 5);
sf::VertexArray triangleFan(sf::TrianglesFan, 5);
triangleFan[4].color = sf::Color::Black;
```



Vertex Arrays and Textures

Just like SFML provides texture functionality with basic shapes, it supports the same feature with vertex arrays.

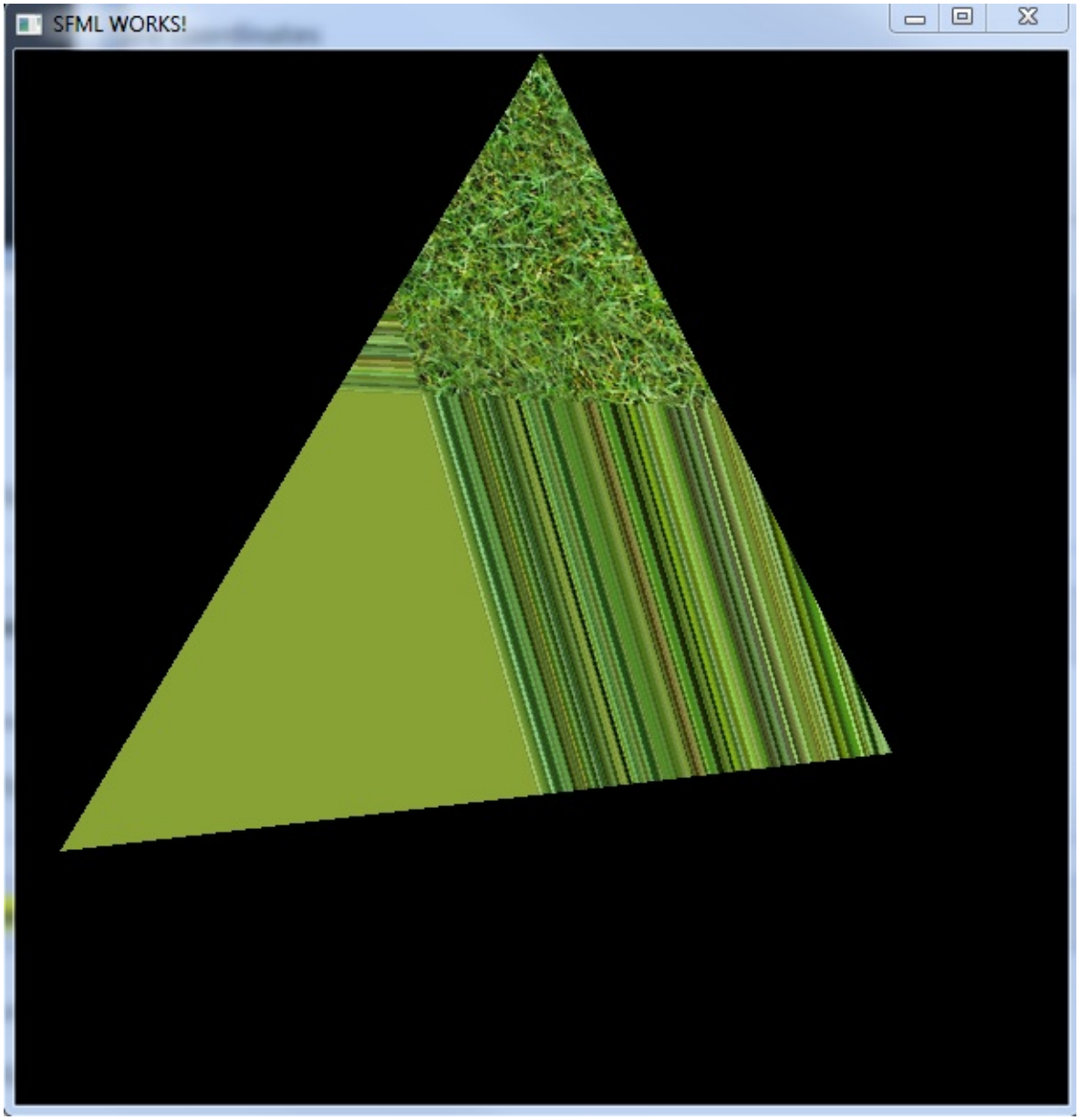
```
sf::VertexArray triangle(sf::TrianglesFan, 3);

triangle[0].position = sf::Vector2f(300, 0);
triangle[1].position = sf::Vector2f(500, 400);
triangle[2].position = sf::Vector2f(25, 456);

sf::Texture texture;

if (!texture.loadFromFile("grasstex.jpg"))
{
    cout << "Error loading texture" << endl;
}

triangle[0].texCoords = sf::Vector2f(100, 0);
triangle[1].texCoords = sf::Vector2f(0, 250);
triangle[2].texCoords = sf::Vector2f(250, 250);

...

windows.draw(triangle, &texture);
```



The first thing to notice is that there is no color attribute. This has been substituted with &texture. Without it, the convex triangle would default to white.

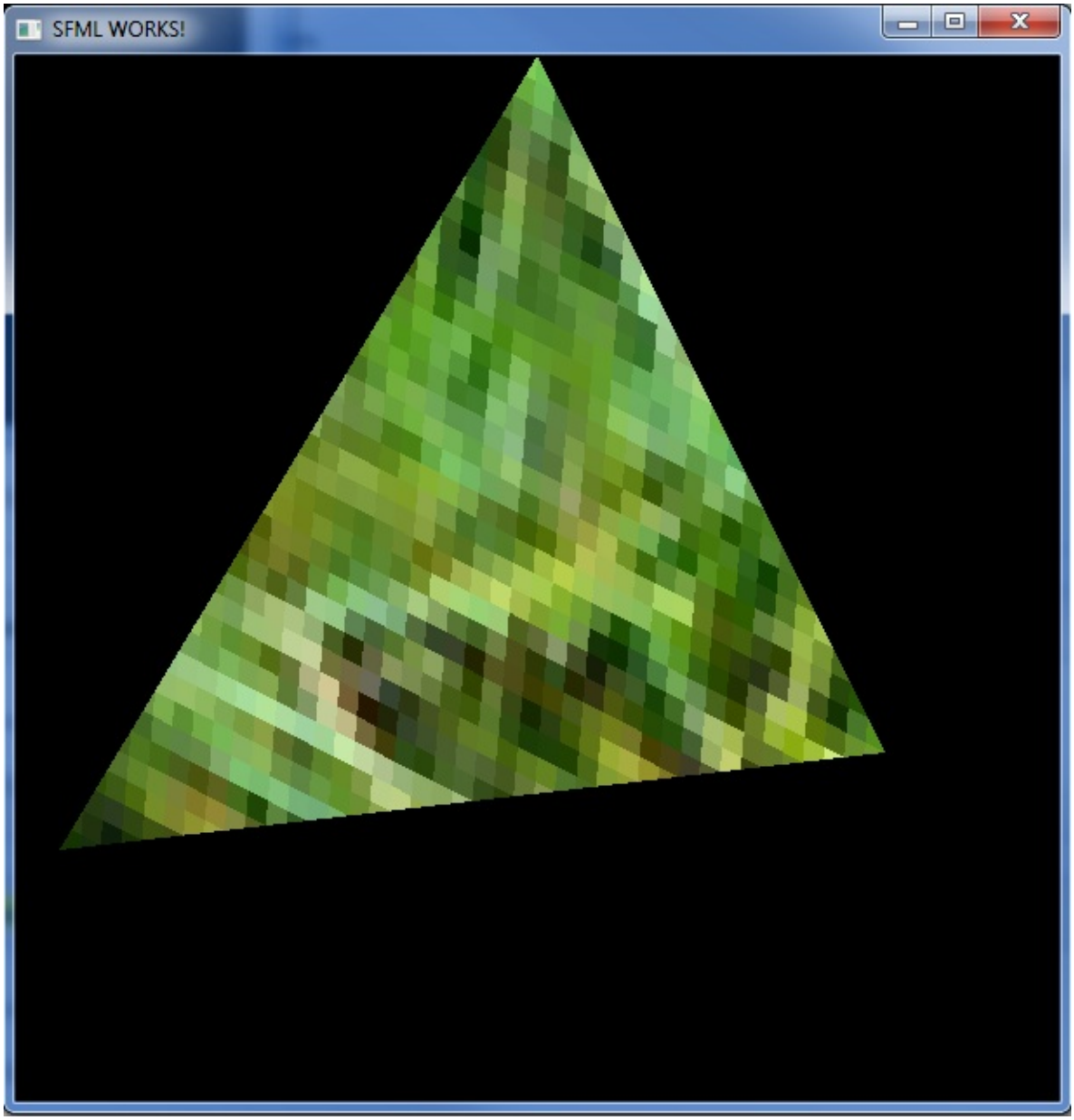
Next is the texCoord. TexCoord specifically applies to the coordinates of our texture. For reference, our texture is 256x256 (widthxheight). What we've done is map out a triangle on our texture, and placed it into the silhouette of our original shape. Because our maximum coordinate was 250, it did not exceed the bounds of texture. So there was no need to resize our image. If we do, we would get an effect experienced before:

```
    triangle[0].texCoords = sf::Vector2f(100, 0);
    triangle[1].texCoords = sf::Vector2f(0, 500);
    triangle[2].texCoords = sf::Vector2f(600, 600);
```




```
    triangle[0].texCoords = sf::Vector2f(20, 0);
    triangle[1].texCoords = sf::Vector2f(0, 20);
    triangle[2].texCoords = sf::Vector2f(40, 40);
```

and an area otherwise smaller than 256x256 would constrain it to the size of the silhouette.

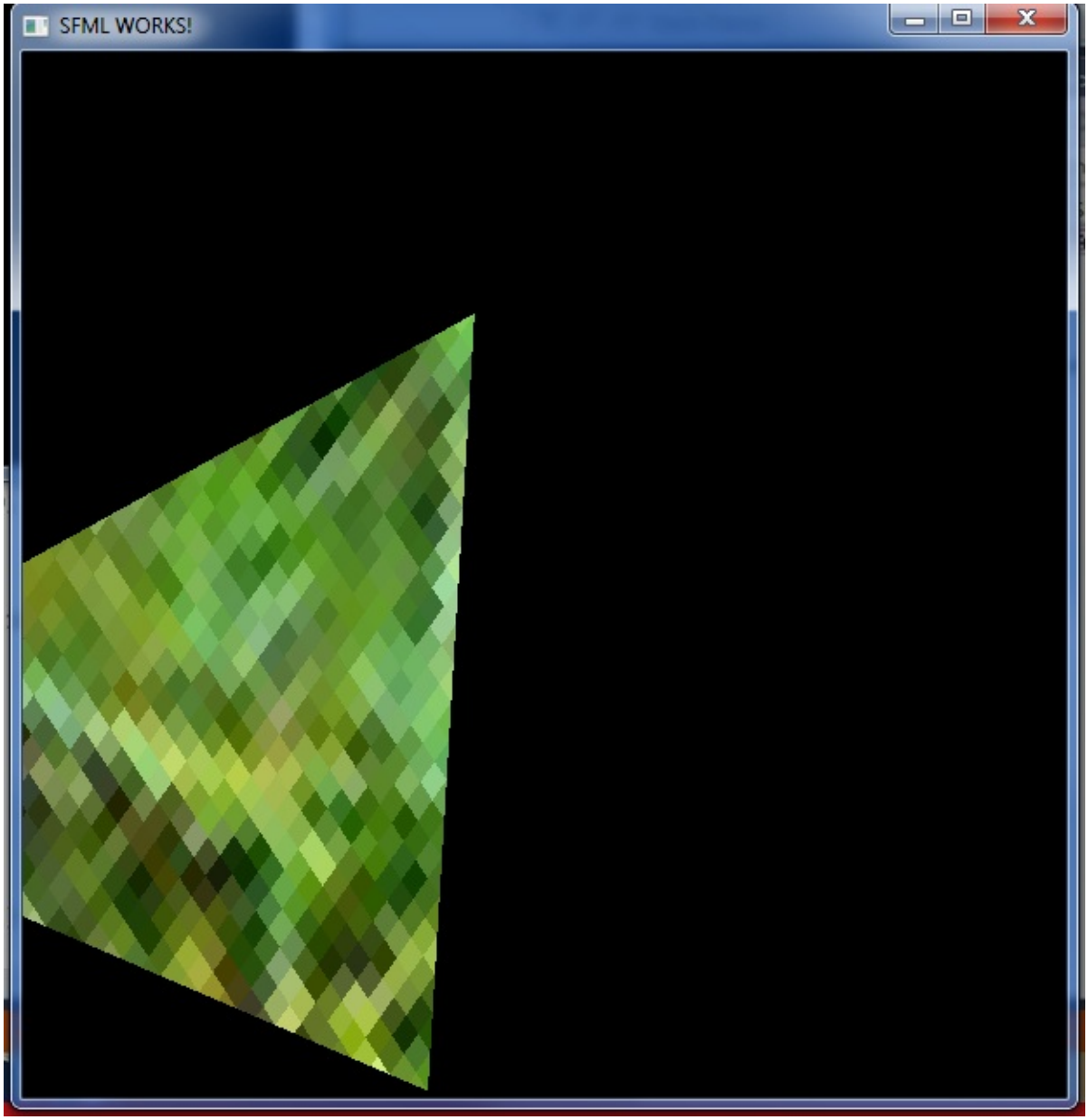


Transforming Vertex Arrays

Say that in addition to textured shapes, we wanted to rotate them.

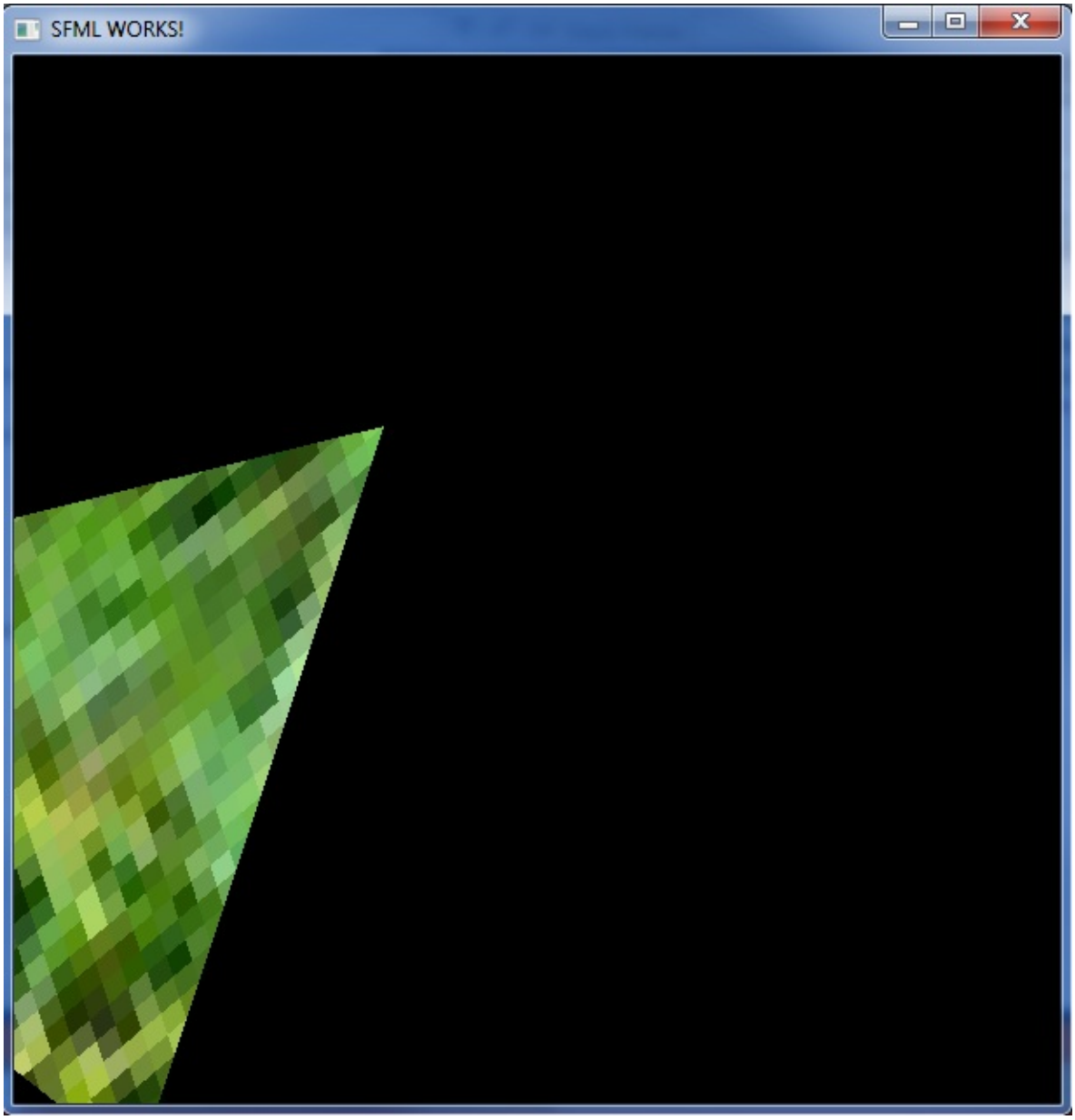
Using the same triangle from above with a texture, would require the class Transform and RenderStates. Without RenderStates, the triangle would transform without the texture.

```
sf::Transform transform;

transform.rotate(30, sf::Vector2f(0, 0));

sf::RenderStates state;
state.transform = transform;
state.texture = &texture;

...

window.draw(triangle, state);
```



Here our triangle is rotated 30 degrees reletive to the origin: 0,0. And addition 15 degrees of rotation would result in the following:

```
    transform.rotate(45, sf::Vector2f(0, 0));
```


SFML WORKS!

Views

Views are what allow you to move around, and provide zoom functionality. Games like Empire Earth, LoL are classic examples.

Black State:

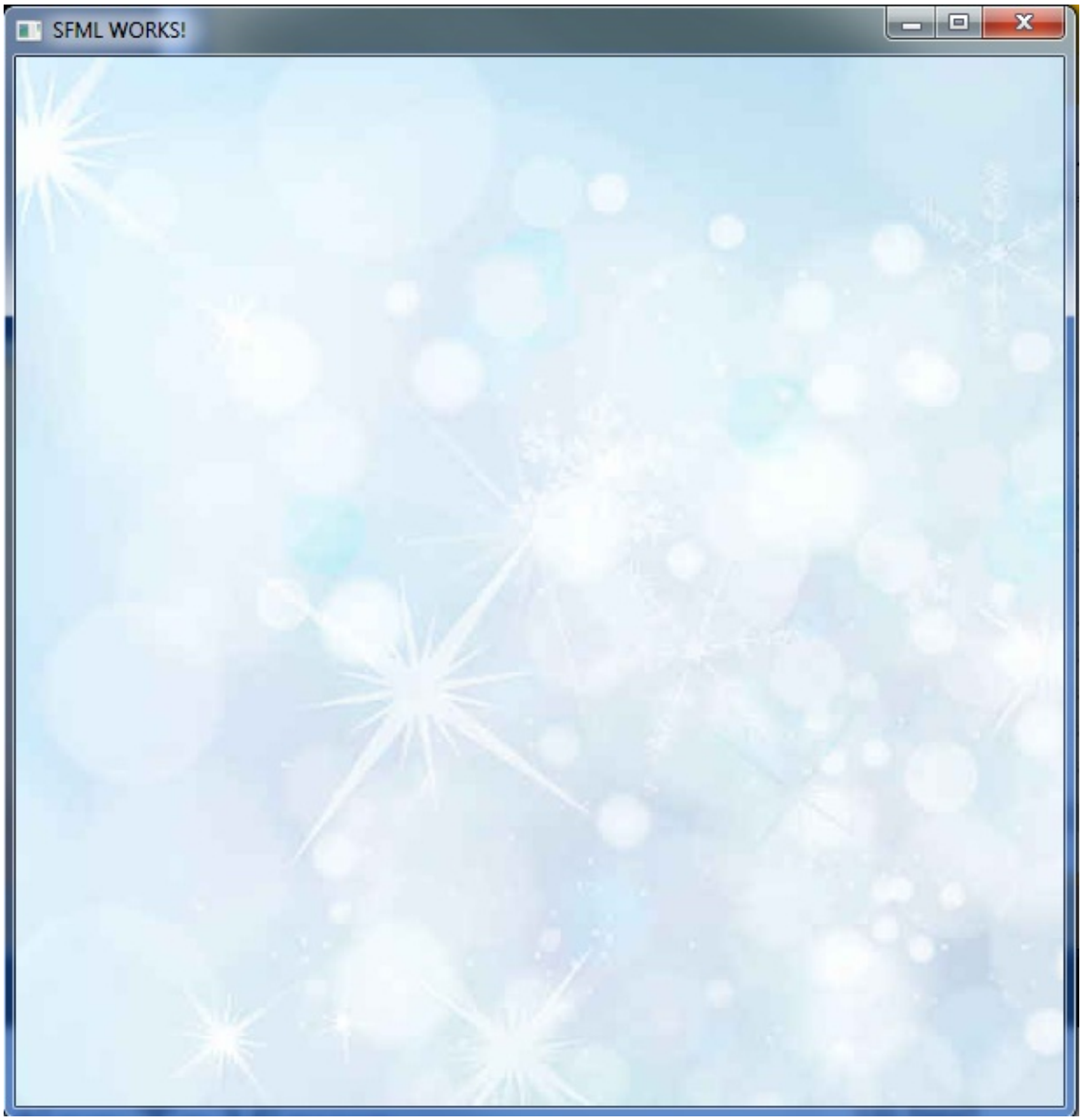
```
#include "SFML/Graphics.hpp"
#include <iostream>

using namespace std;


int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();

                break;
            }
        }
        window.clear();



        window.display();
    }
}
```

We will be combining the applications of textures and sprites. What exactly is the difference?

- Textures are loaded once as a single image in memory. It is just an image.
- Sprites can use textures to be manipulated. E.g. scaled, rotated, positioned, skewed, or colorized just as we have experienced.
- So when we manipulate a texture, we get a sprite.

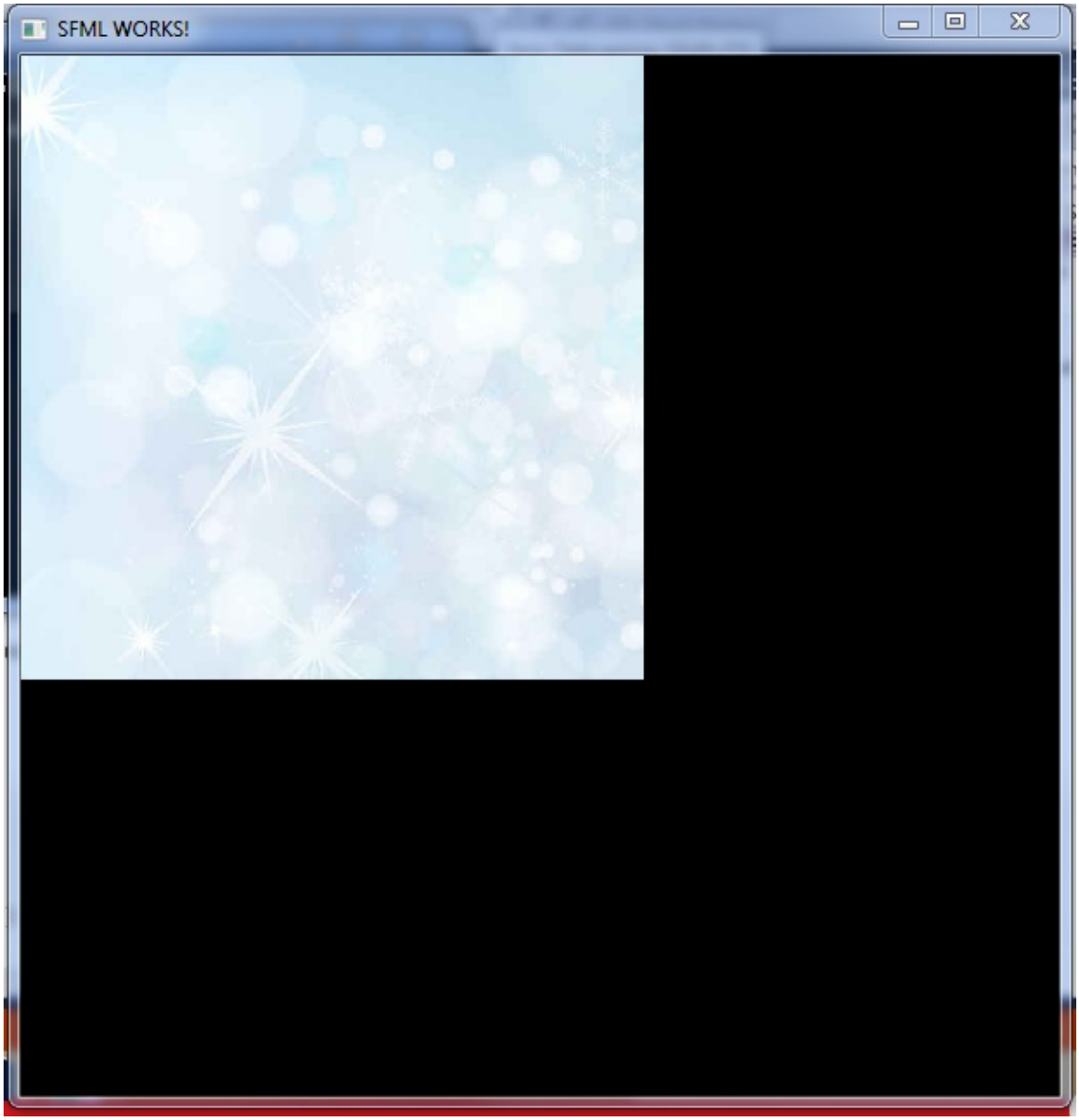

```
    sf::Sprite background;

    sf::Texture texture;
```

```
if (!texture.loadFromFile("background.jpg"))
{
    cout << "Error loading texture" << endl;
}

background.setTexture(texture);
```



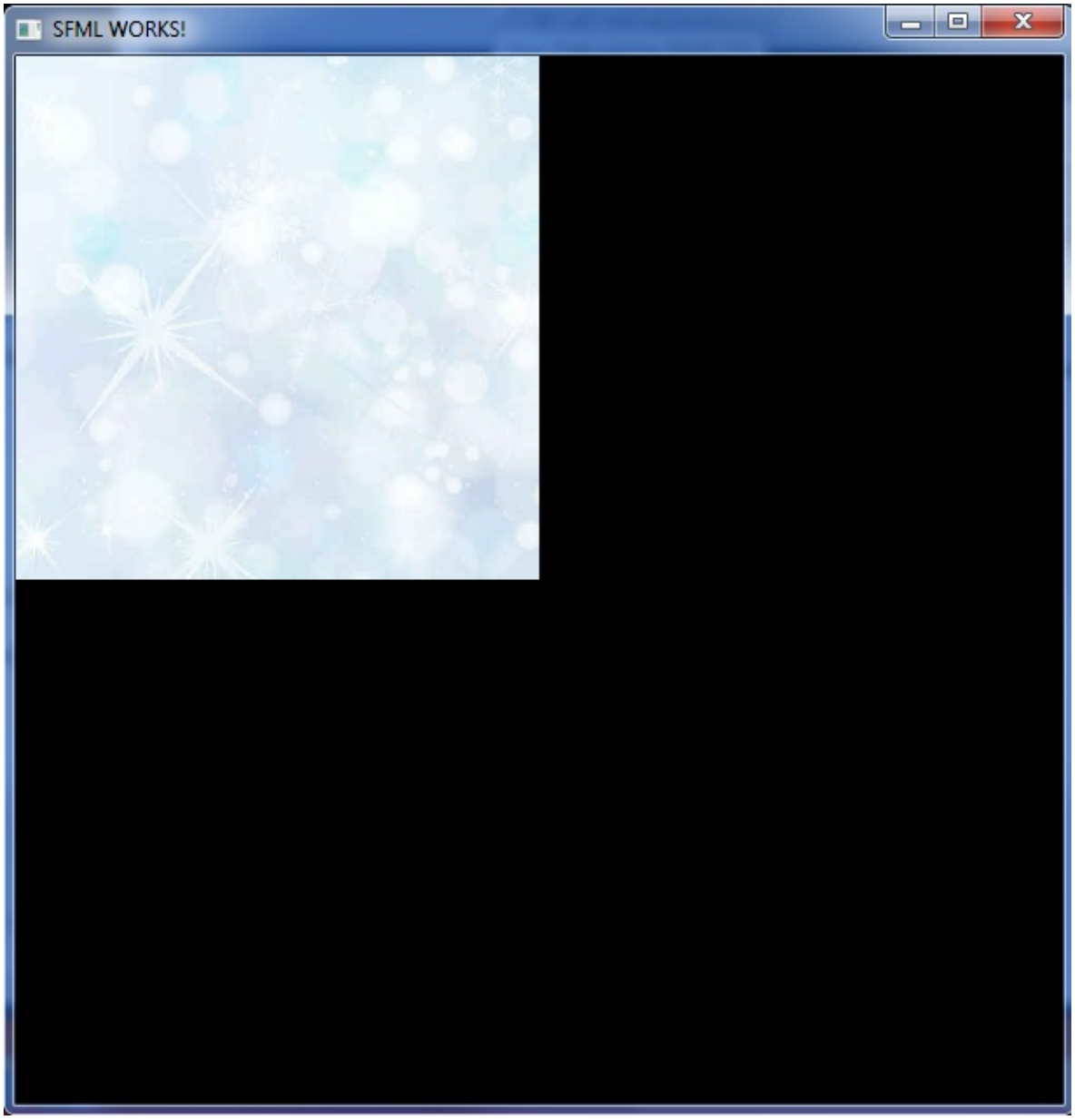
The image was taken from google, and is exactly 600 by 600 - which is why it fits proportionally in our window. Back in the sprite manipulation chapter when we used the function `sprite.setScale(sf::Vector2f(0.5, 4));` we've managed to proportionally scale our image down. This however, *directly* constrains the image itself. It we wanted to change the 2-dimensional camera view, we use `View`, which defines the region where a camera is shown. While the effect is similiar, the difference lies in that a different field of view is given to the user. You can either see less or more of the entirety of your gameplay.

```
sf::View view(sf::FloatRect(0, 0, 1000, 1000));
window.setView(view);
```



Here, we've expanded the view of our window. The scale of our image remains the same, but our window 'zooms out' to 1000x1000. The first 2 parameters specify the starting location of our sprite, as demonstrated here:

```
sf::View view(sf::FloatRect(100,100, 1000, 1000));
```

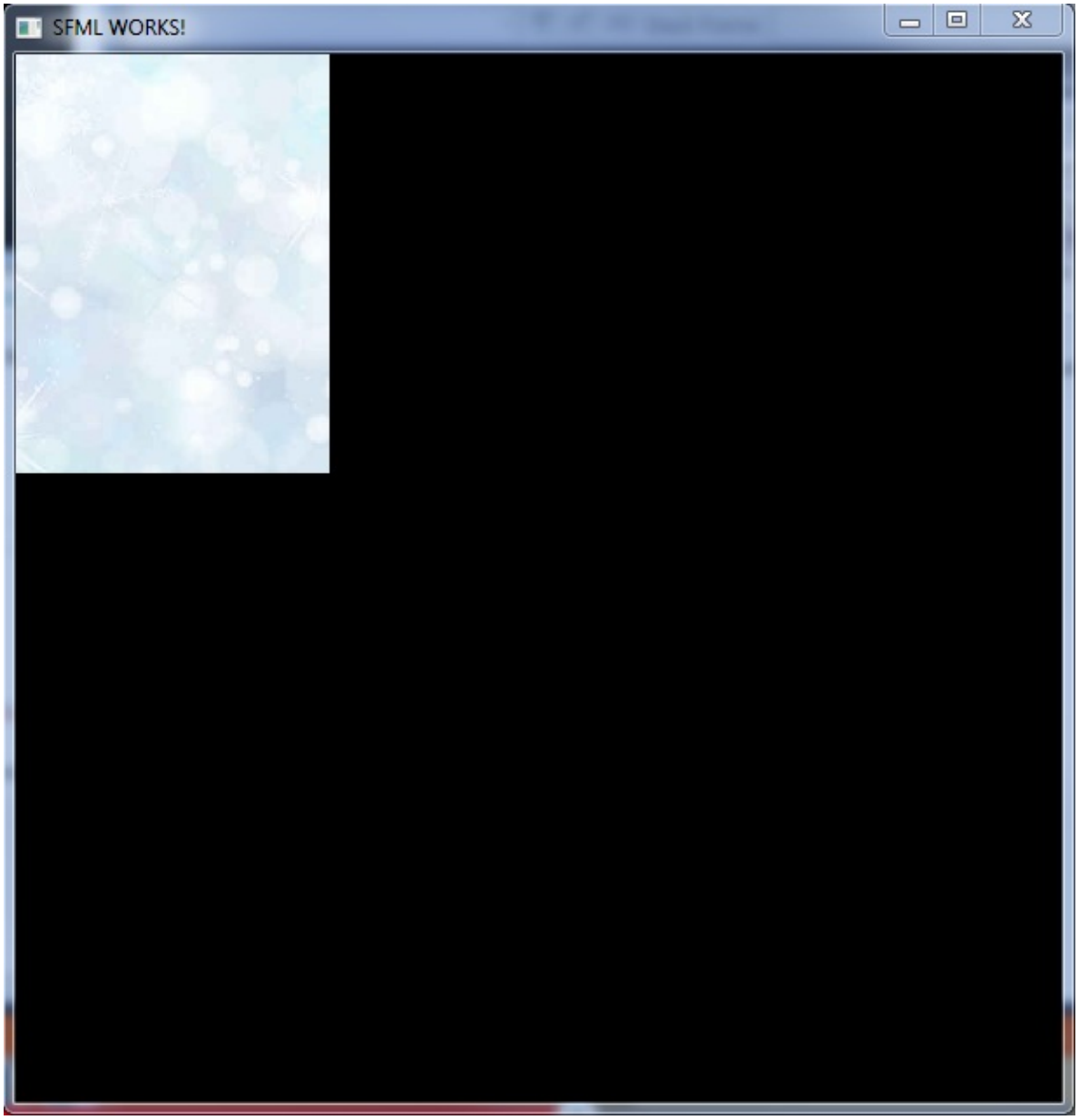


Moving a View

The click and drag for monitoring a camera view can be one application of this. For simplicity, lets move the field of view to a fixed position:

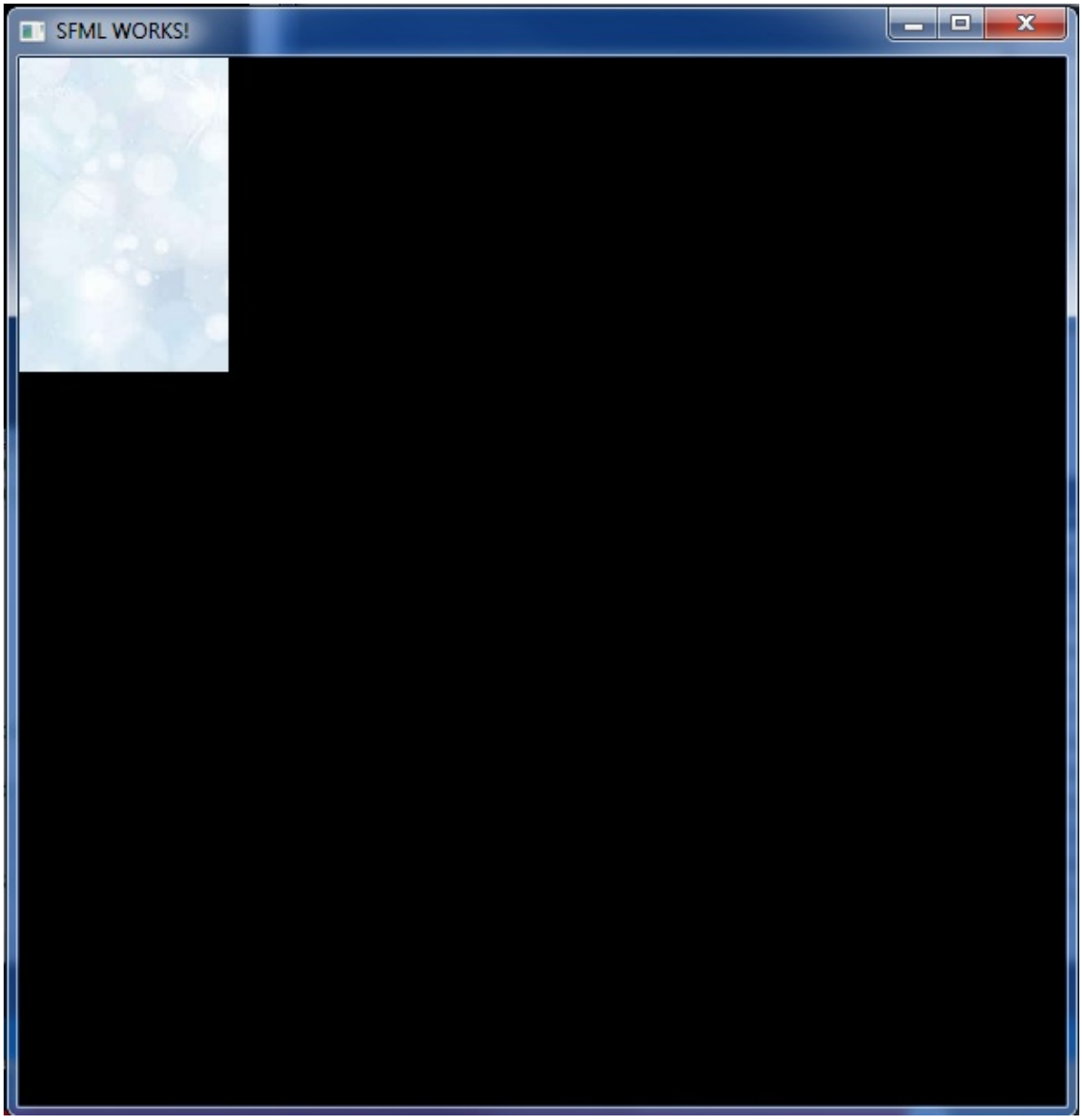
```
    window.setView(view);

    view.move(sf::Vector2f(200, 100));

    window.setView(view);
```

Notice here that window.setView is used twice. This is nessessary because view.move is a local copy, and not a reference. It is a light weight function, so dont be afraid to use it multiple times.



The effect is also stackable:

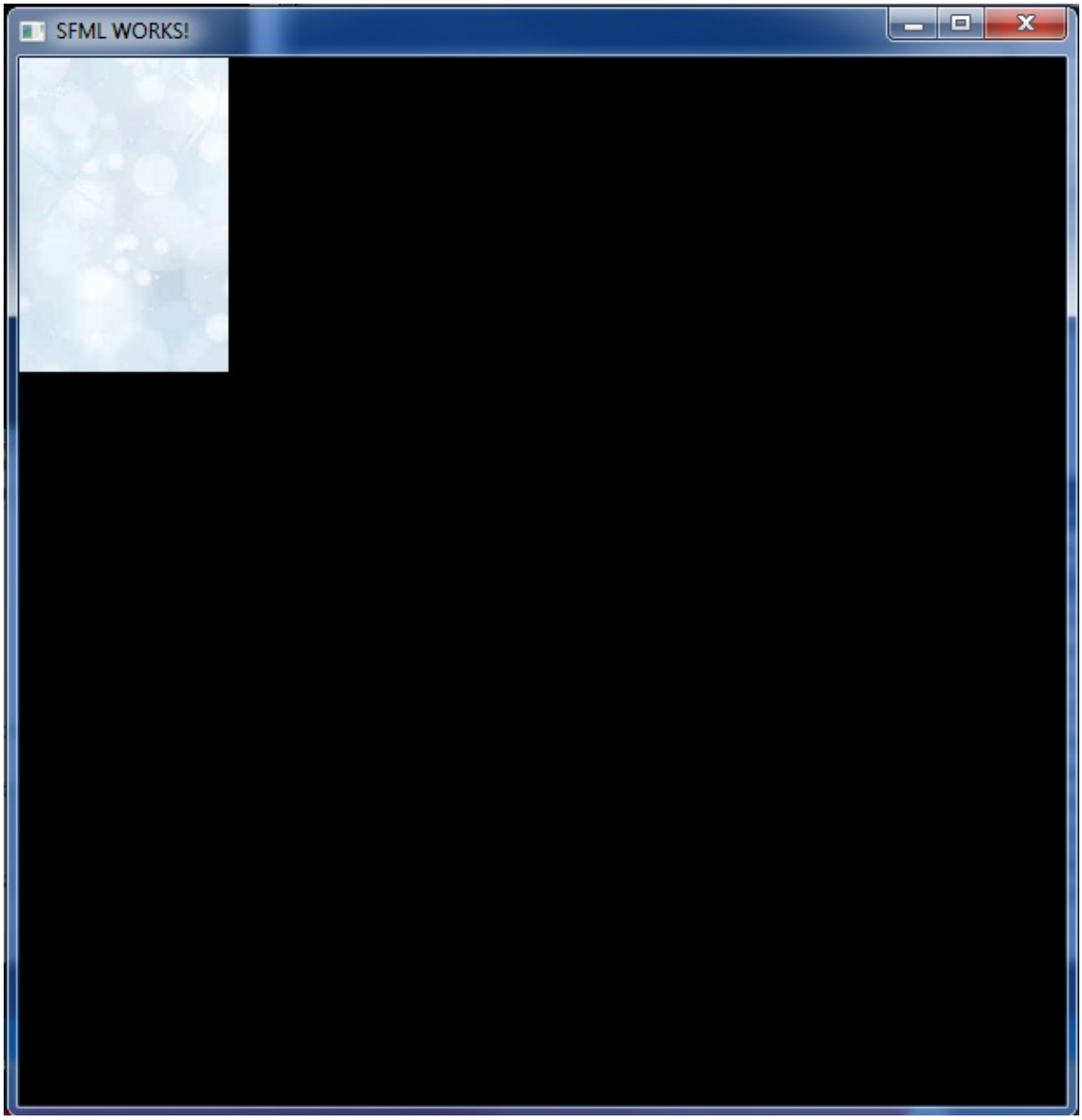
```
    view.move(sf::Vector2f(50, 50));
    view.move(sf::Vector2f(50, 50));
```



Centering a View

```
    view.setCenter(sf::Vector2f(300, 300));
```

Note: SetCenter, will also overwrite all previous moves.



In general, all 'set' functions are non-stackable, and are reletive to to the window size, as demonstrated here:

Rotating a View

```
view.rotate(45);
```

- Is stackable

And as we would expect, its parent function does overwrite and is stackable.

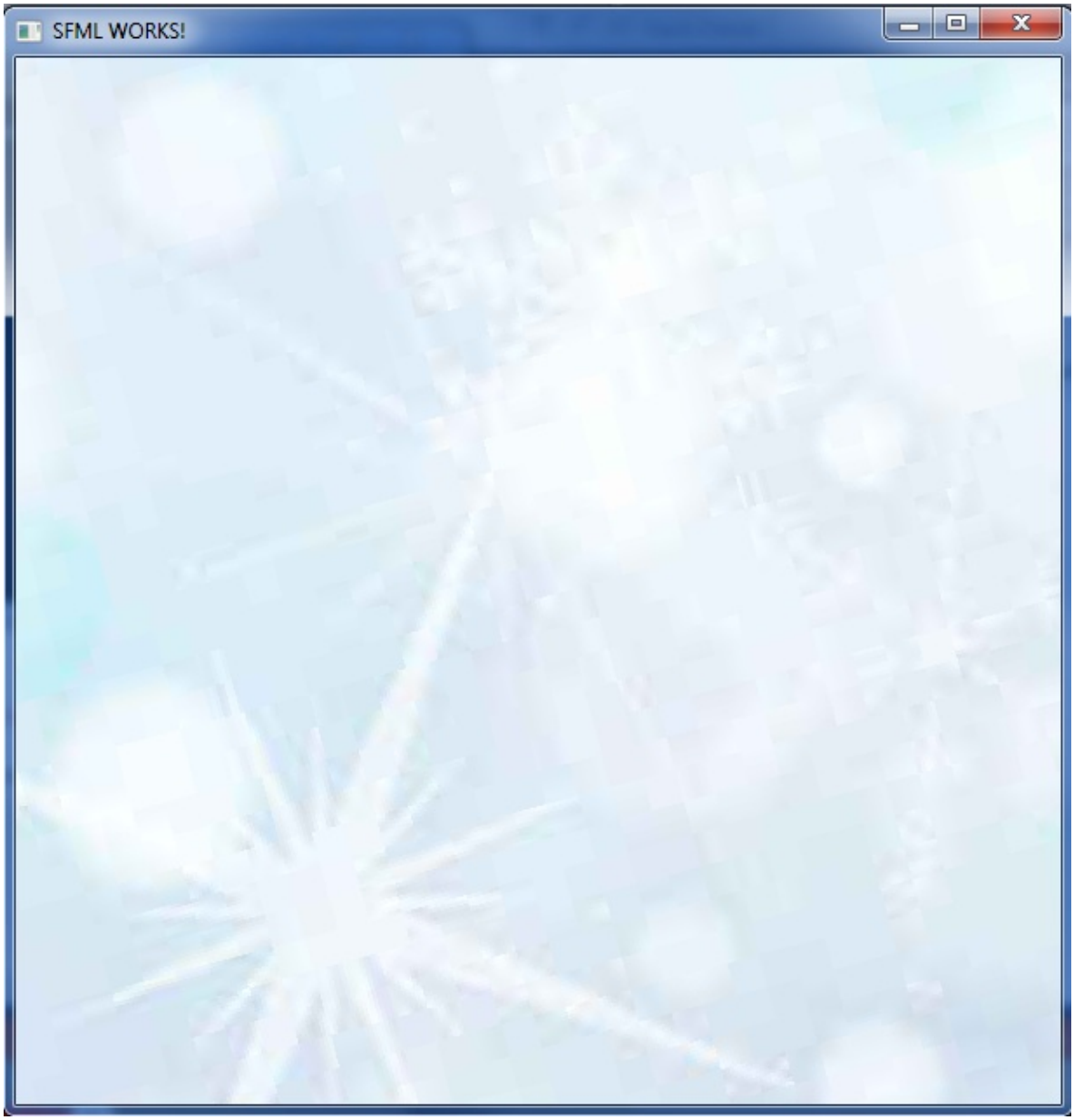
```
view.setRotation(15);
```

Scaling a View

This application can be applied in for example, using a mouse scroll to zoom in, or using the two finger method to zoom in mobile platforms.

Here are the accumulative changes we've tracked so far:

```
    window.setView(view);

    view.setCenter(sf::Vector2f(300, 300));
    view.rotate(45);
    view.setRotation(15);
    view.zoom(0.25f);

    window.setView(view);
```



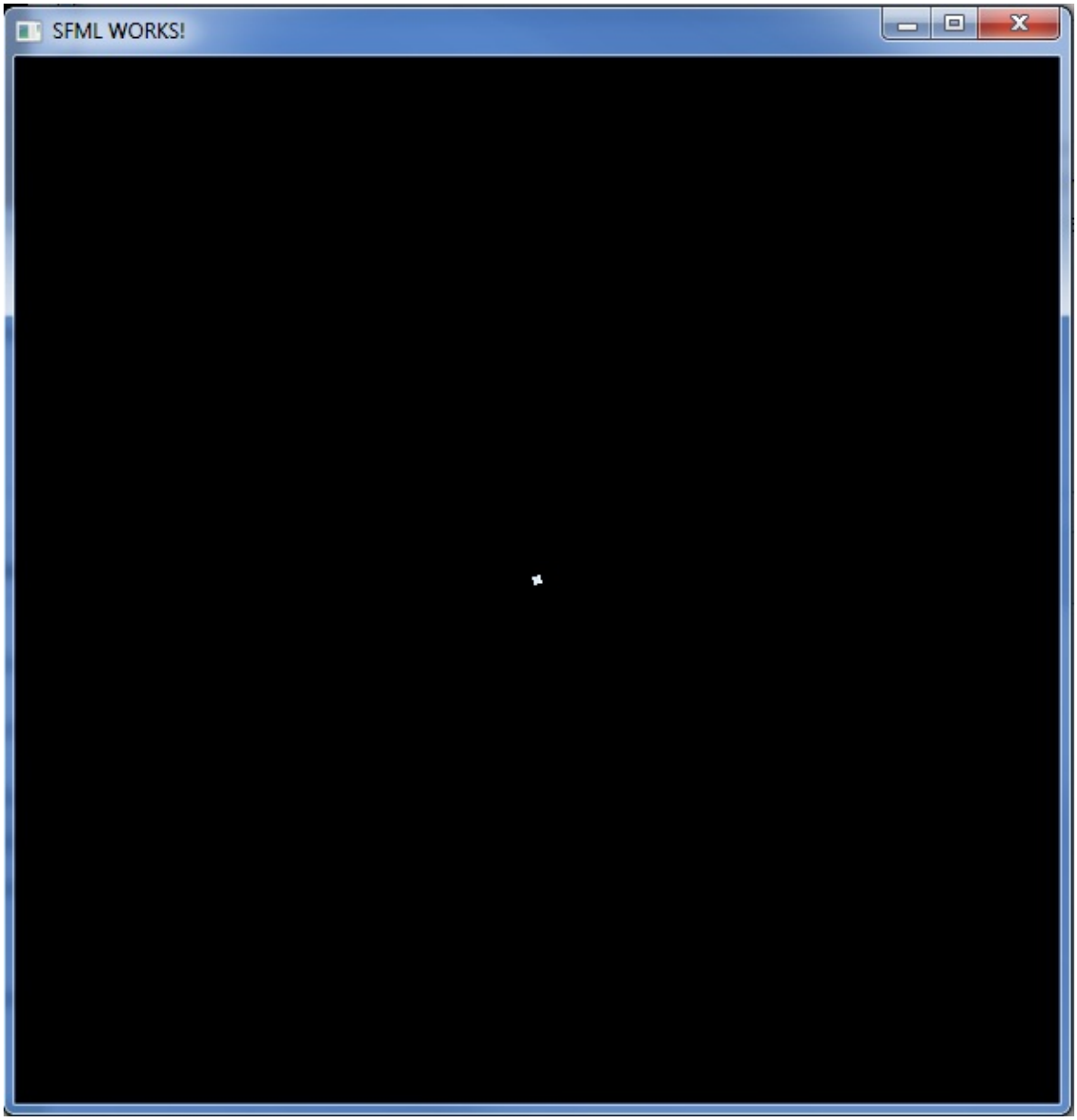
The floating point parameters for zoom scale by: (can also take integers)

- > 1; zoom out
- = 1; no change
- < 1; zoom in

Negative values are irrelevent, and if they are used will be abs(param). This is demonstrated here:

```
    view.zoom(-75.0f);
```

- Is stackable



The view and addition be scaled reletive to the window through getSize(), and overwriting all previous zooms. It is zooms, set function brother.

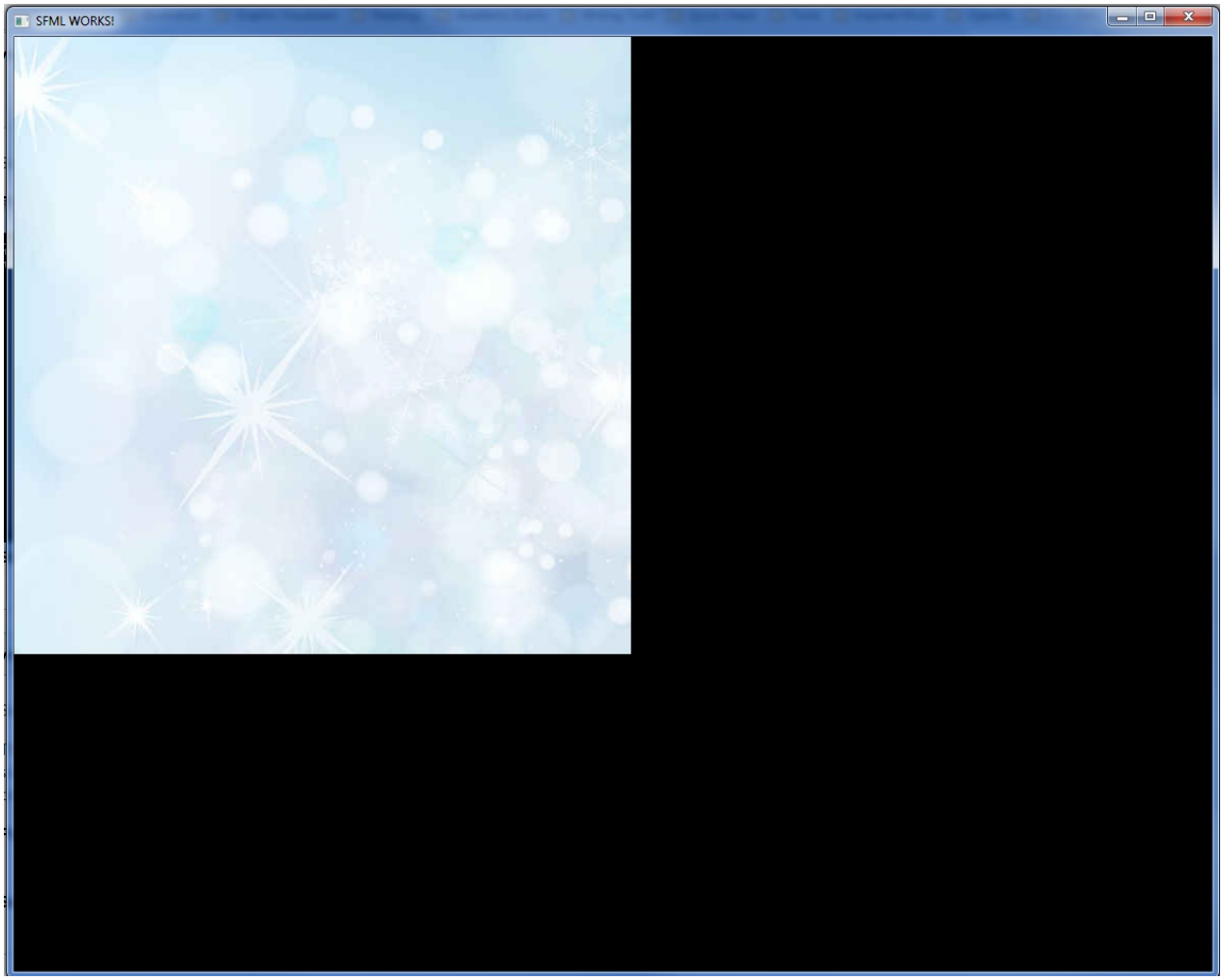
```
    view.zoom(1000);
    view.setSize(sf::Vector2f(1000, 1000));
```

Setting Window Proportionality Constraints

If, in the case we want to resize our window and not have our context inside the window not disfigured, we set up a case event:

```
            case sf::Event::Resized:
                sf::FloatRect area(0, 0, event.size.width, event.size.height);
                window.setView(sf::View(area));
                break;
```

And so in the event of a resize, the dimensions will adjust independently from the origin and the width and height of the resized window.


SFML WORKS!

Sound

Our canvas:

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

using namespace std;


int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;
            }
        }
        window.clear();



        window.display();
    }
}
```

Sound Effects

These refer to small music files that can fit in memory.

Step 1)

```
#include "SFML/Audio.hpp"
```

Step 2)

- Instantiating the sound buffer
- Loading the file
- Playing the sound through the sound buffer

```
    sf::SoundBuffer buffer;

    if (!buffer.loadFromFile("Hammering.wav"))
    {
        cout << "Error loading wav" << endl;
    }

    sf::Sound sound;
    sound.setBuffer(buffer);

    sound.play();
```

One thing to note here, is what .mp3 files were not supported. I had to convert it into a .wav (uncompressed .mp3).

Playing a Music File

The process is equivalent to above. Ill just choose another epic sound track:

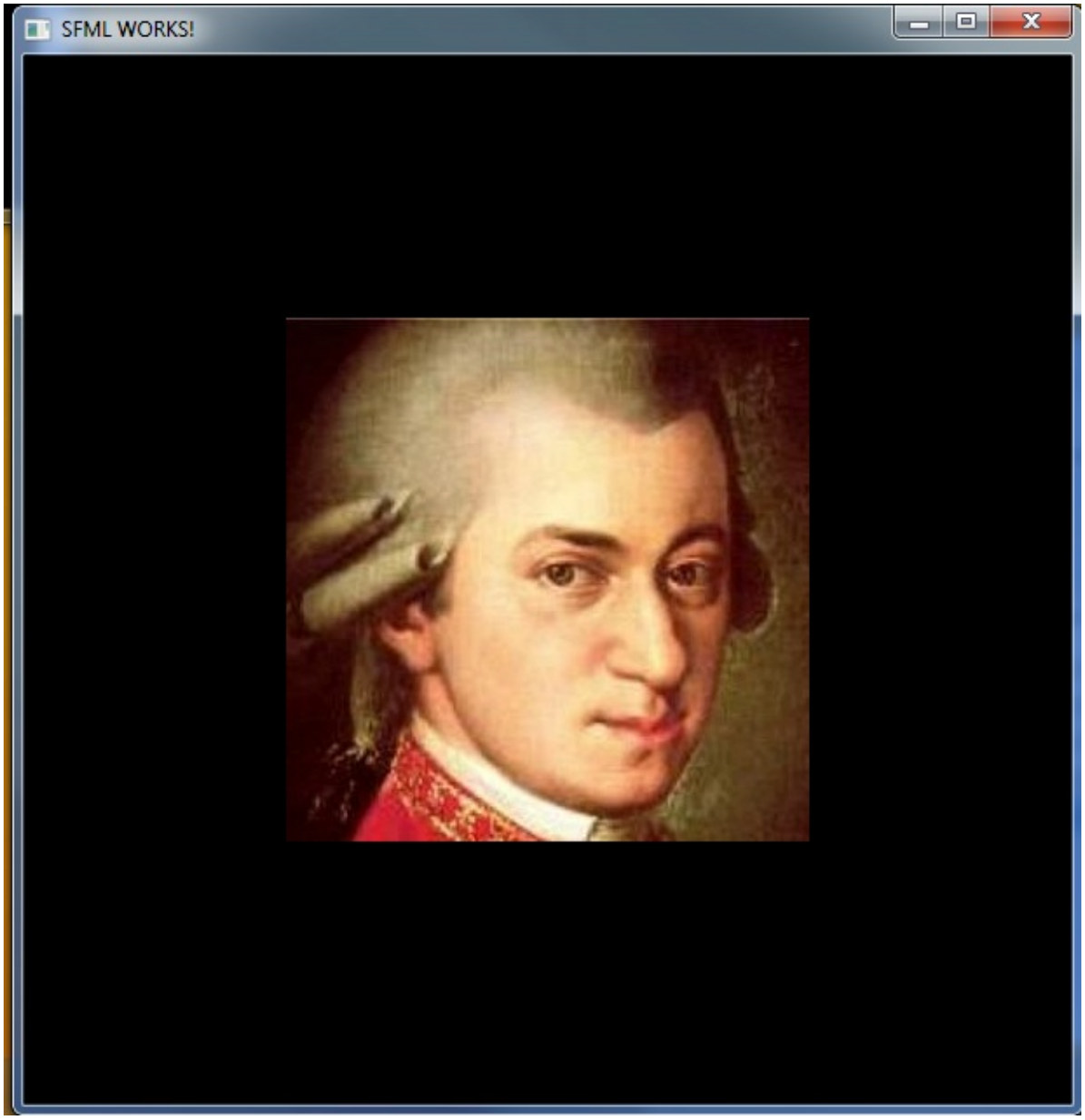
```
    if (!buffer.loadFromFile("Mozart - Requiem.wav"))
```

Which turned out to a whopping 391 MB when I converted .mp3 -> .wav (yeah, I should probably use another audio platform). It took quite a while to load, but hey, the sound quality was fantastic.

For reference, the supported file types are: ogg, wav, flac, aiff, au, raw, paf, svx, nist, voc, ircam, w64, mat4, mat5 pvf, htk, sds, avr, sd2, caf, wve, mpc2k, rf64.

And just for fun, lets make this a little more appropriate:

```
    sf::Texture texture;

    texture.loadFromFile("mozart.jpg");

    sf::Sprite sprite(texture);
    sprite.setPosition(sf::Vector2f(150, 150));

    ...

    window.draw(sprite);
```



An update: I've converted the music to an ogg extension to save some time. (391 - > 6MB)

```
    if (!buffer.loadFromFile("Mozart - Requiem.ogg"))
```

More Music Functionality and Tools

1). Start time (self explanitory)

```
    // start 15 seconds ahead
    sound.setPlayingOffset(sf::seconds(15));
```

2). Sound Volume control

```
    // parameter controlled as a percentage
    sound.setVolume(5);
```

3). Stop sound

```
sound.stop();
```

Important Considerations

- Use sf::SoundBuffer for short soundeffects. [Seen above, I have done this inappropriately]
- Use sf::Music to handle musical themes.
  - This is more effective, since the music stream in chunks, rather than loading that data from hard disk as a whole (as done with sf::SoundBuffer.

  -

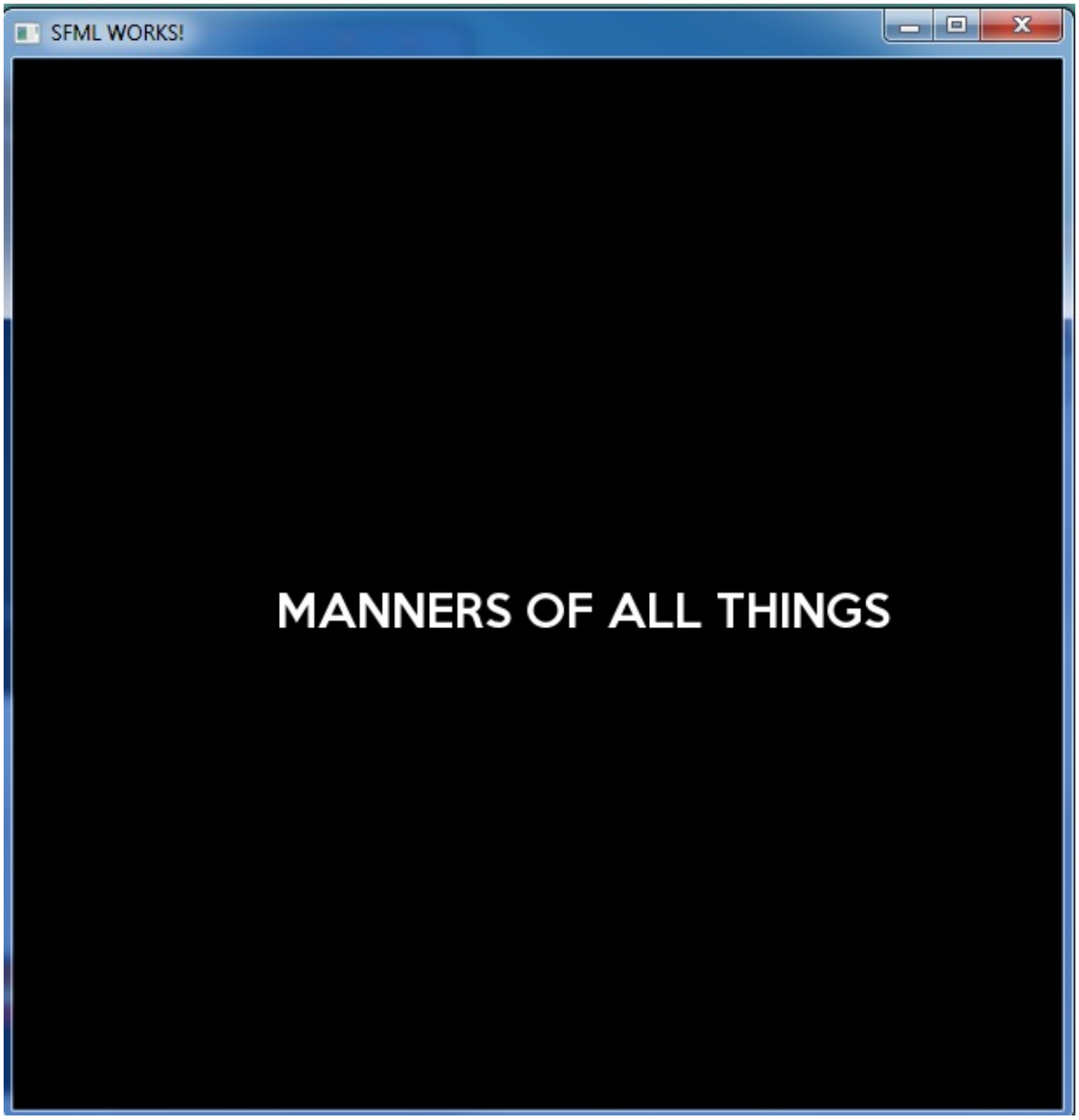
Text

You may notice that instantiating text is very similiar with loading textures and sound. First, we load the font.

```
sf::Font font;

if (!font.loadFromFile("KeepCalm-Medium.otf"))
{
    // handle error
}
```

Next, and very simply, we can apply the properties we want for the text.

```
sf::Text text;
text.setFont(font);
text.setString("MANNERS OF ALL THINGS");
text.setColor(sf::Color::White);
text.setCharacterSize(25);
text.setPosition(150, 300);

...

window.draw(text)
```


SFML WORKS!
MANNERS OF ALL THINGS

Window Applications

Prevent Window Resize

Our canvas:

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

using namespace std;

int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");


    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;
            }
        }
        window.clear();



        window.display();
    }
}
```

According to the lastest data (2015), the most popular screen resolution is 1366×768. If we wanted to prevent window resize (which can serve useful if you want to have comparitable control over features), we can do so.

Lets make the changes we want.

```
    sf::RenderWindow window(sf::VideoMode(1366, 768), "My SFMLPad", sf::Style::Titlebar | sf::Style::Close);
```

Window FullScreen

A single word is changed:

```
sf::RenderWindow window(sf::VideoMode(1366, 768), "My SFMLPad", sf::Style::Fullscreen | sf::Style::Close);
```

We can use ALT+ESC to get out of fullscreen, in order to close the cmd prompt. You also may notice an uncomfortable shift in quality. This is reletive to the resolution you've set. So if you'd like you get set it to your personal resolution:

```
sf::RenderWindow window(sf::VideoMode(1920, 1080), "My SFMLPad", sf::Style::Fullscreen | sf::Style::Close);
```

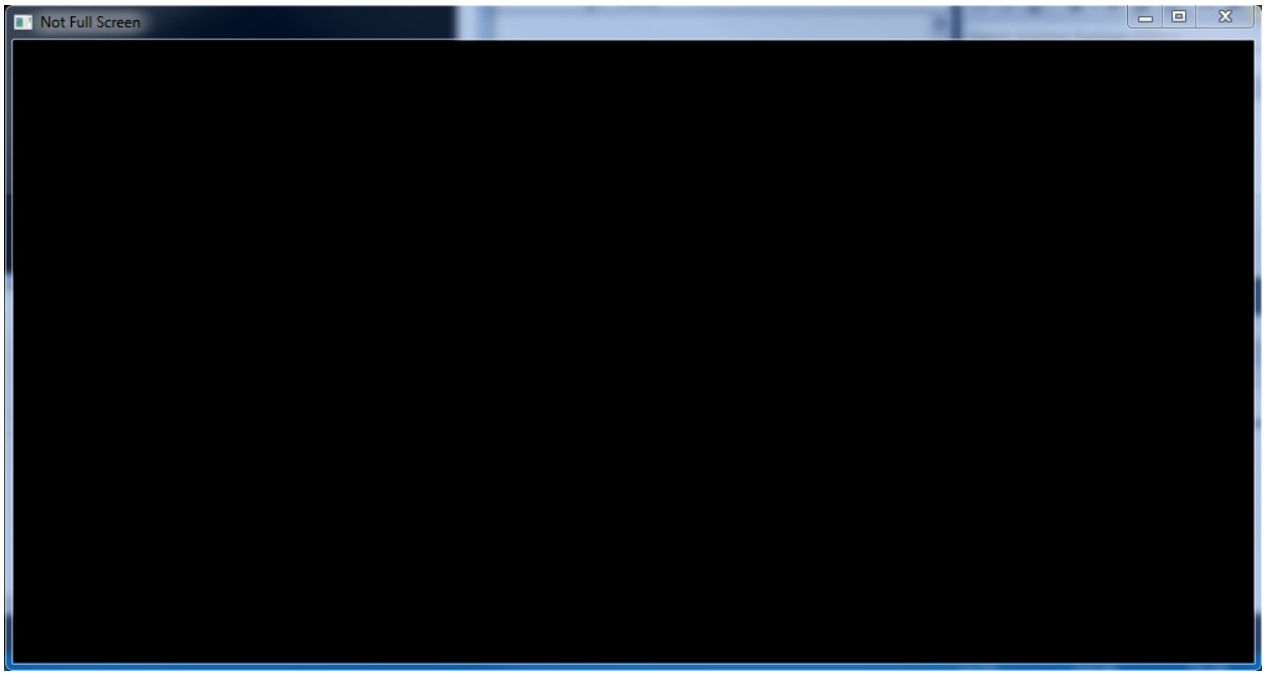
Switching between fullscreen and windowscreen

Lets say we wanted our user to toggle between full and window screen. This can be handled through a case statement and a single boolean variable.

```
bool FullScreen = true; // true based off original window rendering

...

        case sf::Event::KeyPressed:
            switch (event.key.code)
            {
            case sf::Keyboard::Return:
                if (Fullscreen)
                {
                    window.create(sf::VideoMode(1000, 500), "Not Full Screen", sf::Style::Default);
                    Fullscreen = false;
                }

                else
                {
                    window.create(sf::VideoMode(1920, 1080), "Is Full Screen", sf::Style::Fullscreen | sf::Style::C
                    Fullscreen = true;
                }
```

```
            }
            break;
```

One difference to note is window.create() that is used to toggle between window views.



Hiding the Mouse

This can serve useless when, for example, we want to avoid the cursor getting in the way with our gameplay experience. It can, for example, be customized with an arrow and enter keyboard interface.

Lets set this standard for both non and full screen windows.

```
window.setMouseCursorVisible(false);

...

window.setMouseCursorVisible(false);
```

Removing the Console Window

The console is great for debugging purposes, where SFML will at times link where an issue is located. A finished application however, it would safe to assume that it is not needed anymore.

Code is not nessessary here. The change is made in our project proporties (visual studio):

Project Properties -> Linker -> System -> Subsystem -> change to "Windows (/SUBSYSTEM:WINDOWS)"

Frame Rate

- 60FPS is a developemental standard for gameplay.
- SFML has built in functionality to calculate the time between frames.

Our Canvas:

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

using namespace std;

int main()
{
    sf::RenderWindow window(sf::VideoMode(1000, 500), "My SFMLPad");


    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;
            }
        }
        window.clear();



        window.display();
    }
}
```

Calculating time within each frame

Instantiate SFML time control. Clock is used to measure elapsed time, while is used to represent a time value.

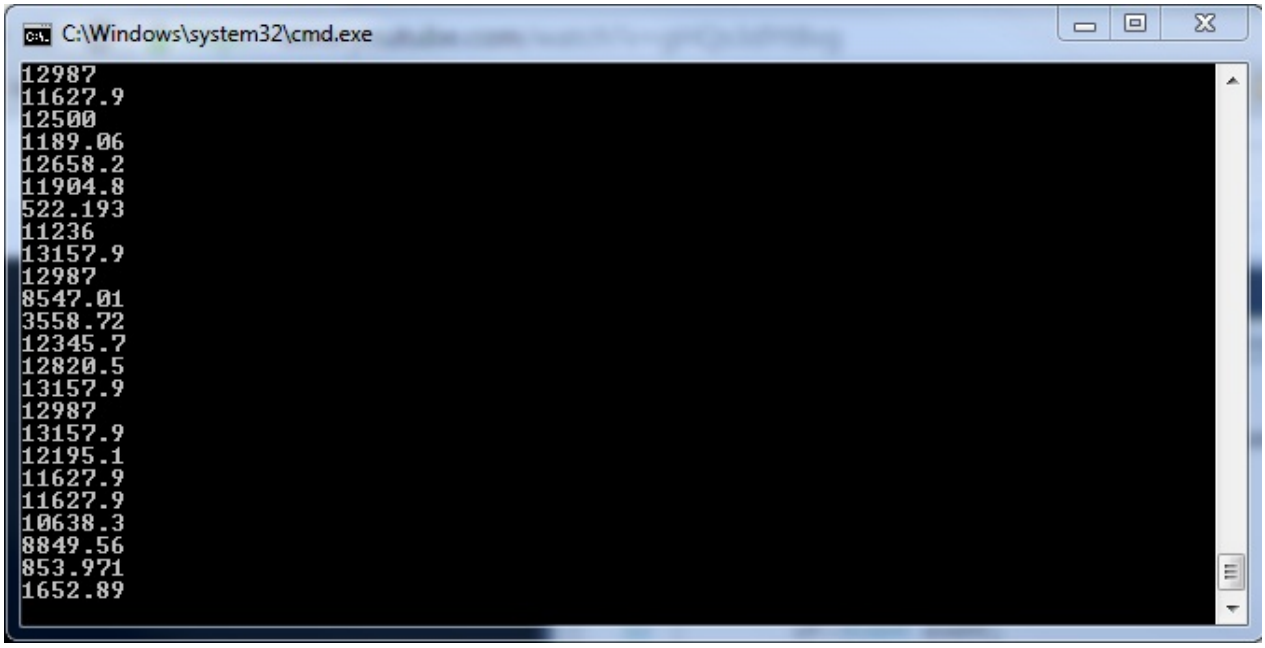
```
    sf::Clock clock;
    sf::Time time;
```

If we want to display the time required for each individual frame, what we want to do

```
        time = clock.getElapsedTime();
        cout << time.asSeconds() << endl;
        clock.restart().asSeconds();
```

This displays the $$ seconds/frame $$. We want its reciprocal, so lets inverse it.

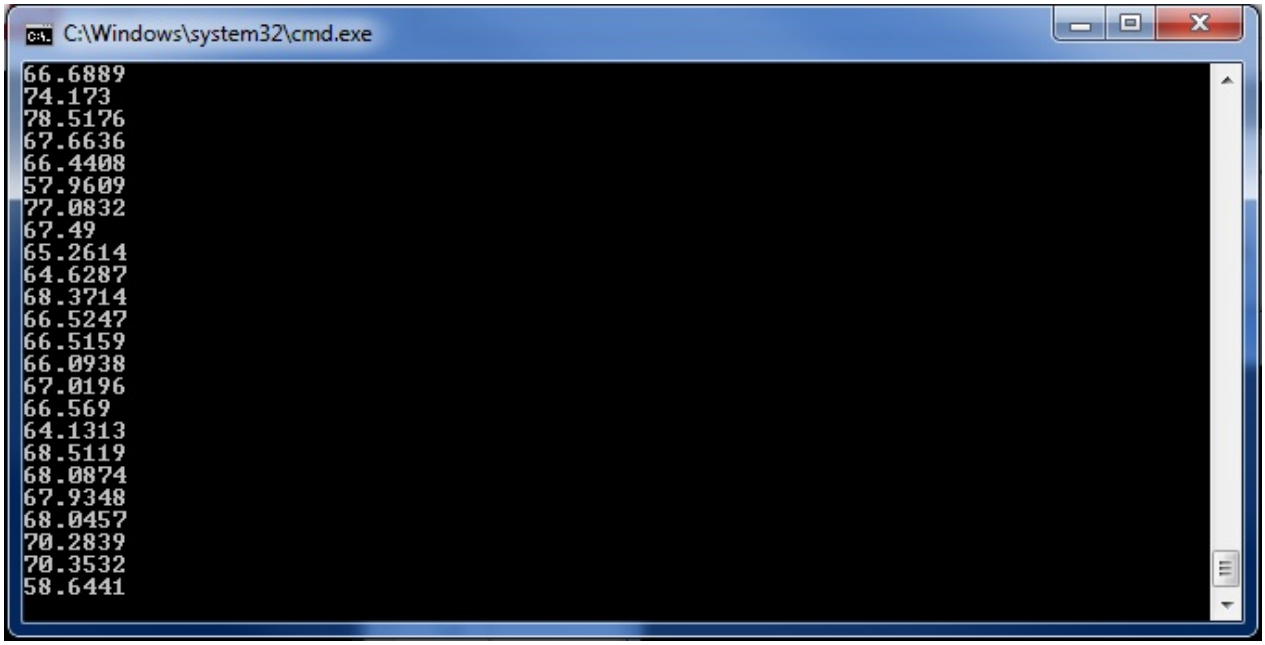
```
    cout << 1.0f/time.asSeconds() << endl;
```



Limiting Frame Rate

13000 frames/second is a bit overdoing it. To limit its capacity:

```
    window.setFramerateLimit(60);
```



Frame Rate Independent Gameplay

Frame rate can be independent on an individuals machine (computer). As game developers, our job is to make sure that the gameplay is consistant on all machines with independent frame rates.

e.g. Take for example, a racing game. A machine with a great graphics card would speed through the stage, while an more older card would go slower - and dependent on their frame rate.

Sounds unnatural, but this has to be controlled through frame rate independent gameplay.

To demonstrate this, lets animate movement on a sprite:

```
sf::Texture texture;
sf::Sprite sprite;

texture.loadFromFile("runner.jpg");

sprite.setTexture(texture);

...

window.draw(sprite);
```

Remember, `while (window.isOpen())` represents an individual frame. So while this is begin run, we can animate movement through the frames.

```
    sprite.move(sf::Vector2f(0, 0.1f));
```

//some time after 5 seconds

Now previously, we've set `window.setFramerateLimit(60);`. When applied, the limitation of the frame rate will control the speed of the moving picture. Without it, it is very likely (machine dependent), that the speed would 3 to 4 times more fast.

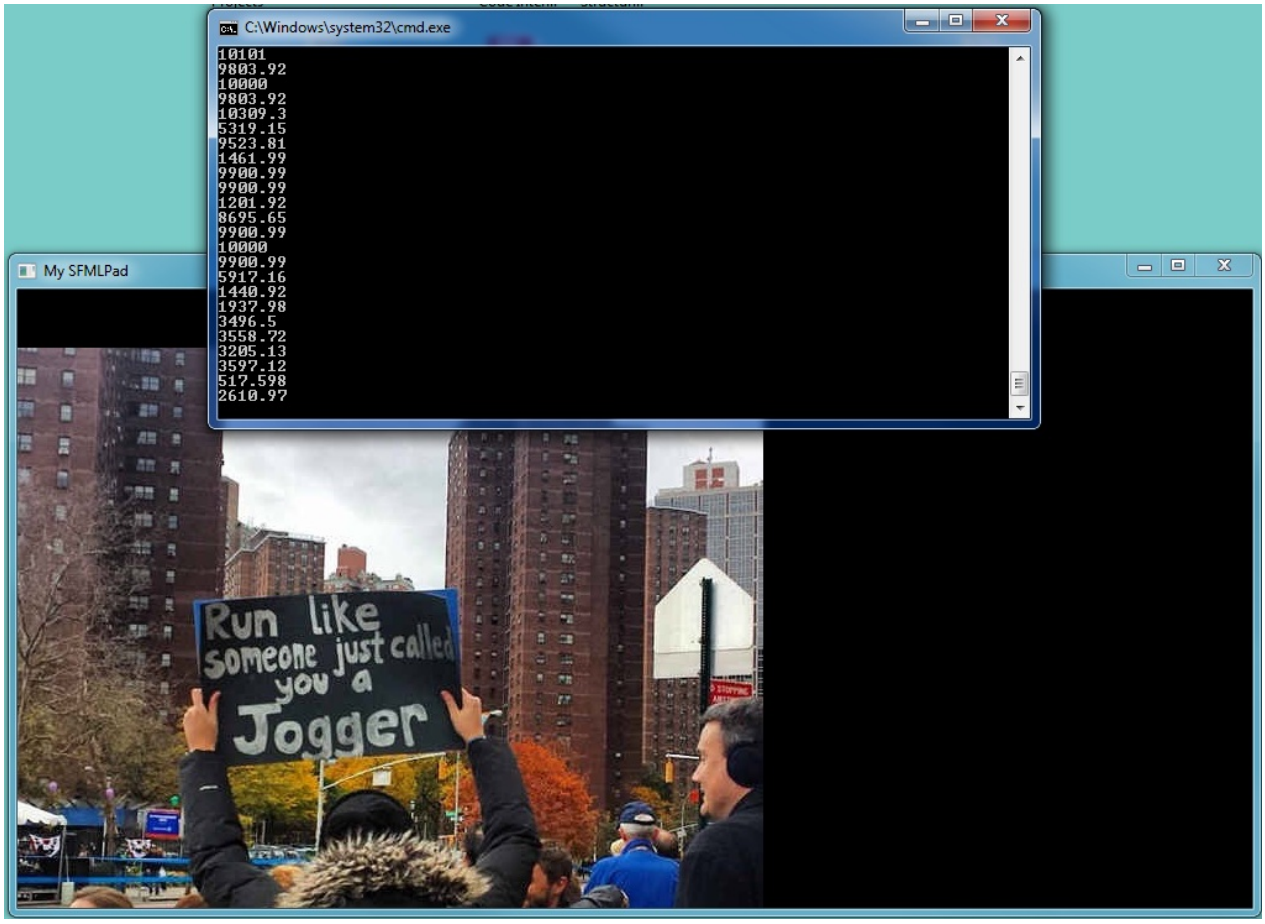
Setting this limitation does not guarentee what we exactly want. For example, Say the frame rate was limited to 140FPS. In the situation of two independent platforms, one that can handle this intensity and one that cannot, it would be clear which machine would have the advantage.

To fix this, what we have to do maintain a constant speed, regardless of the frame rate. So if we were to take the same example from above, regardless of the frame rate, movement on a sprite is limited to how much time elapsed (which is a literally a global constant).

So lets constrain movement based off the time elapsed.

```
    window.setFramerateLimit(1);

    ...
```

```
        time = clock.getElapsedTime();

        sprite.move(sf::Vector2f(0, 0.2 * time.asMilliseconds()));
        cout << 1.0f / time.asSeconds() << endl;
        clock.restart().asSeconds();

        window.clear();

        window.draw(sprite);

        window.display();
```



Lets do a quick test. In the properties above, it took about 3 seconds to have the image move disappear down the window.

Now lets change the frame rate.

```
        window.setFramerateLimit(60);
```

Also 3 seconds. Frame independent gameplay ladies and gentlemen.

Collisons

SFML had built in collision detection handling between two sprites.

Canvas:

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

using namespace std;


int main()
{
    sf::RenderWindow window(sf::VideoMode(600, 600), "SFML WORKS!");

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;
            }
        }
        window.clear();


        window.display();
    }
}
```

First things, first load up the textures as sprites.

```
    sf::Texture texture;
    sf::Texture texture_two;

    texture.loadFromFile("collision1.png");
    texture_two.loadFromFile("collision2.png");

    sf::Sprite sprite;
    sprite.setTexture(texture);
    sprite.setPosition(sf::Vector2f(0, 250));
    sprite.setScale(sf::Vector2f(0.2, 0.2));

    sf::Sprite sprite_two;
    sprite_two.setTexture(texture_two);
    sprite_two.setPosition(sf::Vector2f(925, 250));
    sprite_two.setScale(sf::Vector2f(0.2, 0.2));

    ...

    window.draw(sprite);
    window.draw(sprite_two);
```

Simple enough. Nows lets animate movement.

```
        if (sprite.getGlobalBounds().intersects(sprite_two.getGlobalBounds()))
        {
            cout << "Collision!" << endl;
```

```
        }

        else
        {
            sprite.move(sf::Vector2f(0.05f, 0));
            sprite_two.move(sf::Vector2f(-0.05f, 0));
        }
```

Because we want to update each frame with a set amount of movement, we place this in `while(window.isOpen())`.

getGlobalBounds returns the coordinates of the bounding rectangular box of the sprite we loaded in. Intersection is determined when any of these return coordinates first return the same value, just like two intersecting functions.

// Pretty Cool

A Step Forward: Utilizing Object Orientation and Classes

Your int main function has been great in learning the functionality of SFML really quickly. Often times we had to restart with a 'blank canvas' to represent some ideas. This is because our project would get clustered with multiple abstractions, it would be hard to represent them cumulatively.

So we use classes for organization and simplicity.

Our goal in this first segement is to create a class to represent a 'Player' or sprite object. Lets begin.

// Player.h

```
#pragma once
#include "SFML/Graphics.hpp"
#include <iostream>

class Player
{
public:
    Player();
    ~Player();

    // dt, change in time
    void update(float dt);
    void draw(sf::RenderWindow &window);

private:
    sf::Texture playerTexture;
    sf::Sprite playerSprite;

};
```

Texture and Sprite are class abstractions from SFML. Really what we are doing is introducing class types into another class. Class abstractions can be though of as variables.

//Player.cpp

```
#include "Player.h"

using namespace std;

Player::Player()
{
    if (!playerTexture.loadFromFile("link.png"))
    {
        cout << "Error loading link.png" << endl;
    }

    playerSprite.setTexture(playerTexture);
}


Player::~Player()
{
}

void Player::update(float dt)
{

}

void Player::draw(sf::RenderWindow &window)
{
    window.draw(playerSprite);
```

```
}
```

Here we instantiate what we need when to load up our gameplay profile.

//source.cpp (main)

```
#include "SFML/Graphics.hpp"
#include <iostream>
#include "Player.h"

using namespace std;

int main()
{
    sf::RenderWindow window(sf::VideoMode(1000, 500), "My SFMLPad");

    Player player;

    sf::Clock clock;
    sf::Time time;

    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;
            }
        }

        time = clock.getElapsedTime();
        player.update(time.asSeconds());
        clock.restart().asSeconds();



        window.clear();

        player.draw(window);

        window.display();
    }
}
```

Our main class is meant to handle frames. So we initiate clock and time.

Menu Class

It is common practice for most games to provide a menu for the user to interact with, to for example, determine level selection, set resolutions, or change any settings.

Our next task is implement, very simply, this idea.

Beginning with a blank canvas:

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

using namespace std;


int main()
{
    sf::RenderWindow window(sf::VideoMode(1000, 500), "SFML WORKS!");



    while (window.isOpen())
    {
        sf::Event event;

        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::Closed:
                window.close();
                break;
            }
        }


        window.clear();



        window.display();
    }
}
```

Go ahead and create a new class to represent the menu.

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


class Menu
{
public:
    Menu();
    ~Menu();
};
```

When we instantiate our menu, the first thing we need to do is render a window. So lets construct it like so.

```
    Menu(float width, float height);
```

The private variables that our class will need access to is a font, and an array of text for the purpose of storing all the text for our menu. `selectedItem;` will be used as a tracker between `[0-MAX_NUMBER_OF_ITMES]` for each menu item. If your read further, this is meant to represent. It is initialized to 0 because by default, play is the first item.

```
menu[0] = Play
menu[1] = Options
menu[2] = Exit
```

```
private:
    sf::Font font;
    sf::Text menu[MAX_NUMBER_OF_ITMES];
    int selectedItem = 1;
```

For dynamic control, we can organize the number of menu items:

```
#define MAX_NUMBER_OF_ITMES 3
```

All our previous sprites where drawn using something like `window.draw(triangle)`. With most classes, we want the same functionality built in, so we understand what we draw based on the class or abstraction. So lets define a function for our menu class that does just that.

```
    void draw(sf::RenderWindow &window);
```

A rendered window will then take something like `sf::VideoMode(1000, 500), "SFML WORKS!"`. And that is what we will exactly pass through later on: `menu.draw(window);`.

Next we will need some functionality for user to navigate through the window. We will define these later.

```
    void moveUp();
    void moveDown();
```

Now lets provide the implementation for our menu.

//Menu.cpp

```
#include "Menu.h"
```

Our constructor should immediately load the contents we need for our window, and apply the properties of display.

```
Menu::Menu(float width, float height)
{
    if (!font.loadFromFile("KeepCalm-Medium.otf"))
    {
        return;
    }

    menu[0].setFont(font);
    menu[0].setColor(sf::Color::Magenta);
    menu[0].setString("Play");
    menu[0].setPosition(sf::Vector2f(width / 2, height / (MAX_NUMBER_OF_ITMES + 1) * 1));

    menu[1].setFont(font);
    menu[1].setColor(sf::Color::White);
    menu[1].setString("Options");
    menu[1].setPosition(sf::Vector2f(width / 2, height / (MAX_NUMBER_OF_ITMES + 1) * 2));

    menu[2].setFont(font);
    menu[2].setColor(sf::Color::White);
    menu[2].setString("Exit");
    menu[2].setPosition(sf::Vector2f(width / 2, height / (MAX_NUMBER_OF_ITMES + 1) * 3));
}
```

Notice how our menu array, which is of type `sf::Text`, is nicely packed into the number of menu items we want dynamically (no smart numbers).

Next, our menu draw should take a RenderWindow, as we've always given a name throughout this series as 'window'. Going through our menu array, we would call `window.draw(text)`, except as in the context of the array.

```
void Menu::draw(sf::RenderWindow &window)
{
    for (int i = 0; i < MAX_NUMBER_OF_ITMES; i++)
    {
        window.draw(menu[i]);
    }
}
```

Finally, lets update our source file.

//source.cpp

To make sure things work the way we want them to, lets display our menu items.

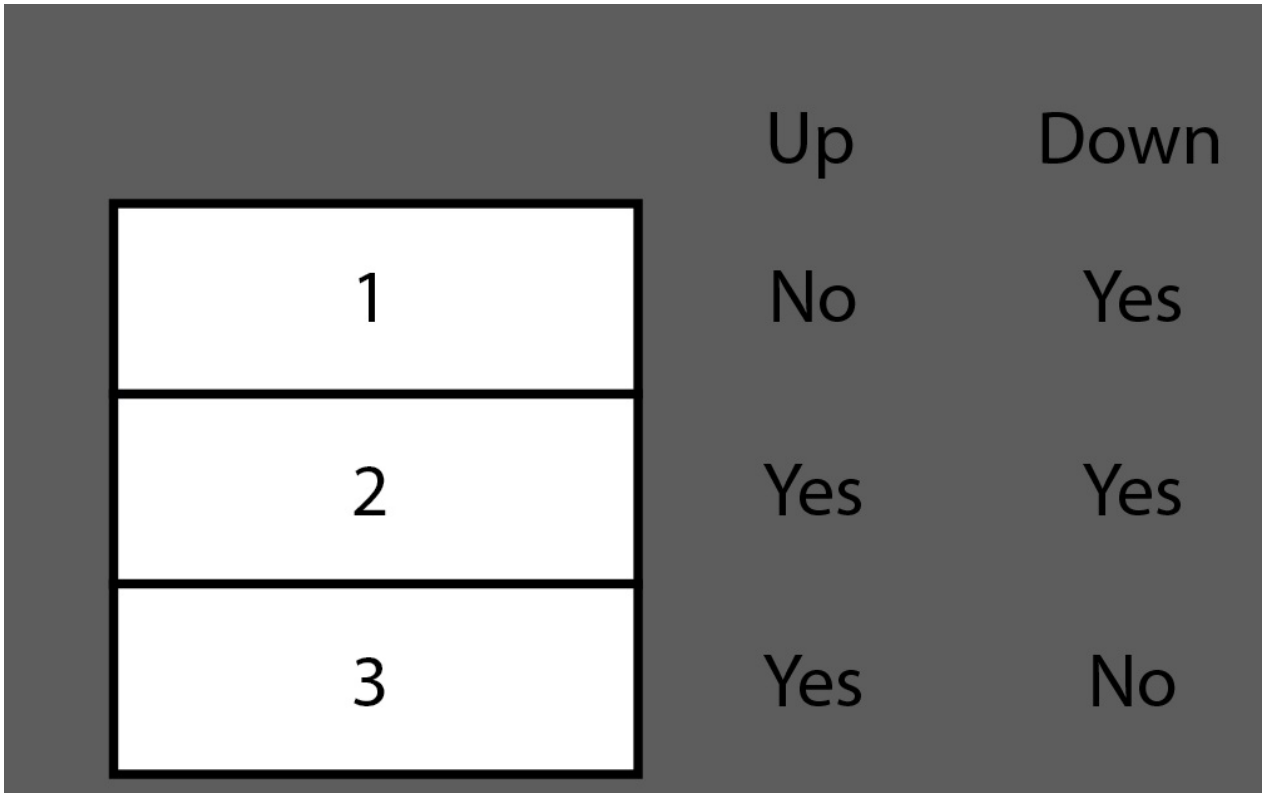
```
#include "Menu.h"
```

```
    Menu menu(window.getSize().x, window.getSize().y);

    ...


    menu.draw(window);
```

Lets move on to implement the arrow key user navigation for our menu.

```
void Menu::moveUp()
{
    if (MAX_NUMBER_OF_ITMES - 1 <= selectedItemIndex+1 && selectedItemIndex+1 <= MAX_NUMBER_OF_ITMES)
    {
        menu[selectedItemIndex].setColor(sf::Color::White);
        selectedItemIndex--;
        menu[selectedItemIndex].setColor(sf::Color::Magenta);
    }
}

void Menu::moveDown()
{
    if (MAX_NUMBER_OF_ITMES - 2 <= selectedItemIndex + 1 && selectedItemIndex + 1 <= MAX_NUMBER_OF_ITMES-1)
    {
        menu[selectedItemIndex].setColor(sf::Color::White);
        selectedItemIndex++;
        menu[selectedItemIndex].setColor(sf::Color::Magenta);
    }
}
```

The two functions are nearly identical, and they should be because they are reverse each other like a mirror. The if conditions are essential. Here we only want to navigate up, if our selectedItemIndex is with 2 <= selectedItemIndex <= 3, and move down only if 1 <= selectedItemIndex <= 2, so that is exactly what I've implemented using `MAX_NUMBER_OF_ITMES`.



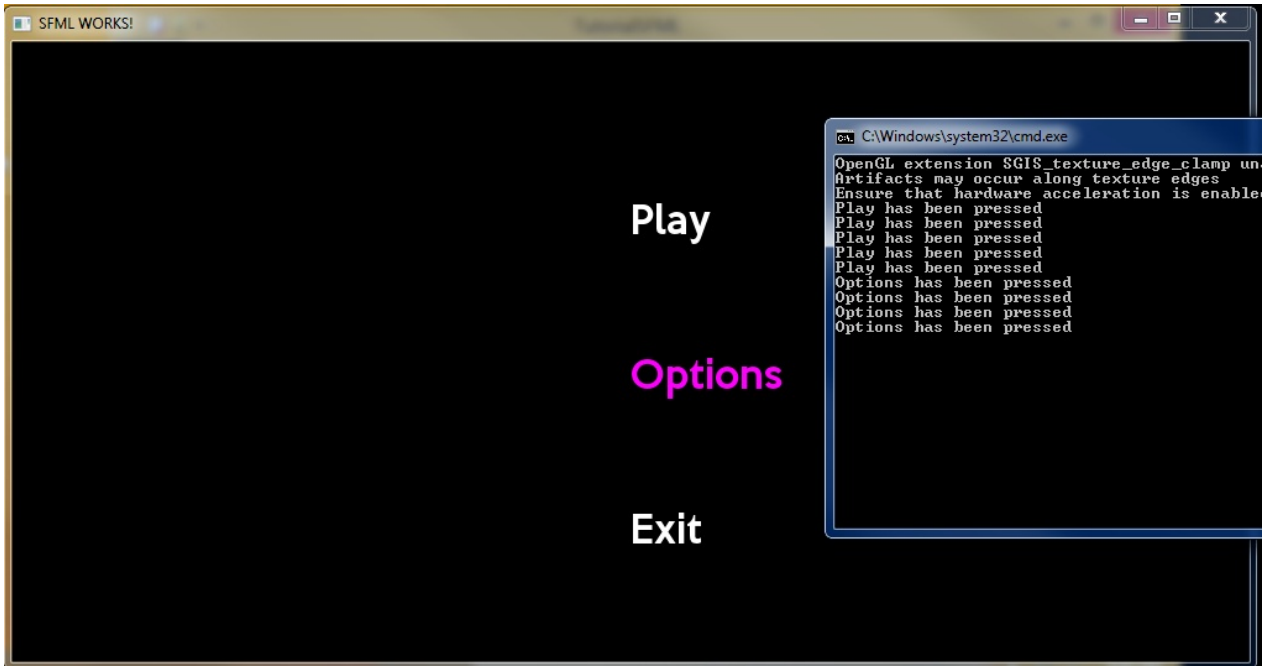
Finally, lets implement some minimal functionality with each menu option. Navigation will be determined by a particular menu item. Lets create a function that handles this.

// Menu.h

```
int getSelectedItemIndex() { return selectedItemIndex; }
```

And its cooresponding action, which are all self-explainitory.

```
            case sf::Event::KeyPressed:
                switch (event.key.code)
                {
                case sf::Keyboard::Up:
                    menu.moveUp();
                    break;
                case sf::Keyboard::Down:
                    menu.moveDown();
                    break;
                case sf::Keyboard::Return:
                    switch (menu.getSelectedItemIndex())
                    {
                    case 0:
                        cout << "Play has been pressed" << endl;
                        break;
                    case 1:
                        cout << "Options has been pressed" << endl;
                        break;
                    case 2:
                        window.close();
                        break;
                    }
                }
```


SFML WORKS!
Play
Options
Exit
C:\Windows\system32\cmd.exe
OpenGL extension SGIS_texture_edge_clamp un
Artifacts may occur along texture edges
Ensure that hardware acceleration is enable
Play has been pressed
Play has been pressed
Play has been pressed
Play has been pressed
Play has been pressed
Options has been pressed
Options has been pressed
Options has been pressed
Options has been pressed

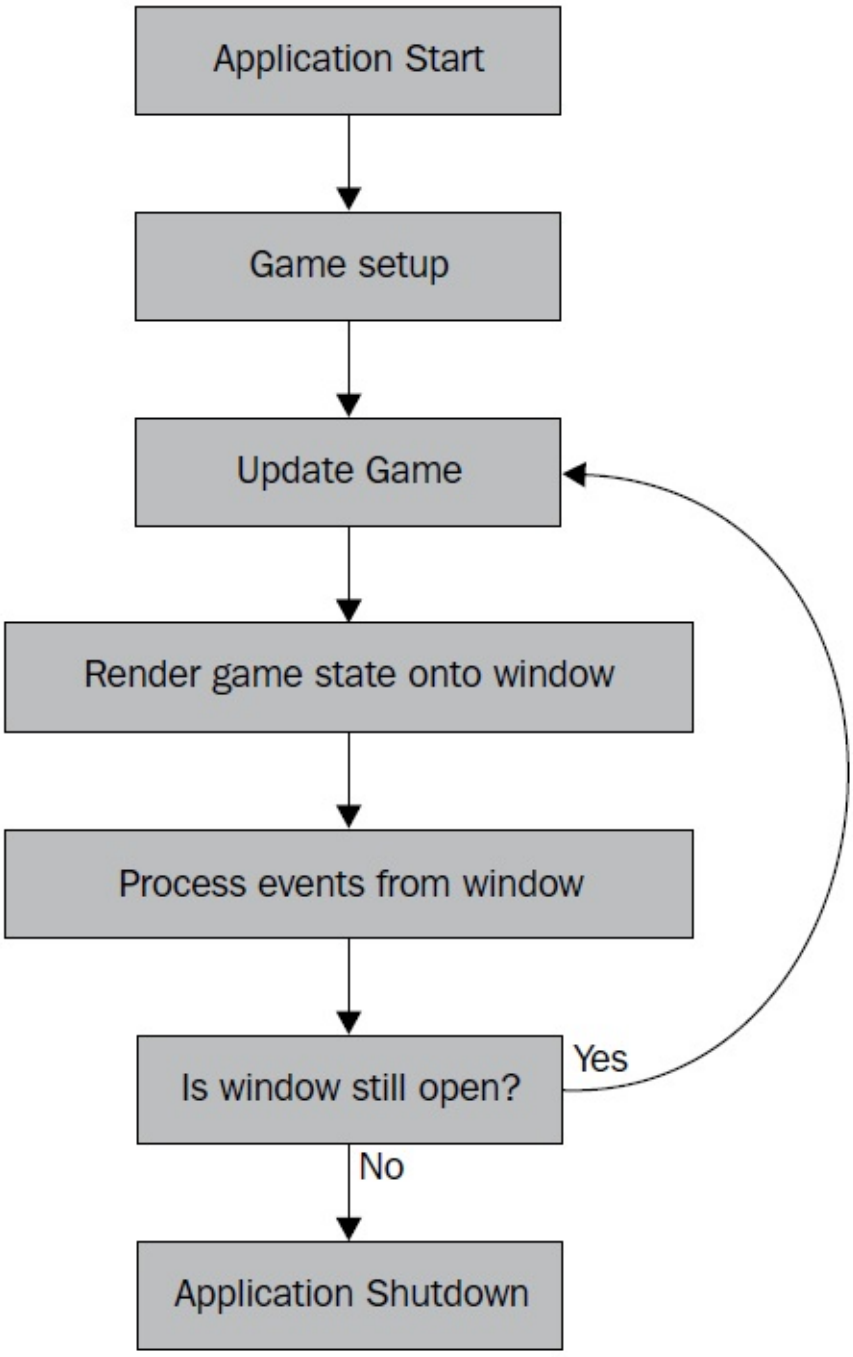
Game Class

- This is your 'main' class. It controls the flow of your program.
  - Use this generic loop to run your program.
    - run()
      - Does everything below, runs one frame to the next.
    - processEvents();
      - Controls user input through 'events'
    - update()
      - Can me used to control movement, for example. Or anything that has to be updated in order for the logic of your game to move along.
    - render()
      - Draws your most recently updated frame onto the window for graphic display.
      - Consists of three parts:

```
window.clear()
window.draw()
window.display()
```

- Initialization:
  - The size of your window, which I would first recommend through by #define
  - If nessessary, but good practice - the frame limit of your window.
- Other things:
  - Our game class can be used to store all of the objects we interact with from other class additions.

This game loop diagram explains the frame process very clearly.



Boiler Plate Code:

// Game.h

```
#pragma once
#include "SFML/Graphics.hpp"

#define WIDTH 1000
#define HEIGHT 500

class Game
{
public:
    Game();
    void run();

private:
    void processEvents();
```

```
    void update();
    void render();

    sf::RenderWindow window;
};
```

// Game.cpp

```
#include "Game.h"

Game::Game() : window(sf::VideoMode(WIDTH, HEIGHT), "Name of Application")
{
    window.setFramerateLimit(60);
}

void Game::run()
{
    while (window.isOpen())
    {
        processEvents();
        update();
        render();
    }
}

void Game::processEvents()
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        switch (event.type)
        {
        case sf::Event::Closed:
            window.close();
            break;
        }
    }
}

void Game::update()
{
}

void Game::render()
{
    window.clear();

    //draw()

    window.display();
}
```

// Source.cpp

```
#include "SFML/Graphics.hpp"
#include "Game.h"

int main()
{
    Game game;
    game.run();

}
```

Class Additions

The next step would be to define a new class abstraction. I've found that with good architectural design, it is often best to define a class when:

- A new object is introduced into the game, that has it's own properties and interactable components.
-

Should contain here definitions:

- Implementation for control over the object
- update() function if the object has movement

Boiler Plate Code:

```
void Ship::handleArrowKeyMovement(sf::Keyboard::Key key, bool isPressed)
{
    if (key == sf::Keyboard::W)
        IsMovingUp = isPressed;
    else if (key == sf::Keyboard::S)
        IsMovingDown = isPressed;
    else if (key == sf::Keyboard::A)
        IsMovingLeft = isPressed;
    else if (key == sf::Keyboard::D)
        IsMovingRight = isPressed;
}
```

User Control

I've specified earlier that user control is 'best' defined in `processEvents()` part of the game loop. But after specifing the 'event' it is best to handle in output through a game class function.

Boiler Plate Code:

```
void Game::processEvents()
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        switch (event.type)
        {
        case sf::Event::Closed:
            window.close();
            break;
        case sf::Event::KeyPressed:
            ship.handleArrowKeyMovement(event.key.code, true);
            break;
        case sf::Event::KeyReleased:
            ship.handleArrowKeyMovement(event.key.code, false);
            break;
        }
    }
}
```

State Machines

Goal

State machines are what allow us to transition between different modes or states of the game. Levels, pauses, menus, settings, etc. Earlier we've implemented a menu class. State machines are what give them practice use.

Becomes implemented in what is known as generically known as the 'Game Class'

Lets Begin

Begin with a list of gamestates. Enums will guarentee that all the gameStates have different associative values.

```
enum class GameStates
{
    STATE_START = 1;
    STATE_MENU; // 2
    STATE_OPTIONS; // n + 1
    STATE_SETTINGS;
    STATE_LEVEL;
    //.. etc
}
```

These are used to represent all of the game states we could possibly have. Then within int main() set the state you want your game to initialize with:

```
int main()
{
    GameStates gameState = GameStates::STATE_START;


}
```

Next we can begin to transfer between different game states while the window is open (which essentially is a game state in and of itself).

```
    while (window.isOpen())
    {

    }
```

The first method in handling game states is to use switch. This is not the best way, but it solves the job with smaller games. Larger games would require an object oriented approach.

The first approach uses a switch method that orients the program based off of the gamestates. Each menu state has its associative functions.

```
    switch( gameState )
        case STATE_MENU:
            // functions
        case STATE_OPTIONS:
            // functions
        case STATE_SETTINGS:
            // functions
        case STATE_LEVEL:
```

```
            // functions
```

Finite State Machine

References: http://lazyfoo.net/articles/article06/index.php

Resource Handling

- It is often best to load all the resources in advance, rather than on an on-demand basis. Possible loading issues from the game are avoided, and non-interrupted.
- Deletion should be called shortly after it they are no-longer needed.

- **RAII**

- Resources are acquired from the constructor and released from the destructor on non-manual basis (out of scope).
- Manual deallocation and allocation is guaranteed to take place on call.
-

Organizing Textures

```
namespace Textures
{
    enum ID { Landscape, Airplane, Missile };
}
```

1. Namespace allows us to clearly describe our intention. e.g. 'Textures::Airplane'.
2. We than wrap them around in unique_ptr for reasons:
    i. Automatically delete in destructor.
    ii. Move without copying.

```
class TextureHolder
{
private:
    std::map<Textures::ID, std::unique_ptr<sf::Texture>> TextureMap;
public:
    TextureHolder();
    ~TextureHolder();
};
```

Our next task is create a method that loads the textures we need based off the texture ids we've specified, and the filepath. `std::map` nicely holds everything in place.

```
    <Landscape, sf::Texture>
    <Airplane, sf::Texture>
    <Missile, sf::Texture>
```

```
    void load(Textures::ID id, const std::string &filename);
```

```
void TextureHolder::load(Textures::ID id, const std::string &filename)
{
    std::unique_ptr<sf::Texture> texture(new sf::Texture());
    texture->loadFromFile(filename);

    TextureMap.insert(std::make_pair(id, std::move(texture)));
}
```

Lets take a look at the improvements we've made from our previous approach:

```
    sf::Texture texture;

    if (!texture.loadFromFile(filepath))
    {
        std::cout << "error loading from file" << std::endl;
    }
```

1. sf::Texture texture is largely singular and can become redundent.
2. Enums organize the textures we know that we will use with an encapulated ID for each texture.
3. Rather than constructing a single sf::Texture on the spot, the combination of a map + unique_ptr does the following:
    i. Creates a new texture.
    ii. Loads the texture
    iii. Inserts the texture into a map with. (Vectors are avoid because push_back would make an unnessessary copy).
    iv. std::move literally take one object and places it onto something else. We therefore avoid duplicates: a texture object is created, loaded, and then move directed to a map with an ID.

Our next task to retrieve the information we've store into our map.

```
 sf::Texture& get(Textures::ID id) ;
```

```
 sf::Texture& TextureHolder::get(Textures::ID id)
{
    auto found = TextureMap.find(id);
    return *found->second;
}
```

Here, .find search from the provided enum id, and returns its `second` marked pair, the sf::texture.

Next, we will also provide a const overload with the same implementation, so we allow our compiler to decide pair of the subscript remains const. In total, we will have the following code:

```
#pragma once
#include "Game.h"
#include <memory>

namespace Textures
{
    enum ID { Landscape, Airplane, Missile };
}

class TextureHolder
{

public:
    TextureHolder();
    ~TextureHolder();
    void load                          (Textures::ID id, const std::string &filename);
    sf::Texture& get                  (Textures::ID id);
    const sf::Texture& get              (Textures::ID id) const;

private:
    std::map<Textures::ID,
        std::unique_ptr<sf::Texture>> TextureMap;
};
```

```
#include "TextureHolder.h"
```

```
TextureHolder::TextureHolder()
{
}


TextureHolder::~TextureHolder()
{
}

void TextureHolder::load(Textures::ID id, const std::string &filename)
{
    std::unique_ptr<sf::Texture> texture(new sf::Texture());
    texture->loadFromFile(filename);

    TextureMap.insert(std::make_pair(id, std::move(texture)));
}

sf::Texture& TextureHolder::get(Textures::ID id)
{
    auto found = TextureMap.find(id);
    return *found->second;
}

const sf::Texture& TextureHolder::get(Textures::ID id) const
{
    auto found = TextureMap.find(id);
    return *found->second;
}
```

Now our information can become easily loaded, and retrieved from our map settings.

```
TextureHolder textures;
textures.load(Textures::[insert id], "filepath")

sf::Sprite [id];
missile.setTexture(textures.get(Textures::[insert id]));
```

Further Optimization through Error Handling

1) Throw a run_time error for notification. 2) assert() - useful for debugging. A break point is triggered, haulting program execution.

```
void TextureHolder::load(Textures::ID id, const std::string &filepath)
{
    std::unique_ptr<sf::Texture> texture(new sf::Texture());
    if (!texture->loadFromFile(filepath))
        throw std::runtime_error("TextureHolder::load - Failed to Load from" + filepath);

    auto inserted = TextureMap.insert(std::make_pair(id, std::move(texture)));
    assert(inserted.second);
}
```

Similiarily, we can note whether the id has been found in our get() method.

```
const sf::Texture& TextureHolder::get(Textures::ID id) const
{
    auto found = TextureMap.find(id);
    assert(found != TextureMap.end());
    return *found->second;
}
```

Generalizing the approach

Because the implementation is largely congruent, it would be good architectural practice to design a class template that works with all media forms.

```
* sf::SoundBuffer
* sf::Font
```

Using our previous class, we can adapt similiar setting based on the type provided.

```
#pragma once
#include "Game.h"
#include <assert.h>
#include <memory>

namespace Textures
{
    enum ID { Landscape, Airplane, Missile };
}


class ResourceHolder
{

public:
    ResourceHolder();
    ~ResourceHolder();
    void load(Textures::ID id, const std::string &filepath);
    sf::Texture& get(Textures::ID id);
    const sf::Texture& get(Textures::ID id) const;

private:
    std::map<Textures::ID,
        std::unique_ptr<sf::Texture >> TextureMap;
};
```

1) Texture::ID enums become replaced with the Identifier. 2) Resource types (sound, font, etc), become replaced with the Resource

I found it largely simplier to large given these substitutions.

```
#pragma once
#include "Game.h"
#include <assert.h>
#include <memory>

namespace Textures
{
    enum ID { Landscape, Airplane, Missile };
}

template <typename Resource, typename Identifier>
class ResourceHolder
{

public:
    ResourceHolder            ();
    ~ResourceHolder            ();
    void load                (Identifier id, const std::string &filepath);
    Resource& get            (Identifier id);
    const Resource&            get(Identifier id) const;

private:
    std::map<Identifier,
        std::unique_ptr<Resource >> TextureMap;
```

```
};
```

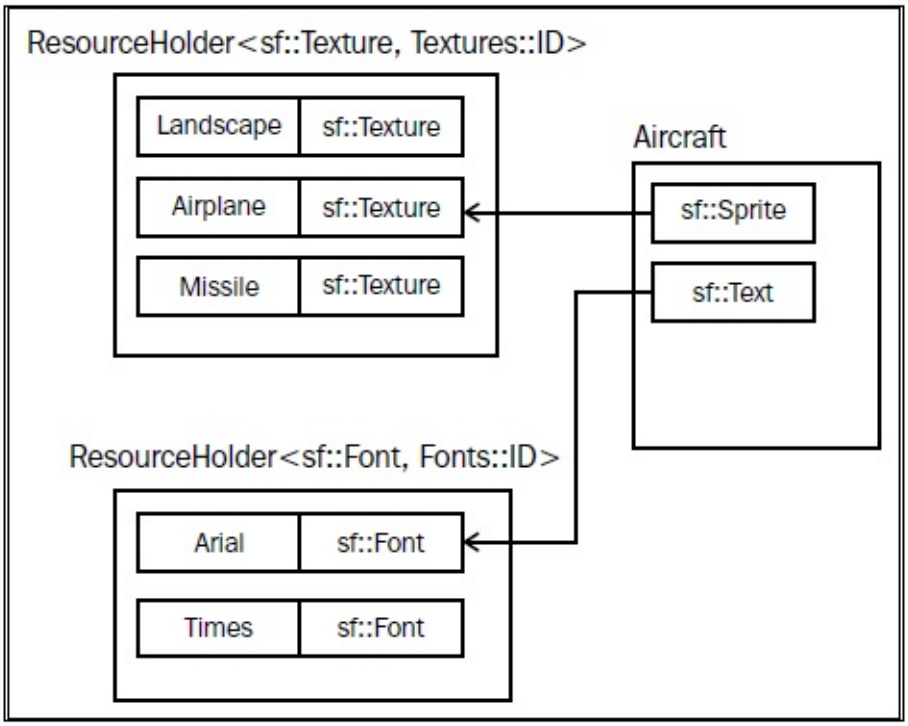
```
#include "ResourceHolder.h"


template <typename Resource, typename Identifier>
void ResourceHolder<Resource, Identifier>::load(Identifier id, const std::string &filepath)
{
    std::unique_ptr<Resource> resource(new Resource());
    if (!resource->loadFromFile(filepath))
        throw std::runtime_error("ResourceHolder::load - Failed to Load from" + filepath);

    auto inserted = TextureMap.insert(std::make_pair(id, std::move(resource)));
    assert(inserted.second);
}

template <typename Resource, typename Identifier>
Resource& ResourceHolder::get(Identifier id)
{
    auto found = TextureMap.find(id);
    return *found->second;
}

template <typename Resource, typename Identifier>
const Resource& ResourceHolder::get(Identifier id) const
{
    auto found = TextureMap.find(id);
    assert(found != TextureMap.end());
    return *found->second;
}
```

Next, in order to generalize our resource holder with shaders, a class overload is nessessary to accommodate both vertex and fragment shaders.

```
    template <typename Parameter>
    void load(Identifier id, const std::string &filepath,
    const Parameter& secondParam);
```

```
template <typename Resource, typename Identifier>
template <typename Parameter>
void ResourceHolder<Resource, Identifier>::load(Identifier id, const std::string &filepath, const Parameter& secondPara
{
    std::unique_ptr<Resource> resource(new Resource());
    if (!resource->loadFromFile(filepath, secondParam))
        throw std::runtime_error("ResourceHolder::load - Failed to Load from" + filepath);

    auto inserted = TextureMap.insert(std::make_pair(id, std::move(resource)));
    assert(inserted.second);
}
```

This gives us the option and possibility of using the second parameter.



sf::Music unfortunately is not compatable with this template, since it calls openFromFile(). But what we have done is provide compability with fonts, sound buffers, and textures.

Entities

- Are what shape and forge the world we create:
  - Missles
  - Upgrades
  - Pickups
  - Abilities
  - Commonality: they all move (have a velocity).
  -

Lets begin with our entity class.

```
#pragma once
#include "Game.h"

class Entity
{
public:
    void                setVelocity(sf::Vector2f velocity);
    sf::Vector2f        getVelocity() const;
    Entity();
    ~Entity();

private:
    sf::Vector2f        velocity;
};
```

```
#include "Entity.h"


Entity::Entity()
{
}


Entity::~Entity()
{
}

void Entity::setVelocity(sf::Vector2f vel)
{
    velocity = vel;
}

sf::Vector2f Entity::getVelocity() const
{
    return velocity;
}
```

Within our class, we can have the classes of several entities. Aircraft, for example. Which again, can be enumerated by type.

```
class Aircraft : public Entity
{
public:
    enum Type
    {
        Eagle,
        Raptor,
    };

public:
```

```
    explicit        Aircraft(Type type);

private:
    Type            mtype;
};
```

We use the explicit keyword to make sure our constructor call uses a Type type aircraft enumeration.

```
Aircraft::Aircraft(Type type) : mtype(type)
{

}
```

Shaders

- Operates directly on the graphics card.
- Can be used to simulate things like hot flickering air, or blur effects.
- Builds upon OPENGL.