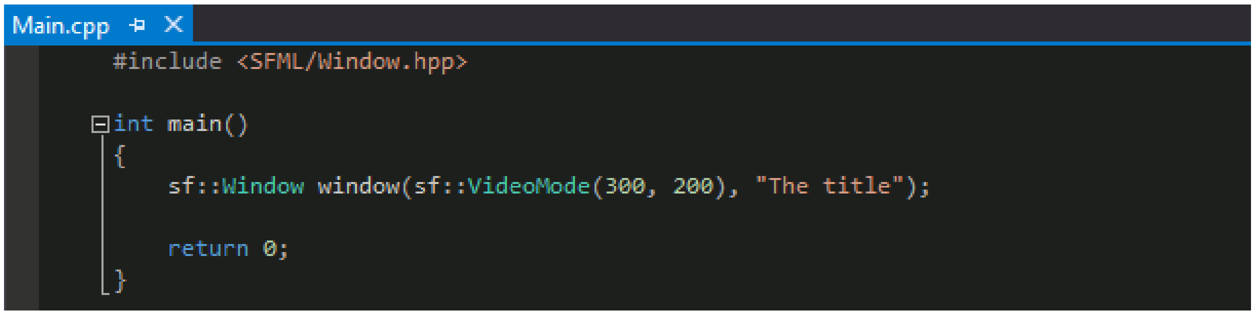
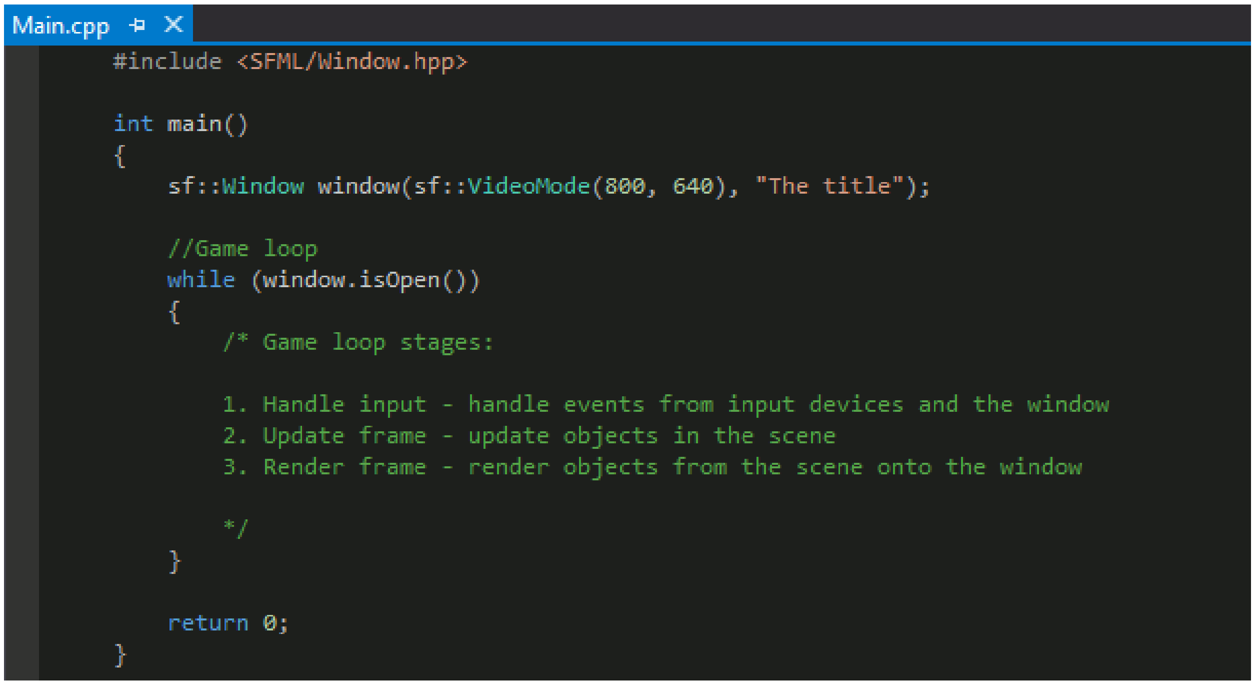
SFML Essentials Graphics Bundle

Chapter 1: Getting Started with SFML



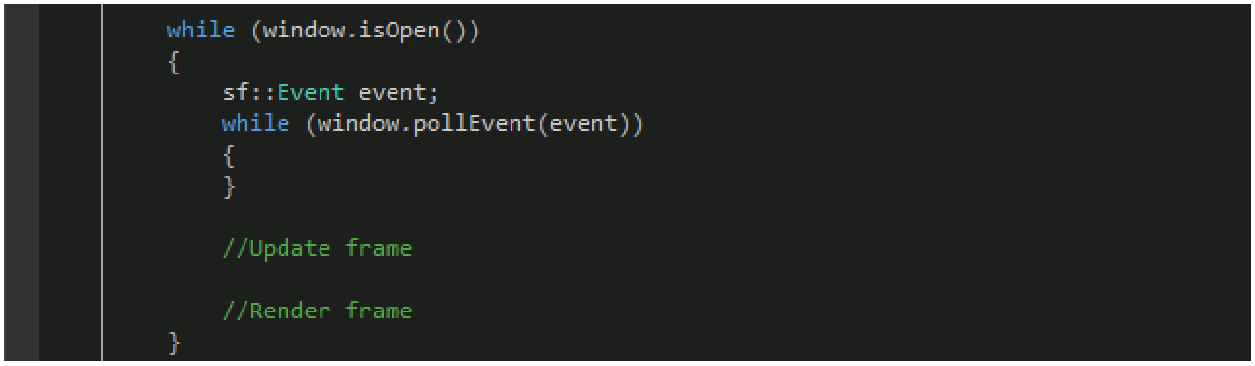
Main.cpp

```cpp
#include <SFML/Window.hpp>

int main()
{
    sf::Window window(sf::VideoMode(300, 200), "The title");

    return 0;
}
```

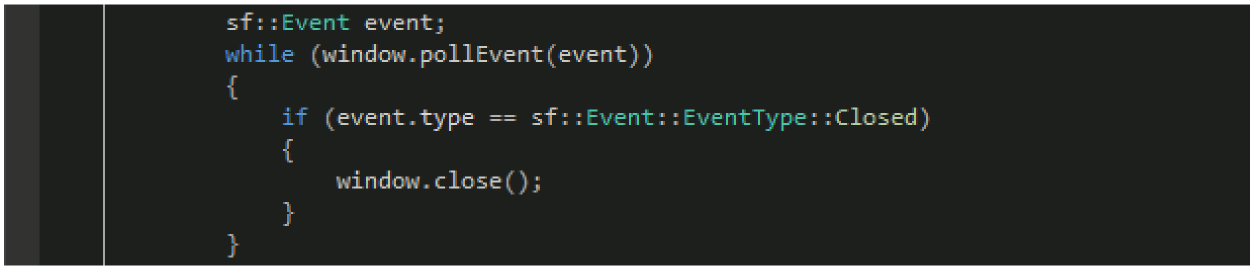


Main.cpp

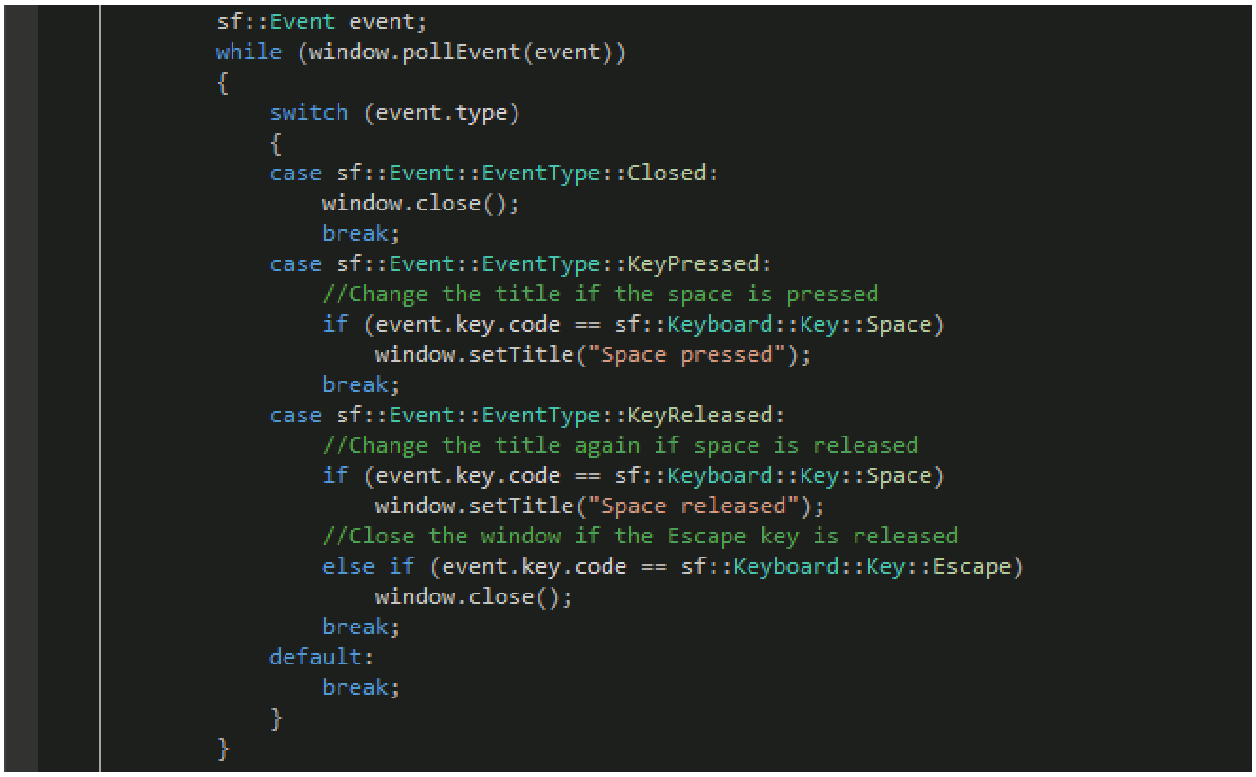
```cpp
#include <SFML/Window.hpp>

int main()
{
    sf::Window window(sf::VideoMode(800, 640), "The title");

    //Game loop
    while (window.isOpen())
    {
        /* Game loop stages:

        1. Handle input - handle events from input devices and the window
        2. Update frame - update objects in the scene
        3. Render frame - render objects from the scene onto the window

        */
    }

    return 0;
}
```



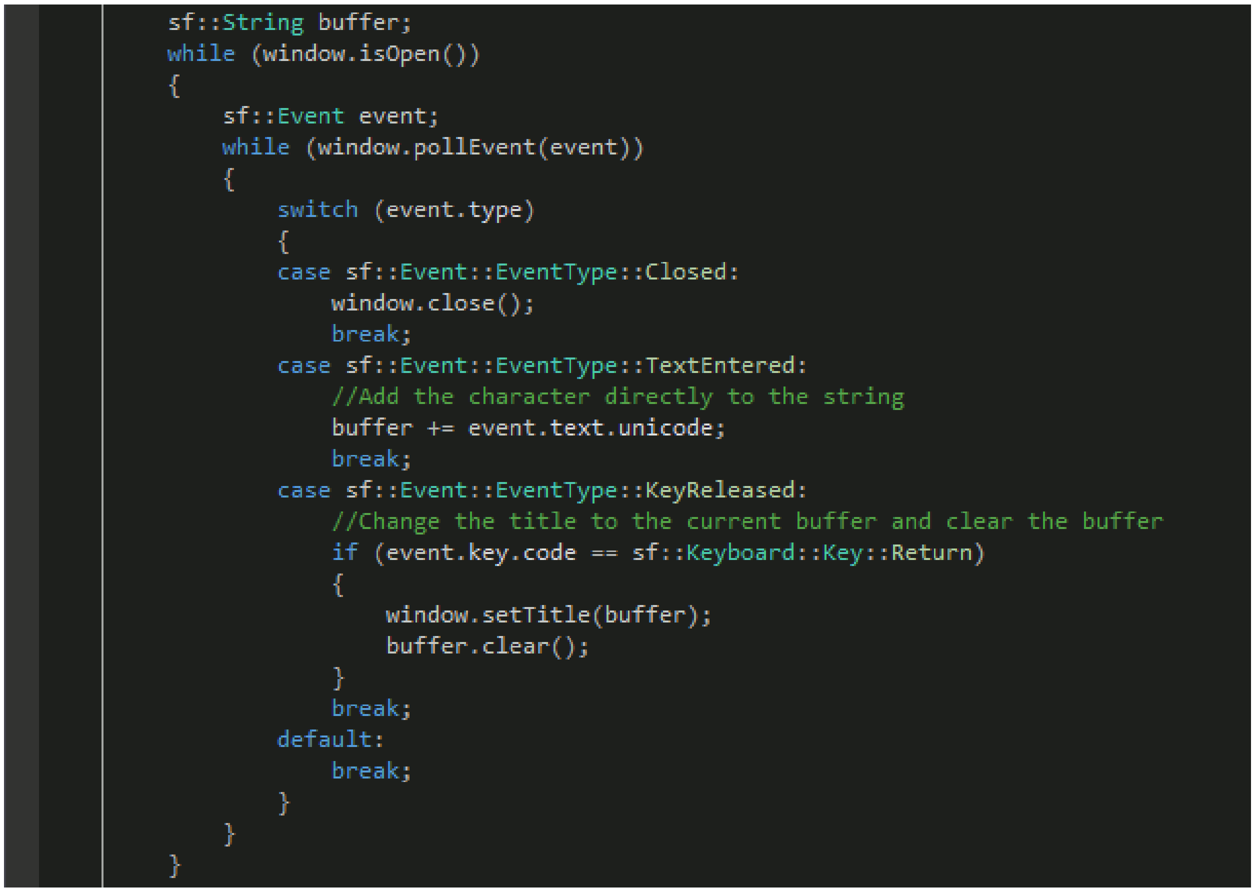
```cpp
    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
        }

        //Update frame

        //Render frame
    }
```



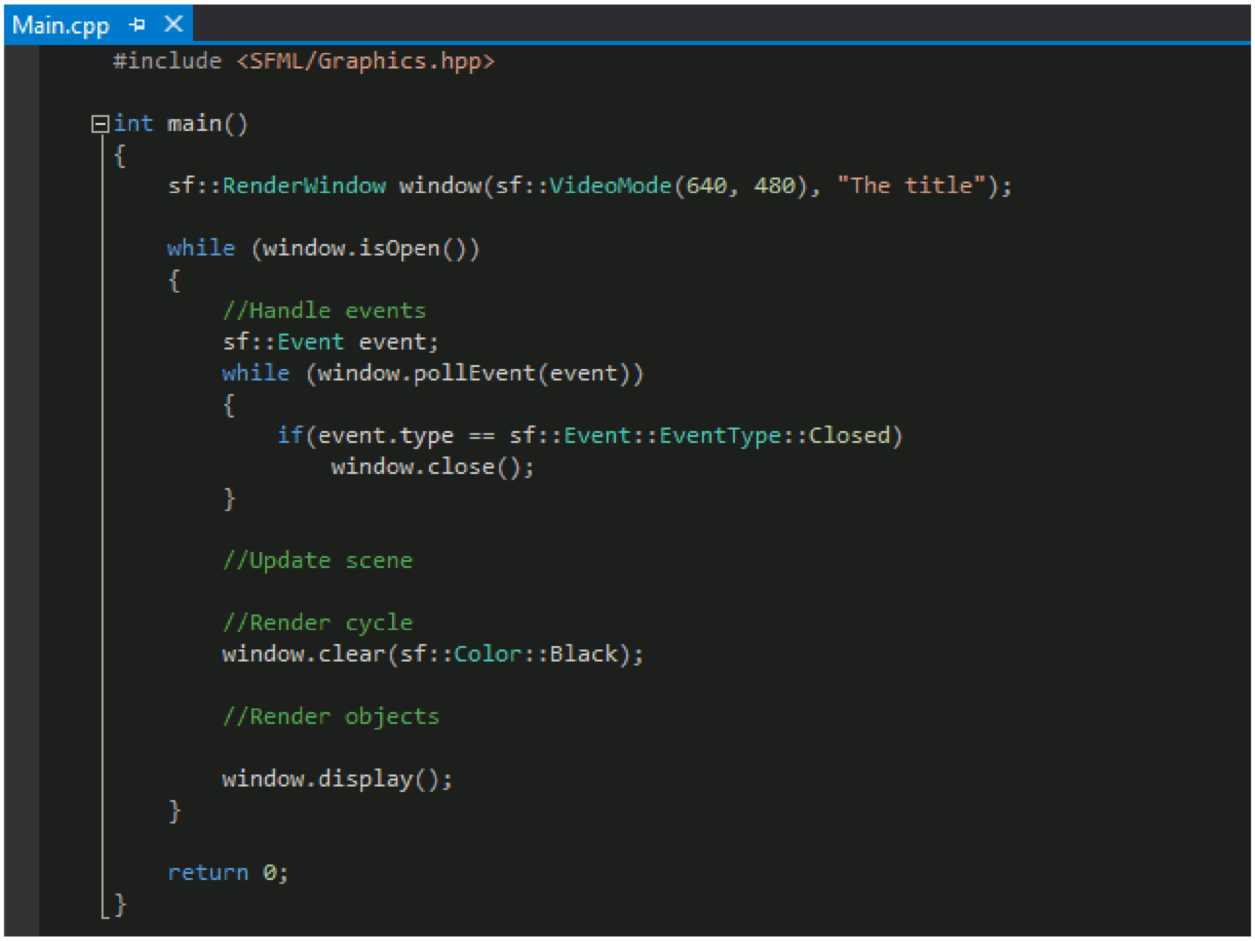
```cpp
sf::Event event;
while (window.pollEvent(event))
{
    if (event.type == sf::Event::EventType::Closed)
    {
        window.close();
    }
}
```



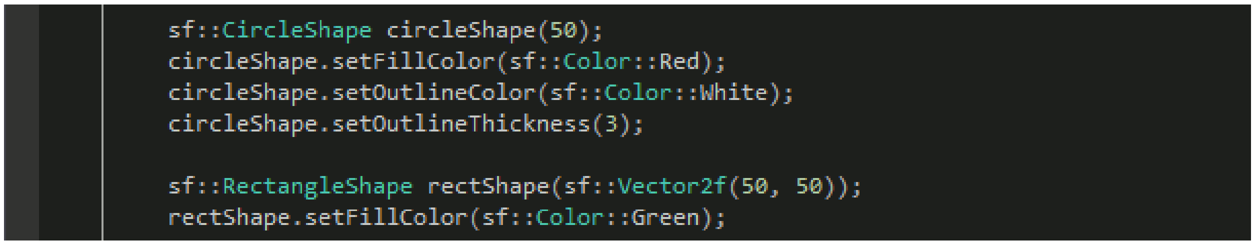
```cpp
sf::Event event;
while (window.pollEvent(event))
{
    switch (event.type)
    {
    case sf::Event::EventType::Closed:
        window.close();
        break;
    case sf::Event::EventType::KeyPressed:
        //Change the title if the space is pressed
        if (event.key.code == sf::Keyboard::Key::Space)
            window.setTitle("Space pressed");
        break;
    case sf::Event::EventType::KeyReleased:
        //Change the title again if space is released
        if (event.key.code == sf::Keyboard::Key::Space)
            window.setTitle("Space released");
        //Close the window if the Escape key is released
        else if (event.key.code == sf::Keyboard::Key::Escape)
            window.close();
        break;
    default:
        break;
    }
}
```



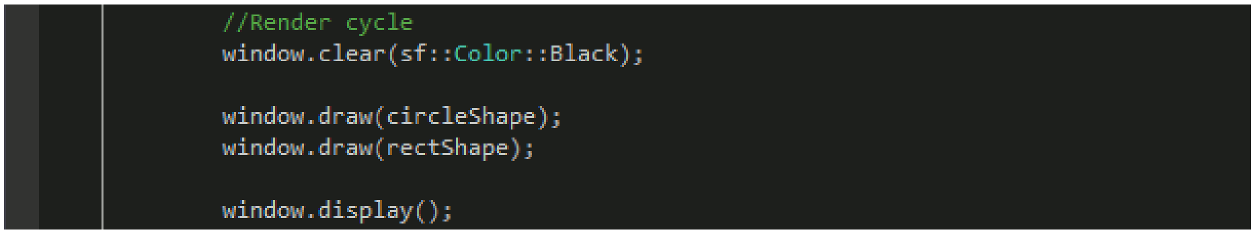
```cpp
sf::String buffer;
while (window.isOpen())
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        switch (event.type)
        {
        case sf::Event::EventType::Closed:
            window.close();
            break;
        case sf::Event::EventType::TextEntered:
            //Add the character directly to the string
            buffer += event.text.unicode;
            break;
        case sf::Event::EventType::KeyReleased:
            //Change the title to the current buffer and clear the buffer
            if (event.key.code == sf::Keyboard::Key::Return)
            {
                window.setTitle(buffer);
                buffer.clear();
            }
            break;
        default:
            break;
        }
    }
}
```


Main.cpp

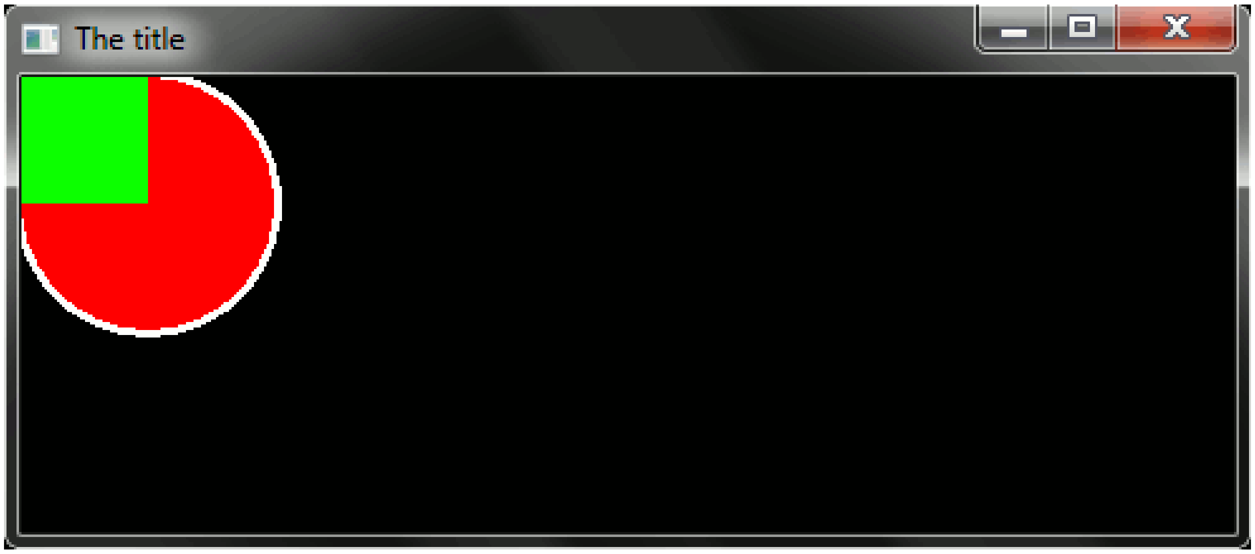
```cpp
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(640, 480), "The title");

    while (window.isOpen())
    {
        //Handle events
        sf::Event event;
        while (window.pollEvent(event))
        {
            if(event.type == sf::Event::EventType::Closed)
                window.close();
        }

        //Update scene

        //Render cycle
        window.clear(sf::Color::Black);

        //Render objects

        window.display();
    }

    return 0;
}
```



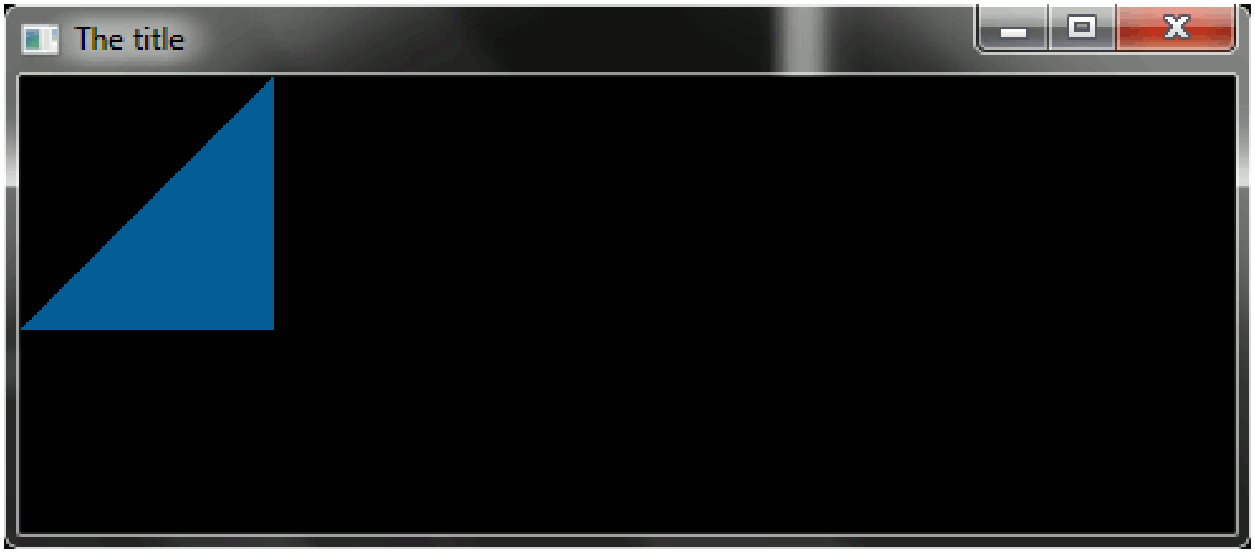
```cpp
    sf::CircleShape circleShape(50);
    circleShape.setFillColor(sf::Color::Red);
    circleShape.setOutlineColor(sf::Color::White);
    circleShape.setOutlineThickness(3);

    sf::RectangleShape rectShape(sf::Vector2f(50, 50));
    rectShape.setFillColor(sf::Color::Green);
```



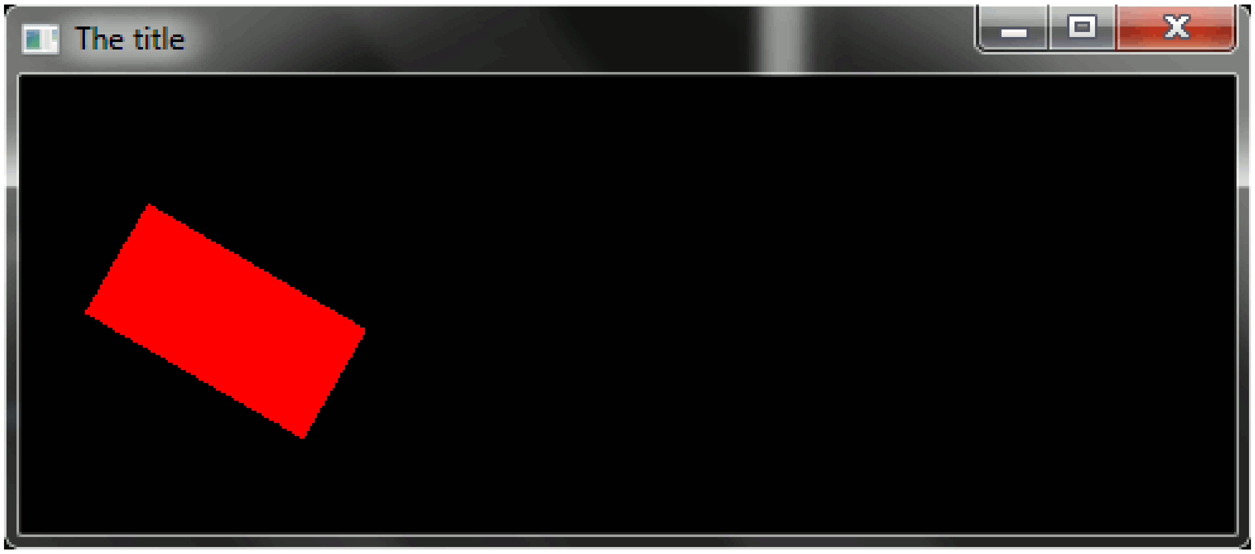
```cpp
        //Render cycle
        window.clear(sf::Color::Black);

        window.draw(circleShape);
        window.draw(rectShape);

        window.display();
```

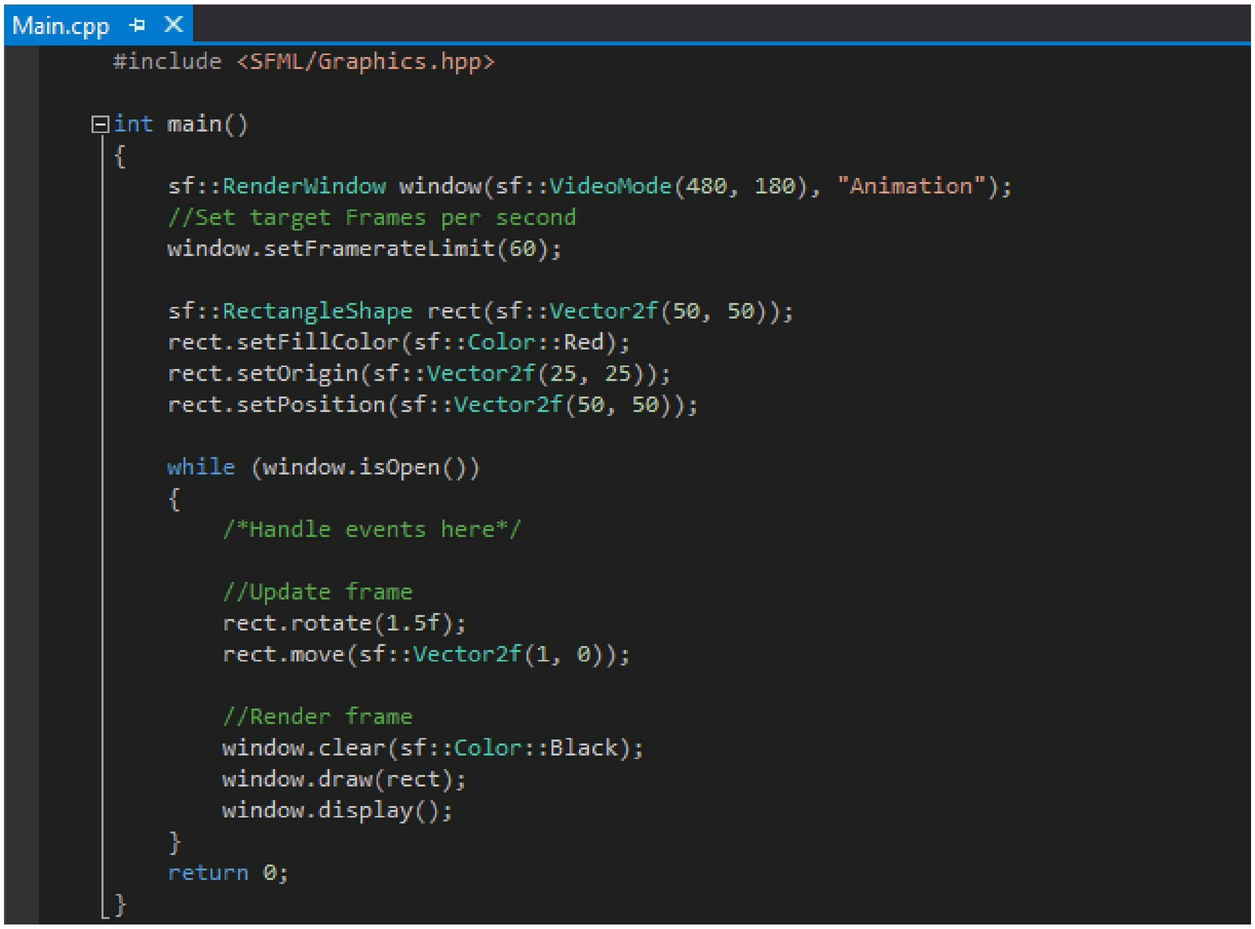


```
sf::ConvexShape triangle;
triangle.setPointCount(3);
triangle.setPoint(0, sf::Vector2f(100, 0));
triangle.setPoint(1, sf::Vector2f(100, 100));
triangle.setPoint(2, sf::Vector2f(0, 100));
triangle.setFillColor(sf::Color::Blue);
```

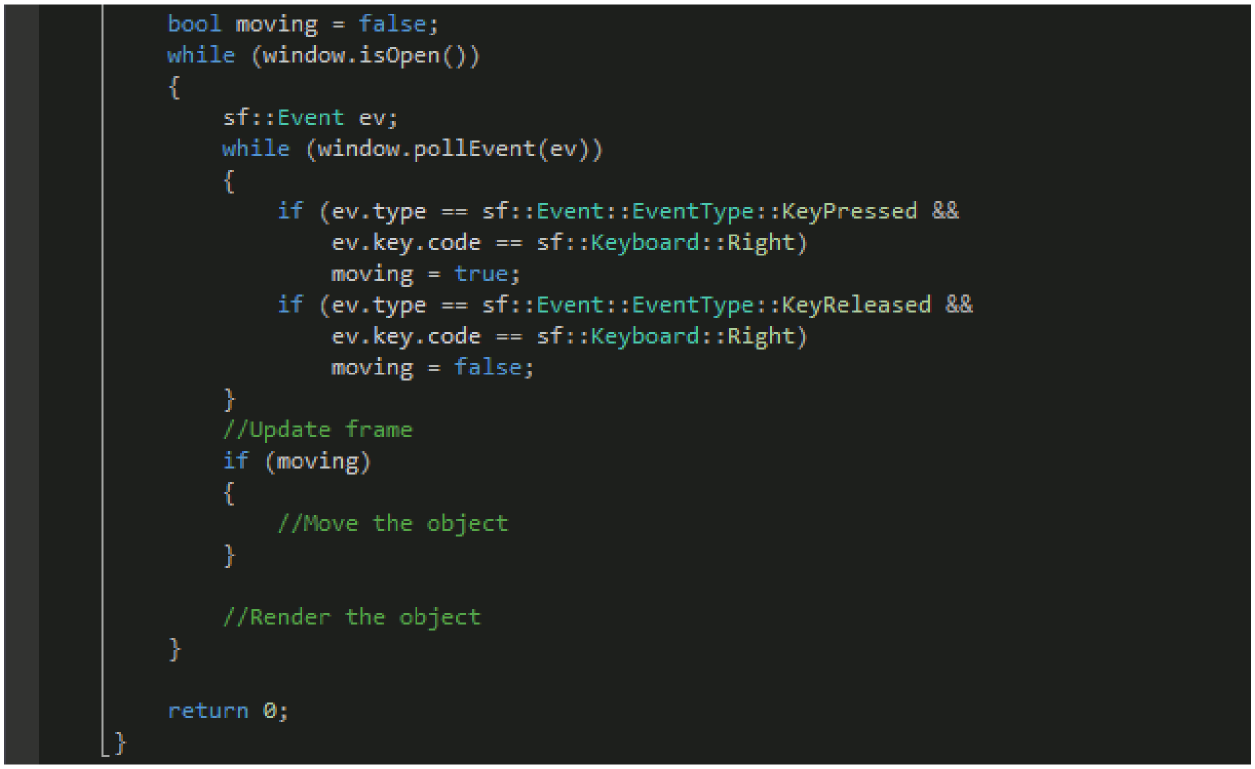


```
sf::RectangleShape rect(sf::Vector2f(50, 50));
rect.setFillColor(sf::Color::Red);
rect.setPosition(sf::Vector2f(50, 50));
rect.setRotation(30);
rect.setScale(sf::Vector2f(2, 1));
```

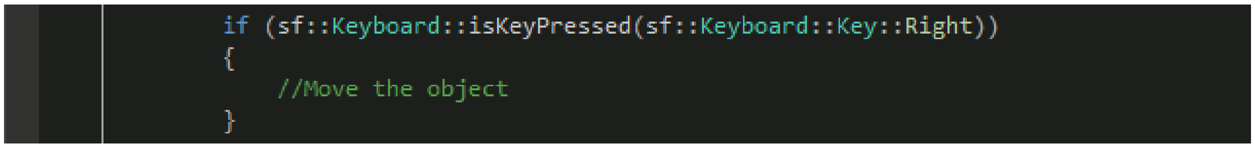




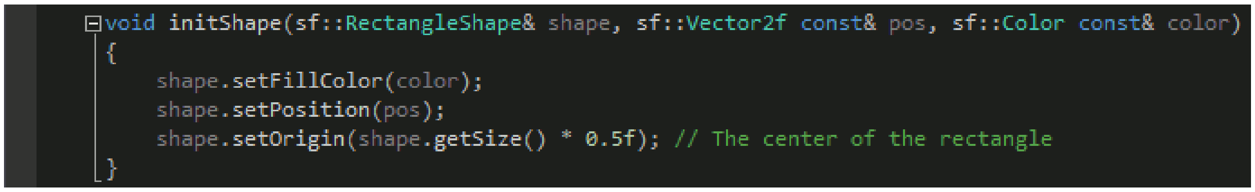
```cpp
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(480, 180), "Animation");
    //Set target Frames per second
    window.setFramerateLimit(60);

    sf::RectangleShape rect(sf::Vector2f(50, 50));
    rect.setFillColor(sf::Color::Red);
    rect.setOrigin(sf::Vector2f(25, 25));
    rect.setPosition(sf::Vector2f(50, 50));

    while (window.isOpen())
    {
        /*Handle events here*/

        //Update frame
        rect.rotate(1.5f);
        rect.move(sf::Vector2f(1, 0));

        //Render frame
        window.clear(sf::Color::Black);
        window.draw(rect);
        window.display();
    }
    return 0;
}
```



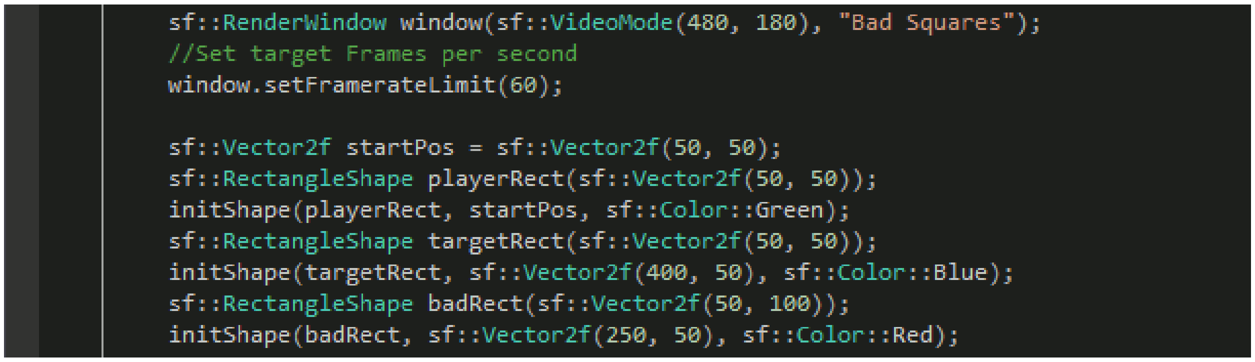
```cpp
    bool moving = false;
    while (window.isOpen())
    {
        sf::Event ev;
        while (window.pollEvent(ev))
        {
            if (ev.type == sf::Event::EventType::KeyPressed &&
                ev.key.code == sf::Keyboard::Right)
                moving = true;
            if (ev.type == sf::Event::EventType::KeyReleased &&
                ev.key.code == sf::Keyboard::Right)
                moving = false;
        }
        //Update frame
        if (moving)
        {
            //Move the object
        }

        //Render the object
    }

    return 0;
}
```



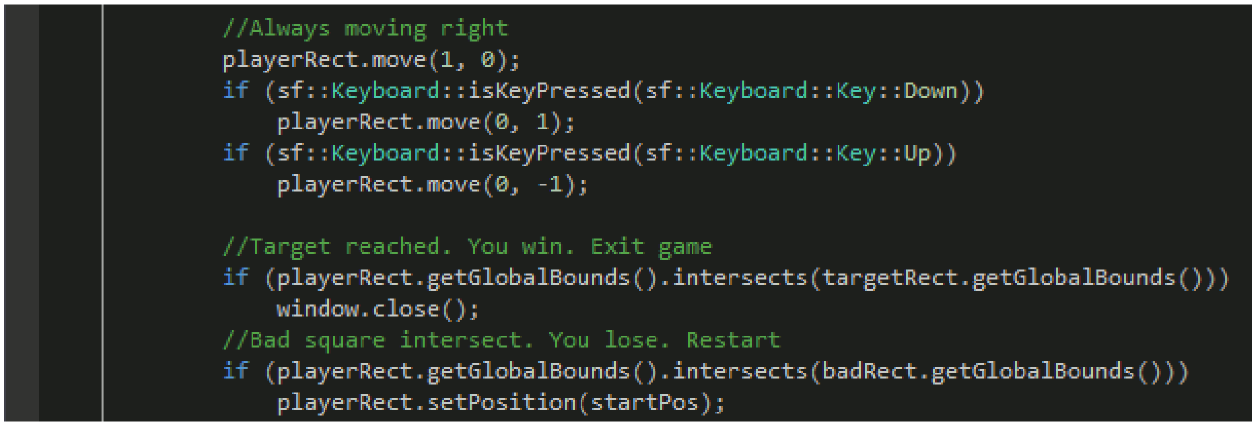
```cpp
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Right))
        {
            //Move the object
        }
```



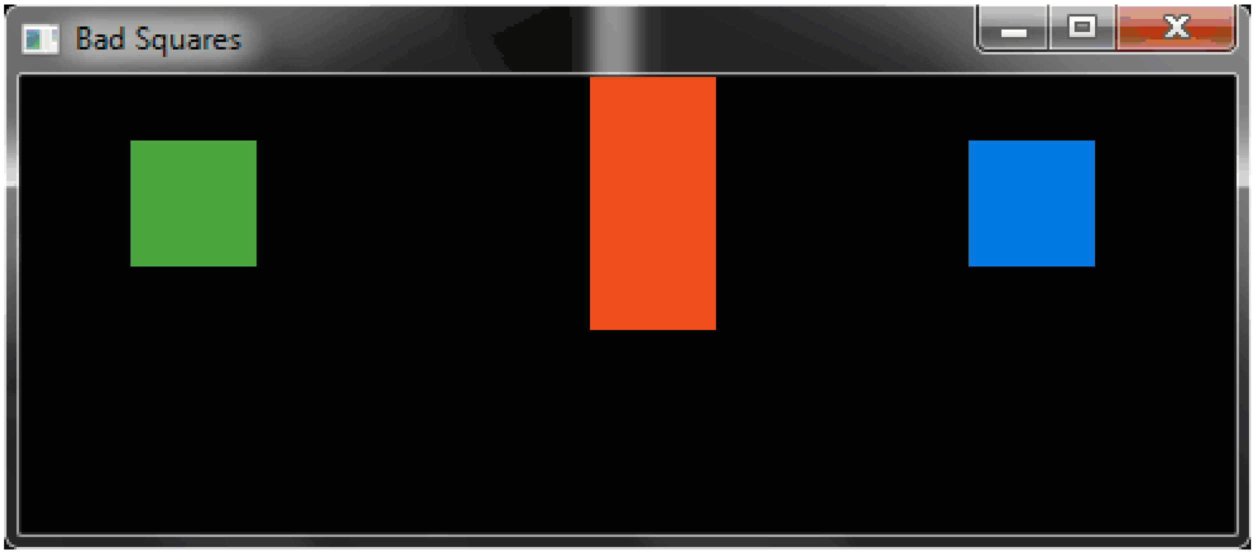
```cpp
void initShape(sf::RectangleShape& shape, sf::Vector2f const& pos, sf::Color const& color)
{
    shape.setFillColor(color);
    shape.setPosition(pos);
    shape.setOrigin(shape.getSize() * 0.5f); // The center of the rectangle
}
```



```cpp
    sf::RenderWindow window(sf::VideoMode(480, 180), "Bad Squares");
    //Set target Frames per second
    window.setFramerateLimit(60);

    sf::Vector2f startPos = sf::Vector2f(50, 50);
    sf::RectangleShape playerRect(sf::Vector2f(50, 50));
    initShape(playerRect, startPos, sf::Color::Green);
    sf::RectangleShape targetRect(sf::Vector2f(50, 50));
    initShape(targetRect, sf::Vector2f(400, 50), sf::Color::Blue);
    sf::RectangleShape badRect(sf::Vector2f(50, 100));
    initShape(badRect, sf::Vector2f(250, 50), sf::Color::Red);
```



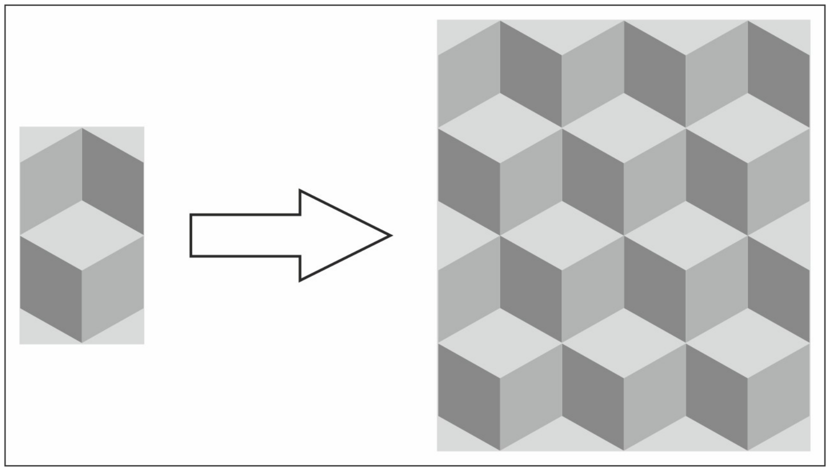
```
//Always moving right
playerRect.move(1, 0);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Down))
    playerRect.move(0, 1);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Up))
    playerRect.move(0, -1);

//Target reached. You win. Exit game
if (playerRect.getGlobalBounds().intersects(targetRect.getGlobalBounds()))
    window.close();
//Bad square intersect. You lose. Restart
if (playerRect.getGlobalBounds().intersects(badRect.getGlobalBounds()))
    playerRect.setPosition(startPos);
```



Chapter 2: Loading and Using Textures

```cpp
sf::Image image;
image.create(50, 50, sf::Color::Red);
```

```cpp
const unsigned int kWidth = 5, kHeight = 5;

//Array size = width * height * 4(RGBA)
sf::Uint8 pixels[kWidth * kHeight * 4] =
{
    255, 255, 255, 255, //White
    0, 0, 0, 255,       //Black
    255, 0, 0, 255,     //Red
    128, 128, 128, 255,  //Gray

    //...all other pixels
};

sf::Image image;
image.create(kWidth, kHeight, pixels);
```



```cpp
sf::Image image;
image.loadFromFile("myImage.png");
```

```
Failed to load image "myImage.png". Reason : Unable to open file
```

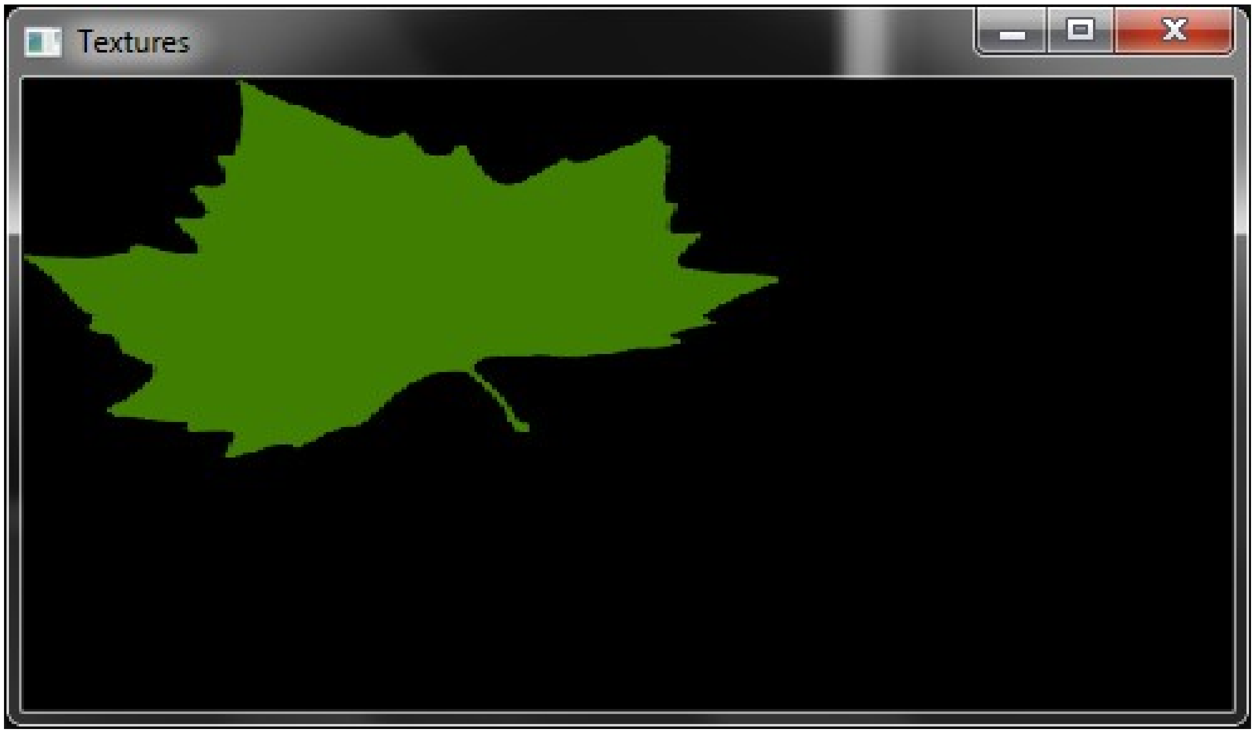
```cpp
sf::Image image;
if (!image.loadFromFile("myImage.png"))
    return -1;
```

```cpp
sf::Texture texture;
if (!texture.loadFromFile("myTexture.png"))
    return -1;
```

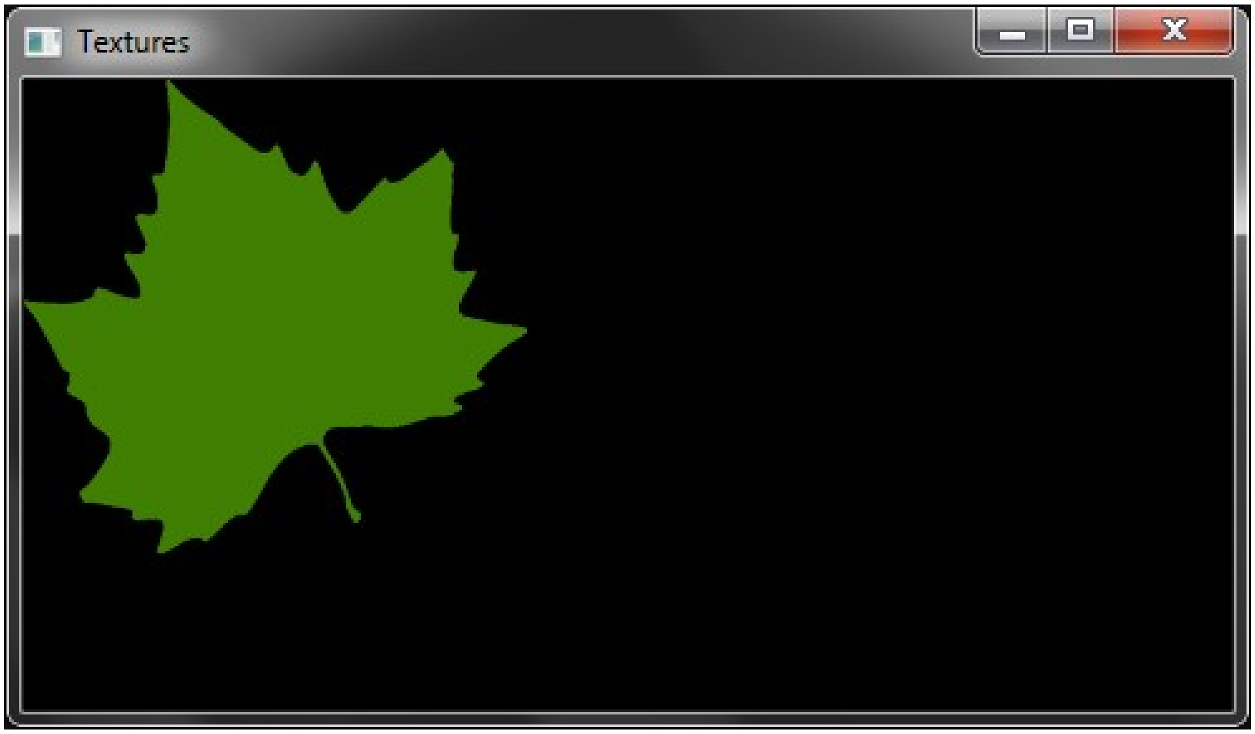
```cpp
sf::Texture texture;
if (!texture.loadFromFile("myTexture.png", sf::IntRect(0, 0, 32, 32)))
    return -1;
```

```cpp
sf::Image image;
image.create(50, 50, sf::Color::Red);

sf::Texture texture;
texture.loadFromImage(image);
```

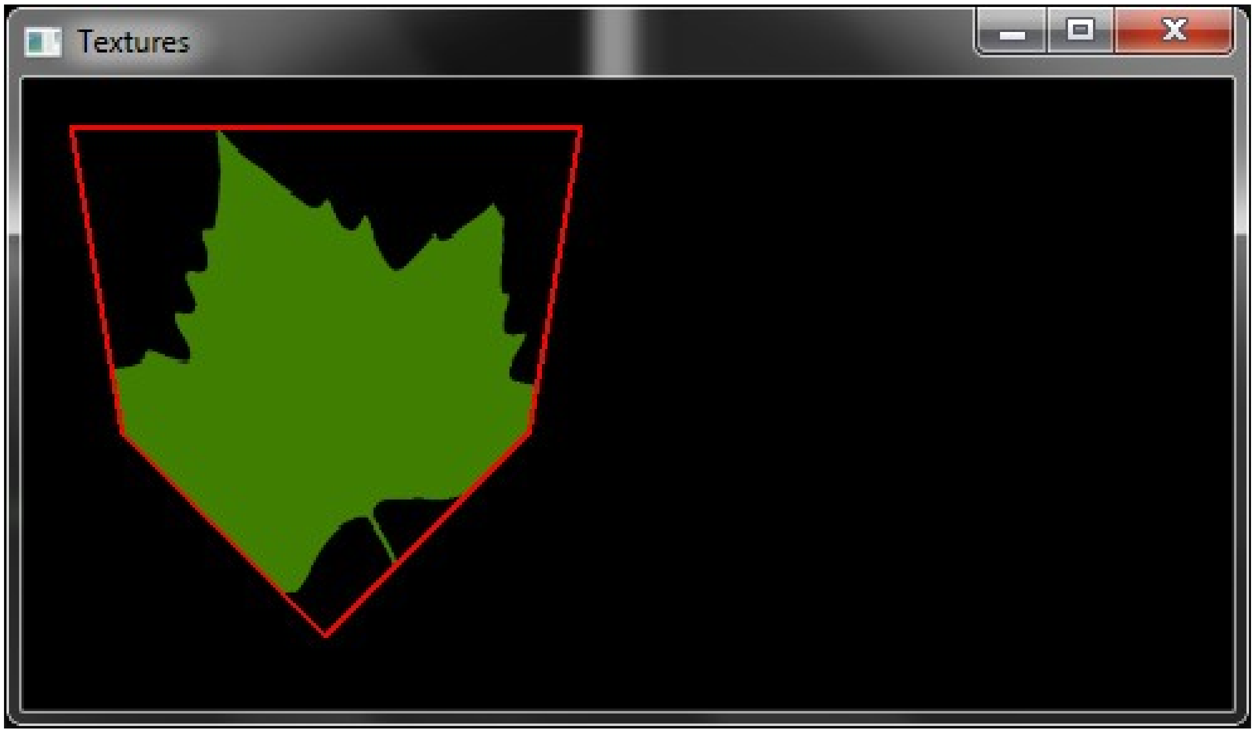
```cpp
sf::Texture texture;
texture.loadFromFile("myTexture.png");

sf::RectangleShape rectShape(sf::Vector2f(300, 150));
rectShape.setTexture(&texture);

while (window.isOpen())
{
    //Handle events

    window.clear(sf::Color::Black);
    window.draw(rectShape);
    window.display();
}
```

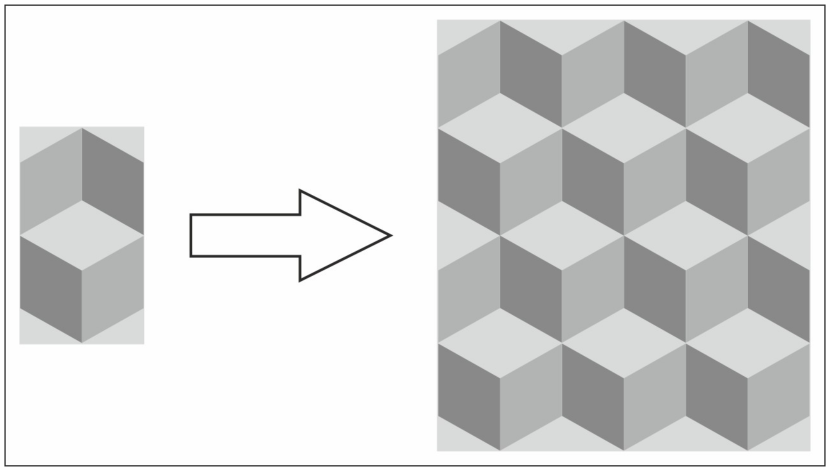


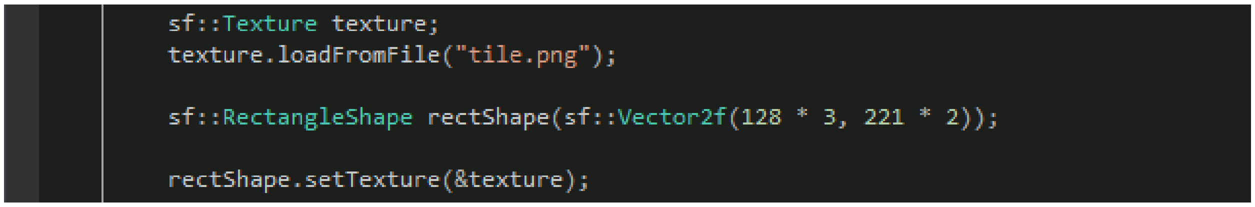
```cpp
sf::Vector2u textureSize = texture.getSize();
float rectWidth = static_cast<float>(textureSize.x);
float rectHeight = static_cast<float>(textureSize.y);
sf::RectangleShape rectShape(sf::Vector2f(rectWidth, rectHeight));
rectShape.setTexture(&texture);
```

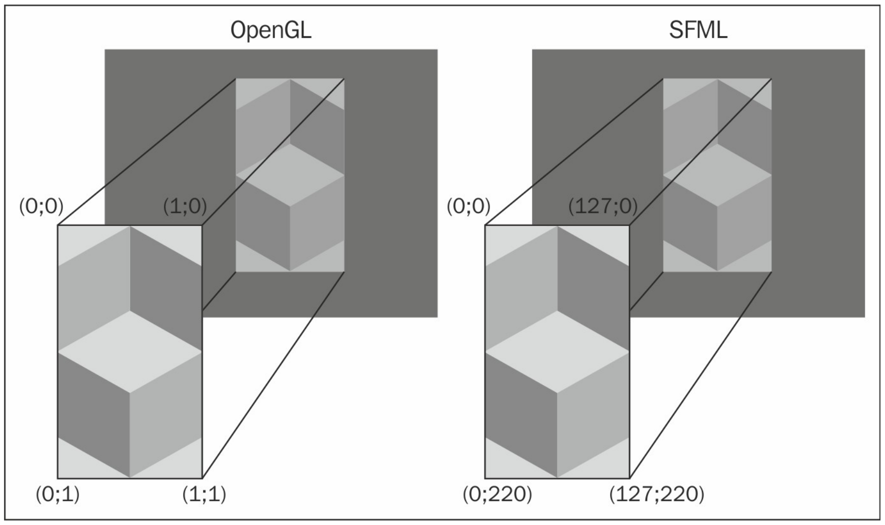


```cpp
sf::ConvexShape shape(5); //Convex shape has 5 points
shape.setPoint(0, sf::Vector2f(0, 0));
shape.setPoint(1, sf::Vector2f(200, 0));
shape.setPoint(2, sf::Vector2f(180, 120));
shape.setPoint(3, sf::Vector2f(100, 200));
shape.setPoint(4, sf::Vector2f(20, 120));
shape.setTexture(&texture);
shape.setOutlineThickness(2);
shape.setOutlineColor(sf::Color::Red);
shape.move(20, 20); //Move it, so the outline is clearly visible
```

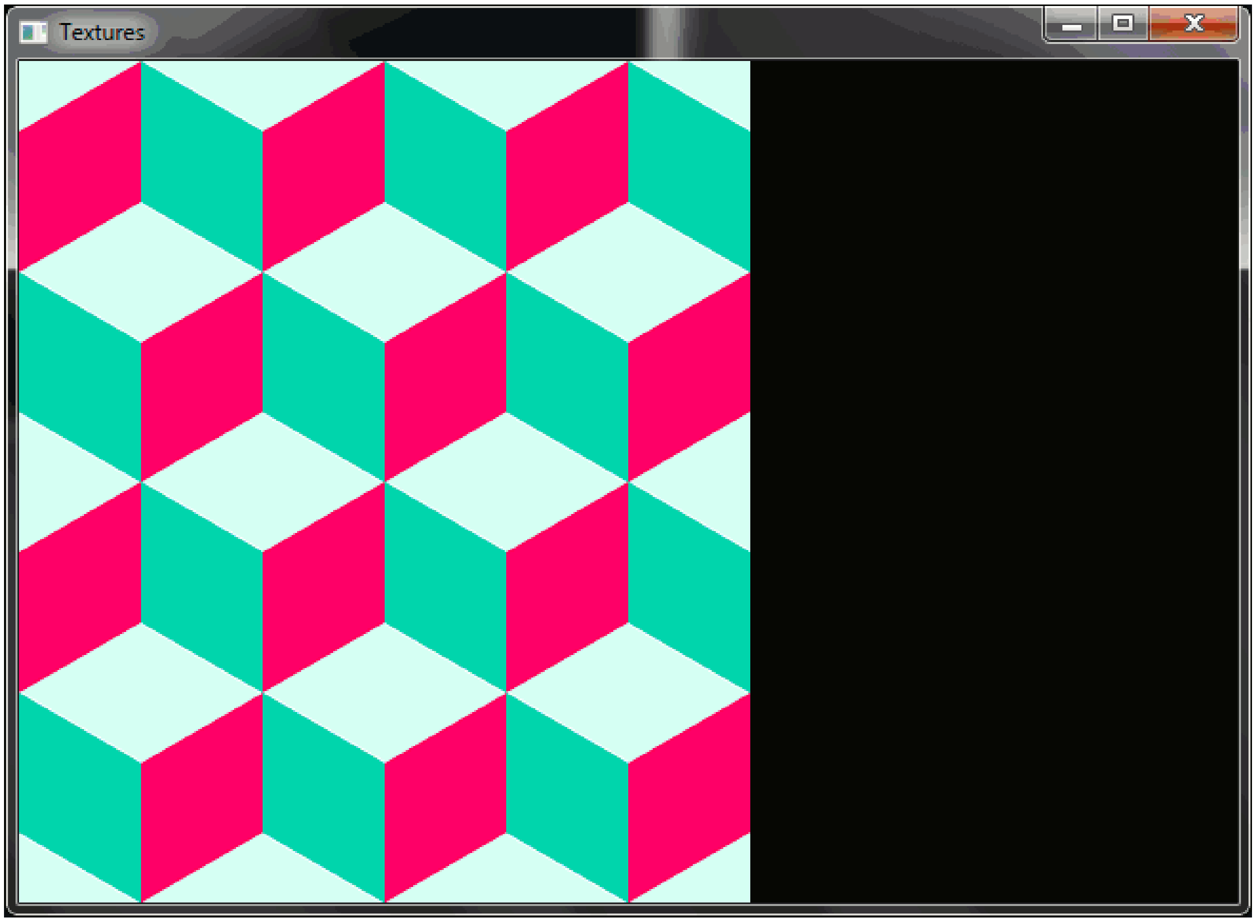


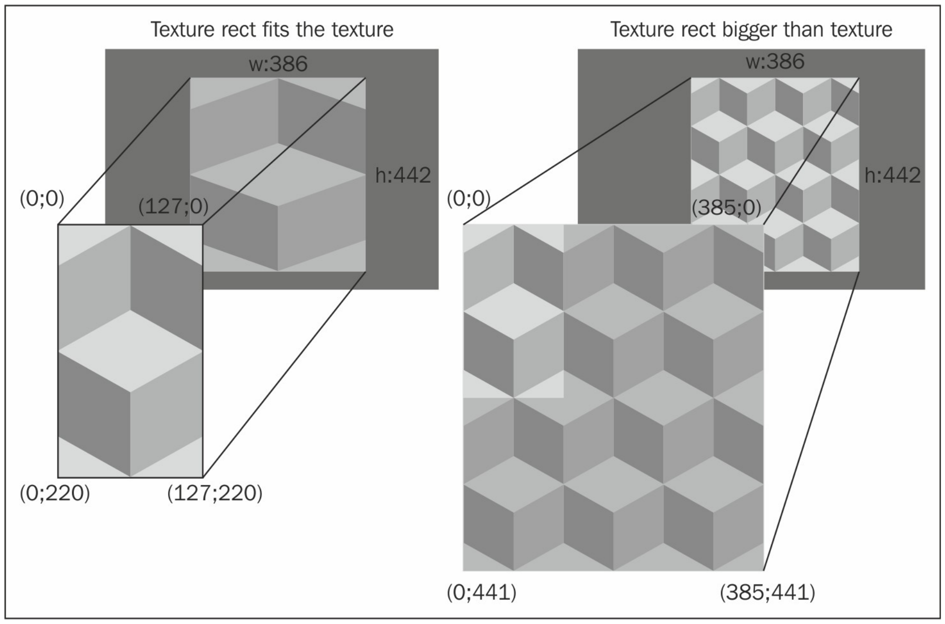



```
sf::Texture texture;
texture.loadFromFile("tile.png");

sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```

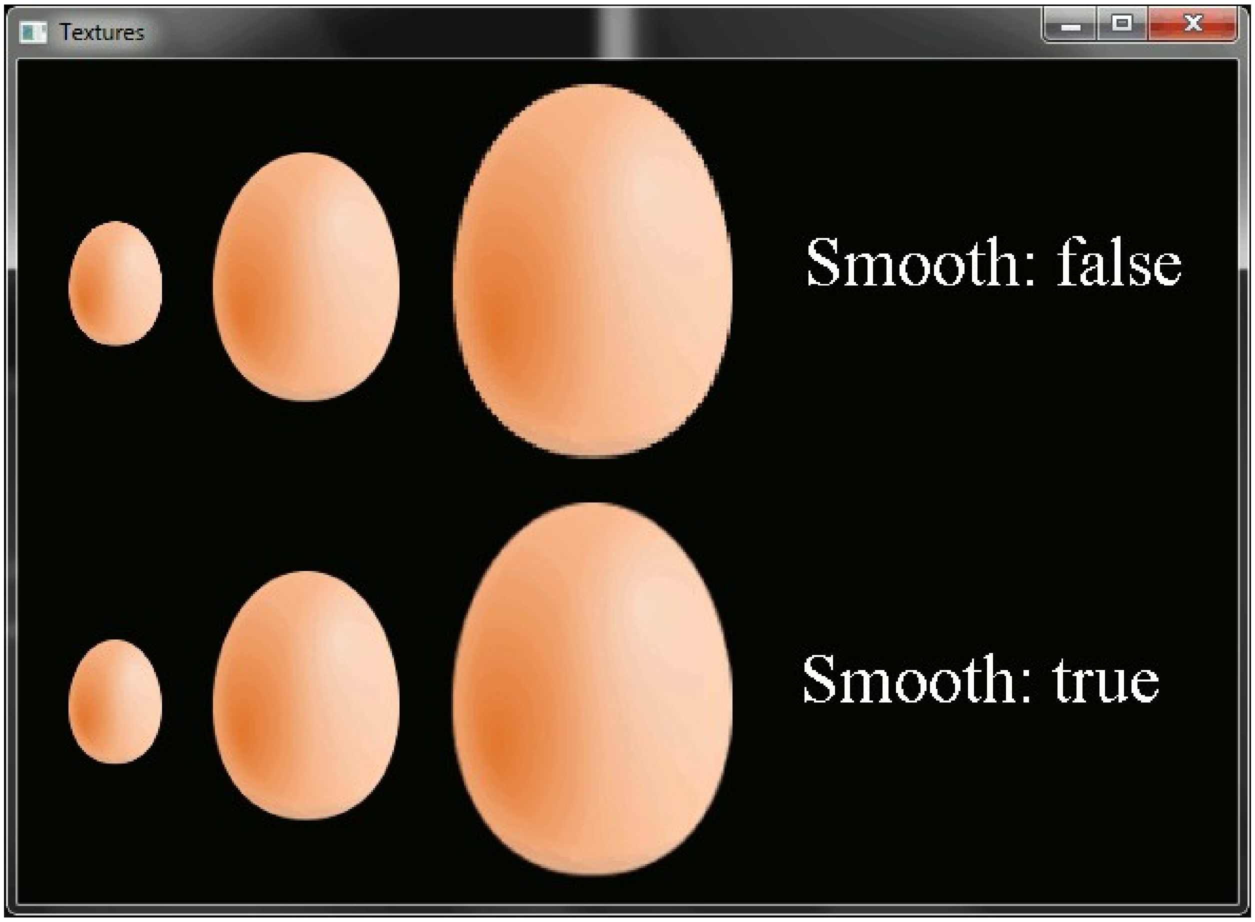


```cpp
sf::Texture texture;
texture.loadFromFile("tile.png");
//Set the texture in repeat mode
texture.setRepeated(true);

sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));
//Bigger texture rectangle than the size of the texture
rectShape.setTextureRect(sf::IntRect(0, 0, 128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```



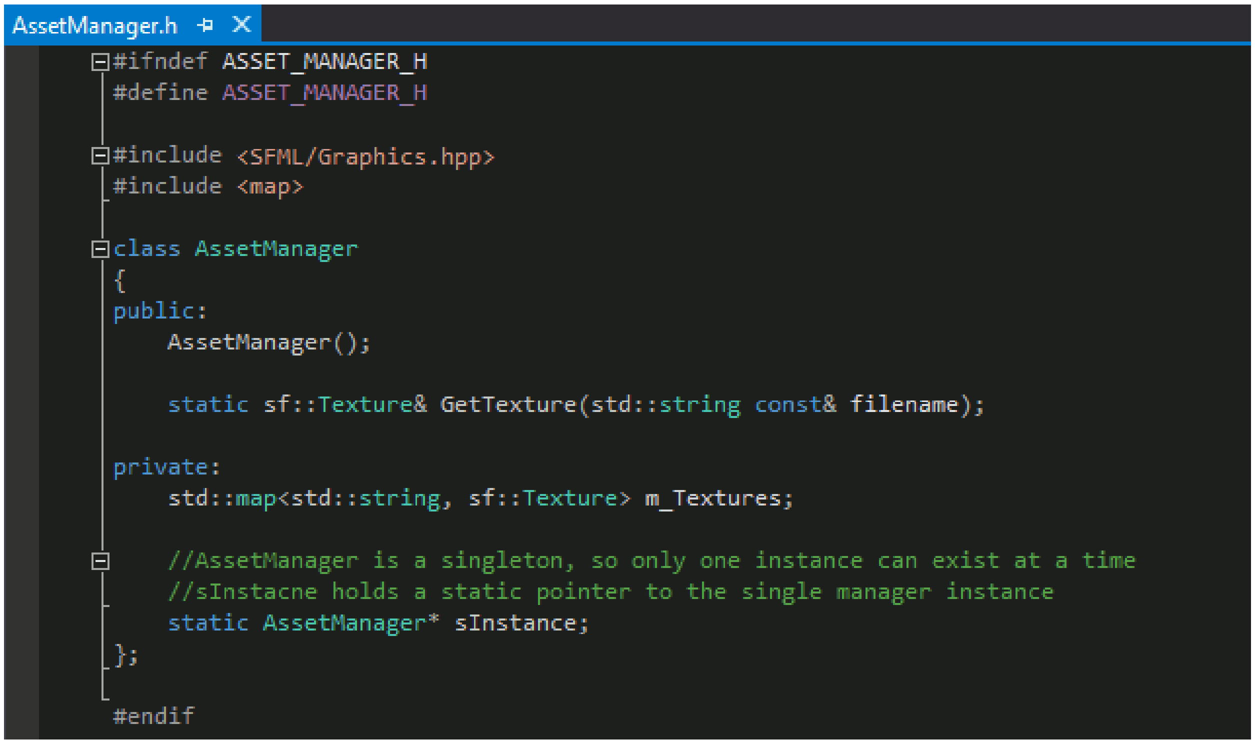


```
sf::Texture texture;
texture.loadFromFile("myTexture.png");
texture.setSmooth(true);
```

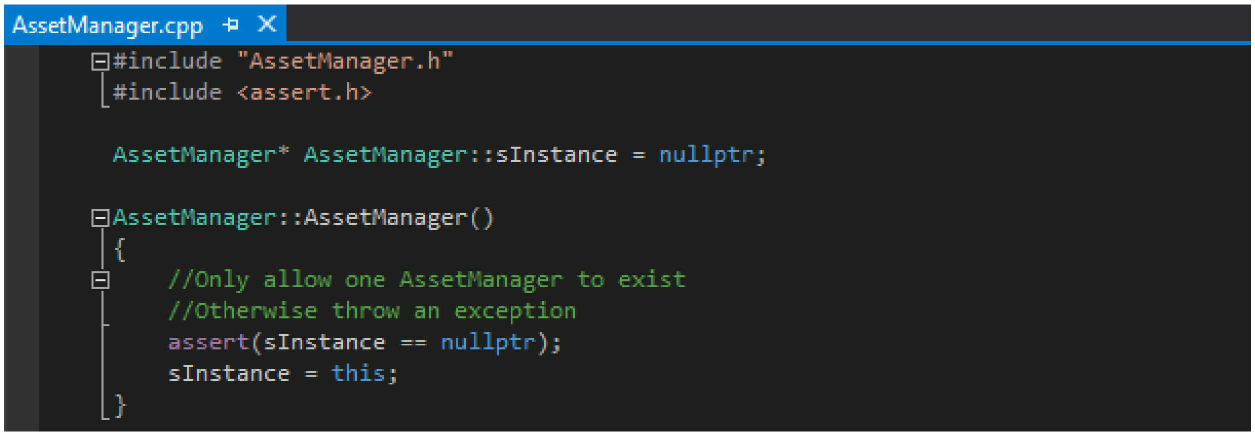


```
//Create a shape with a texture
sf::RectangleShape rectShape(sf::Vector2f(100, 100));
rectShape.setTexture(&texture);

//Create a sprite
sf::Sprite sp(texture);
```

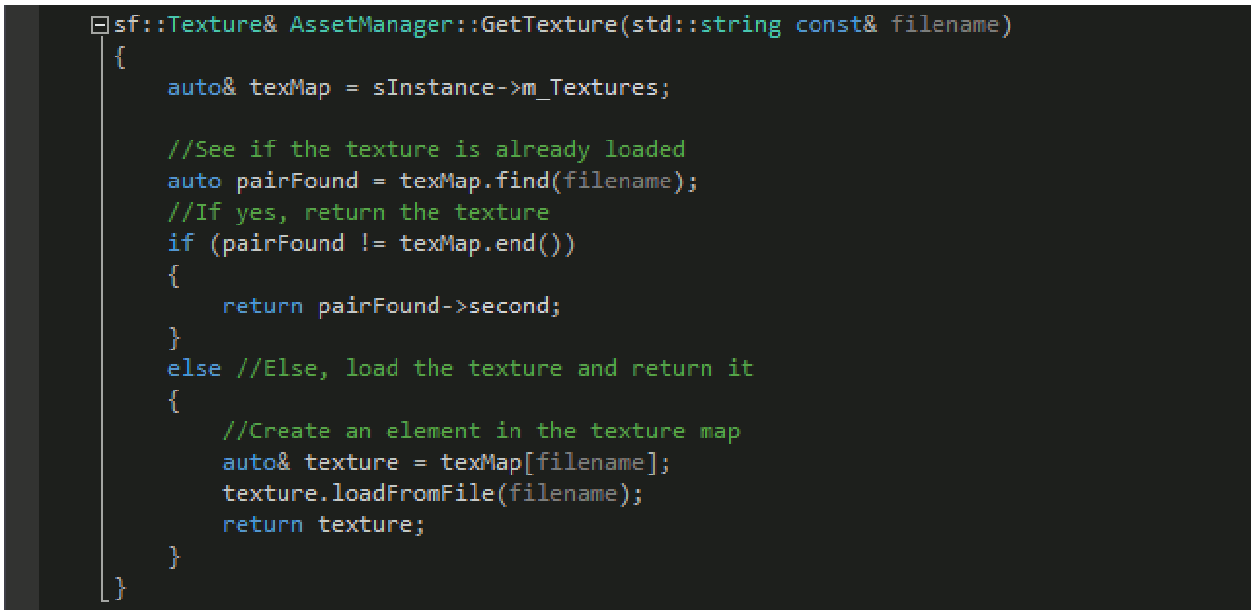
```
sf::Sprite createSprite(std::string const& filename)
{
    sf::Texture texture;
    texture.loadFromFile(filename);

    //This is bad. As soon as the function returns
    //the texture will be destroyed
    return sf::Sprite(texture);
}
```



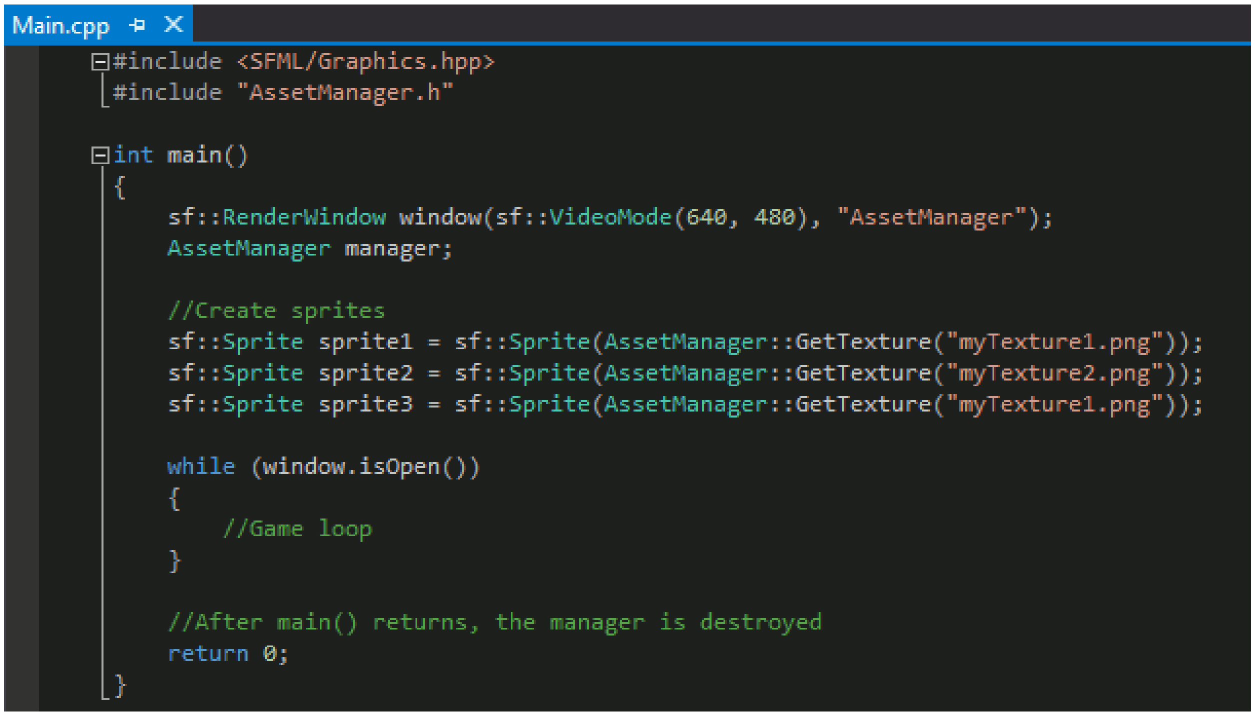
```cpp
#ifndef ASSET_MANAGER_H
#define ASSET_MANAGER_H

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

class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};

#endif
```



```cpp
#include "AssetManager.h"
#include <assert.h>

AssetManager* AssetManager::sInstance = nullptr;

AssetManager::AssetManager()
{
    //Only allow one AssetManager to exist
    //Otherwise throw an exception
    assert(sInstance == nullptr);
    sInstance = this;
}
```

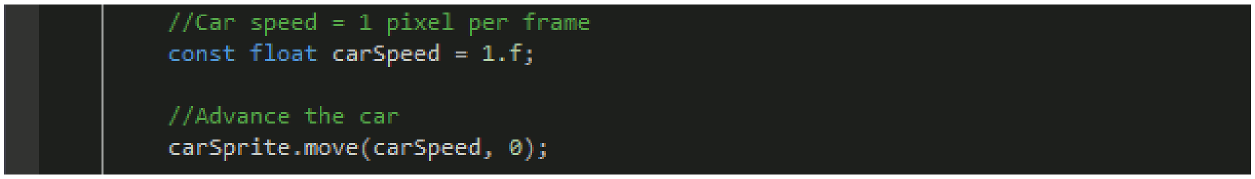


```cpp
sf::Texture& AssetManager::GetTexture(std::string const& filename)
{
    auto& texMap = sInstance->m_Textures;

    //See if the texture is already loaded
    auto pairFound = texMap.find(filename);
    //If yes, return the texture
    if (pairFound != texMap.end())
    {
        return pairFound->second;
    }
    else //Else, load the texture and return it
    {
        //Create an element in the texture map
        auto& texture = texMap[filename];
        texture.loadFromFile(filename);
        return texture;
    }
}
```

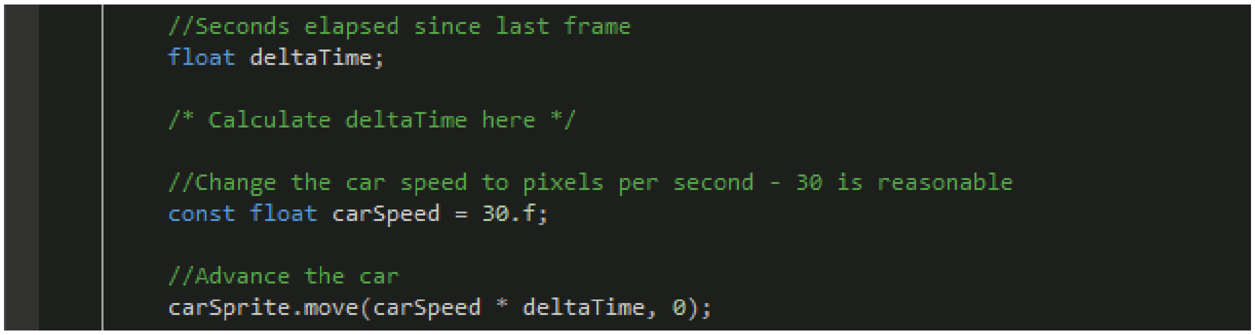


```cpp
#include <SFML/Graphics.hpp>
#include "AssetManager.h"

int main()
{
    sf::RenderWindow window(sf::VideoMode(640, 480), "AssetManager");
    AssetManager manager;

    //Create sprites
    sf::Sprite sprite1 = sf::Sprite(AssetManager::GetTexture("myTexture1.png"));
    sf::Sprite sprite2 = sf::Sprite(AssetManager::GetTexture("myTexture2.png"));
    sf::Sprite sprite3 = sf::Sprite(AssetManager::GetTexture("myTexture1.png"));

    while (window.isOpen())
    {
        //Game loop
    }

    //After main() returns, the manager is destroyed
    return 0;
}
```

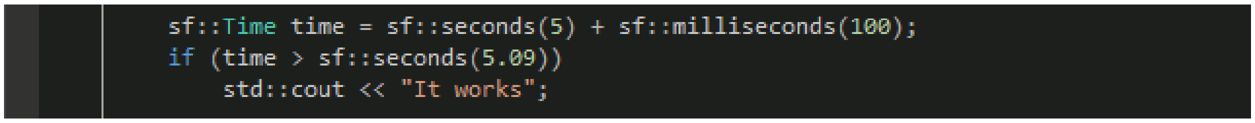
Chapter 3: Animating Sprites



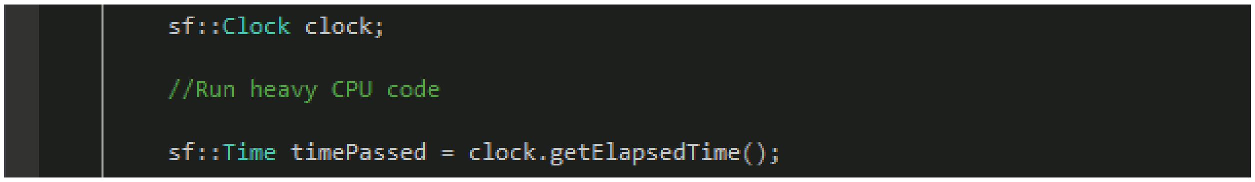
```cpp
//Car speed = 1 pixel per frame
const float carSpeed = 1.f;

//Advance the car
carSprite.move(carSpeed, 0);
```



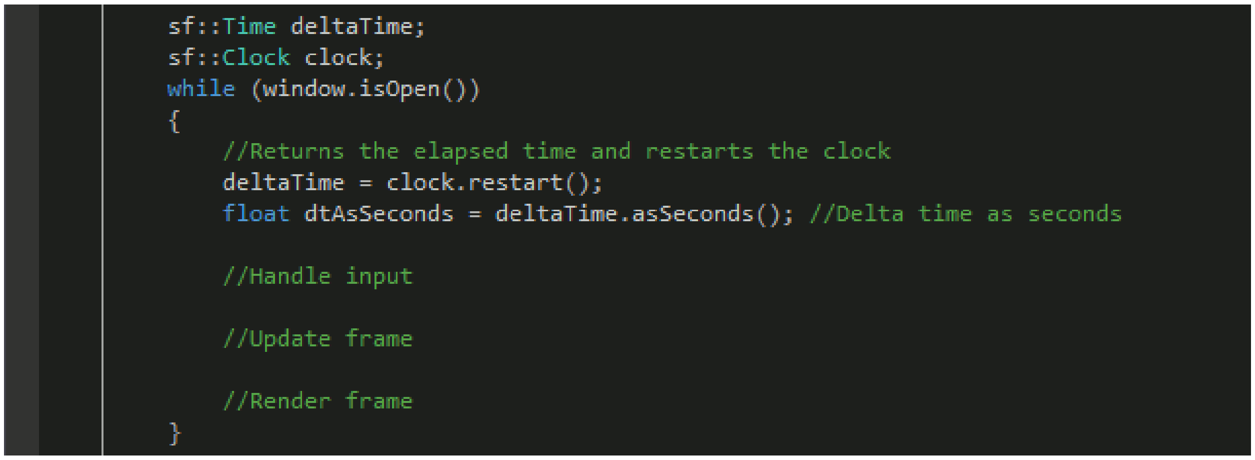
```cpp
//Seconds elapsed since last frame
float deltaTime;

/* Calculate deltaTime here */

//Change the car speed to pixels per second - 30 is reasonable
const float carSpeed = 30.f;

//Advance the car
carSprite.move(carSpeed * deltaTime, 0);
```



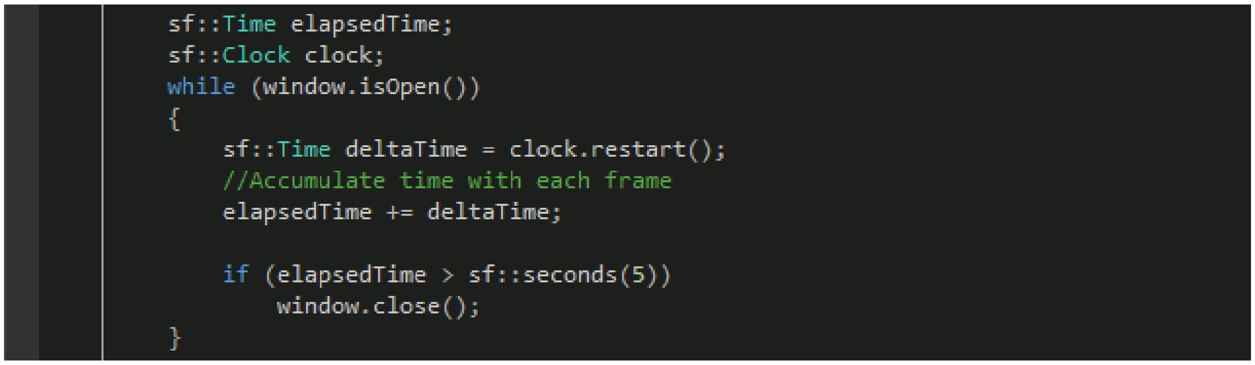
```cpp
sf::Time time = sf::seconds(5) + sf::milliseconds(100);
if (time > sf::seconds(5.09))
    std::cout << "It works";
```



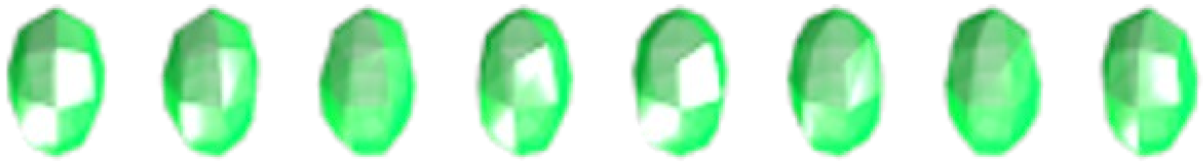
```cpp
sf::Clock clock;

//Run heavy CPU code

sf::Time timePassed = clock.getElapsedTime();
```

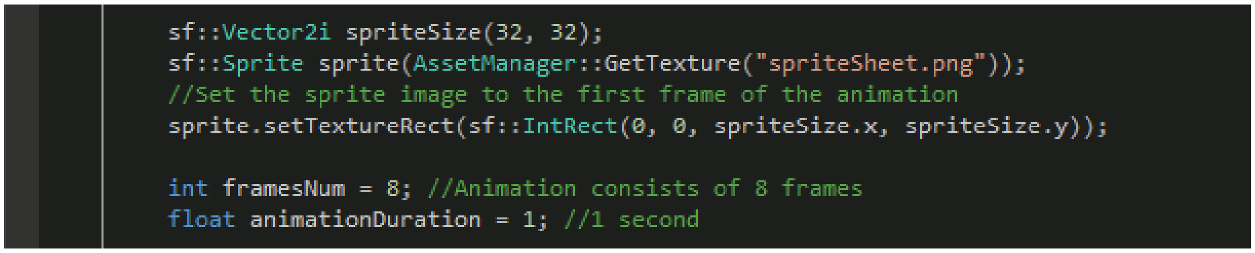


```cpp
sf::Time deltaTime;
sf::Clock clock;
while (window.isOpen())
{
    //Returns the elapsed time and restarts the clock
    deltaTime = clock.restart();
    float dtAsSeconds = deltaTime.asSeconds(); //Delta time as seconds

    //Handle input

    //Update frame

    //Render frame
}
```

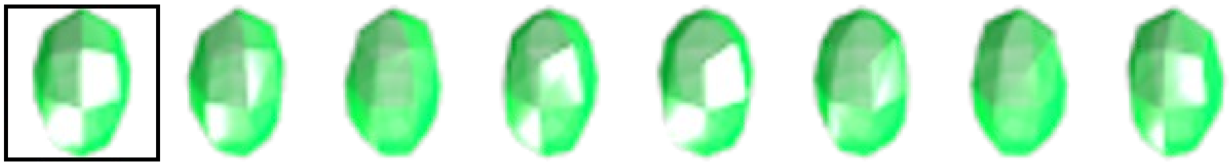


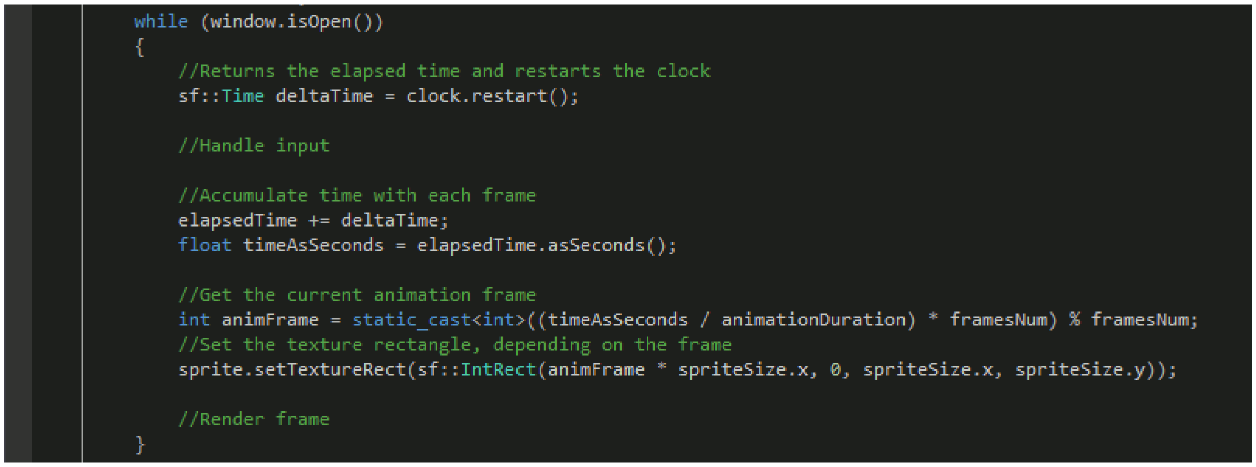
```cpp
    sf::Time elapsedTime;
    sf::Clock clock;
    while (window.isOpen())
    {
        sf::Time deltaTime = clock.restart();
        //Accumulate time with each frame
        elapsedTime += deltaTime;

        if (elapsedTime > sf::seconds(5))
            window.close();
    }
```



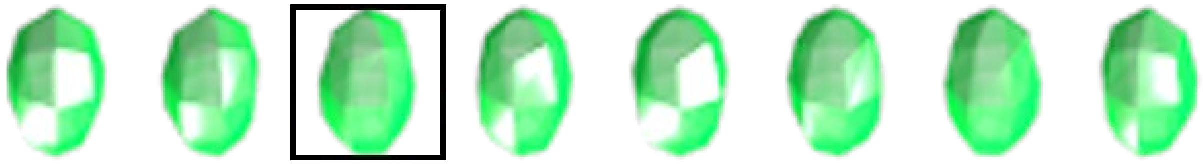


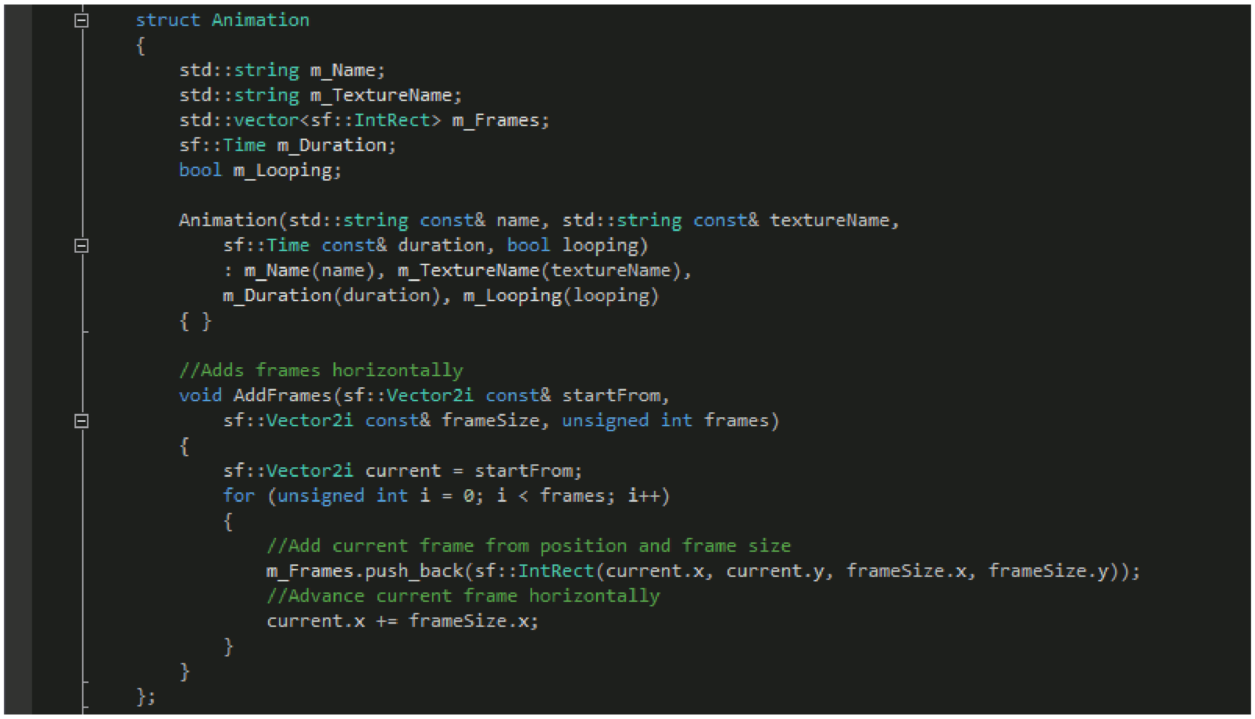
```cpp
    sf::Vector2i spriteSize(32, 32);
    sf::Sprite sprite(AssetManager::GetTexture("spriteSheet.png"));
    //Set the sprite image to the first frame of the animation
    sprite.setTextureRect(sf::IntRect(0, 0, spriteSize.x, spriteSize.y));

    int framesNum = 8; //Animation consists of 8 frames
    float animationDuration = 1; //1 second
```

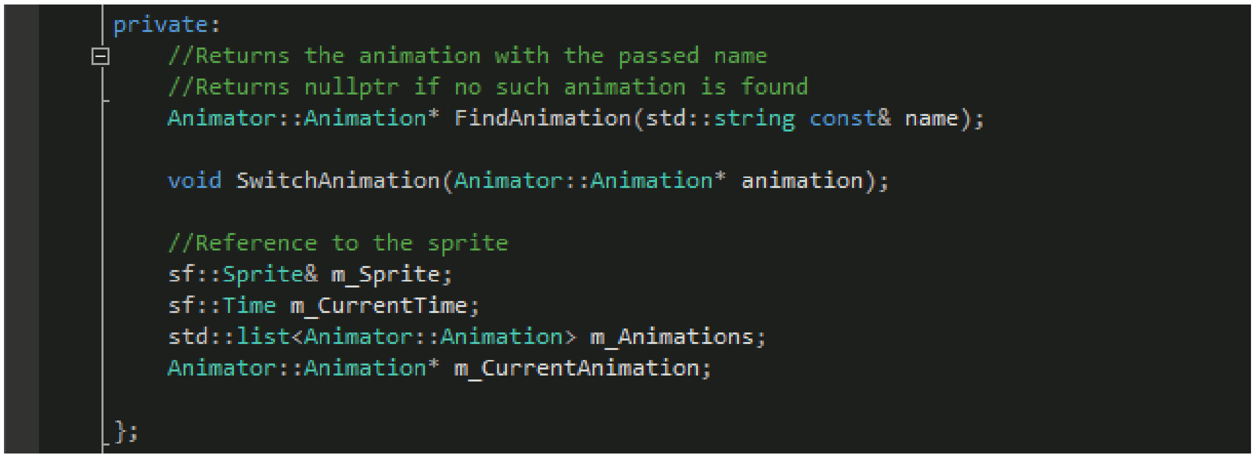


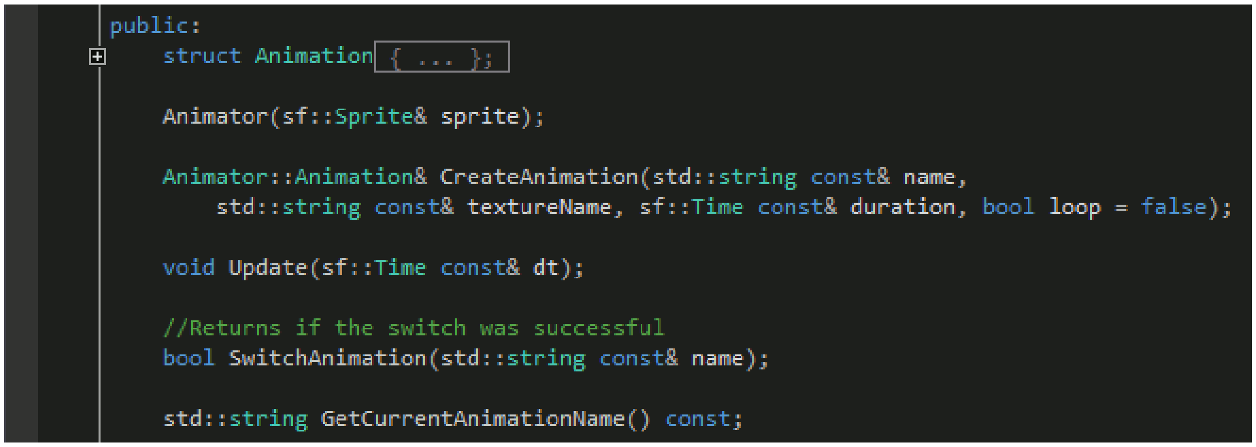


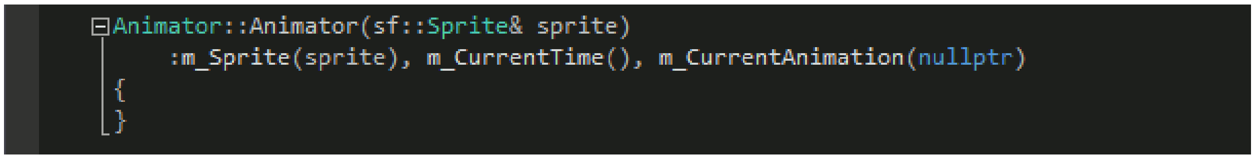
```cpp
    while (window.isOpen())
    {
        //Returns the elapsed time and restarts the clock
        sf::Time deltaTime = clock.restart();

        //Handle input

        //Accumulate time with each frame
        elapsedTime += deltaTime;
        float timeAsSeconds = elapsedTime.asSeconds();

        //Get the current animation frame
        int animFrame = static_cast<int>((timeAsSeconds / animationDuration) * framesNum) % framesNum;
        //Set the texture rectangle, depending on the frame
        sprite.setTextureRect(sf::IntRect(animFrame * spriteSize.x, 0, spriteSize.x, spriteSize.y));

        //Render frame
    }
```



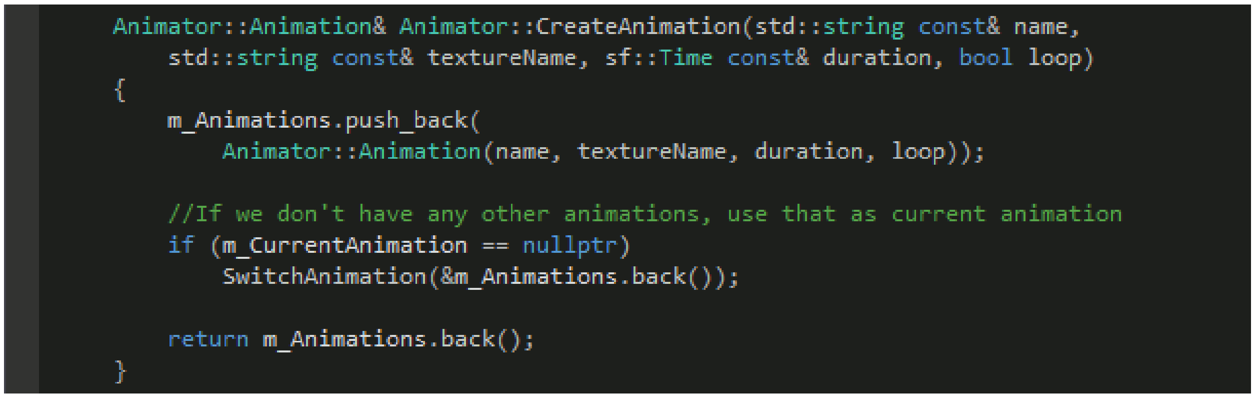

```cpp
struct Animation
{
    std::string m_Name;
    std::string m_TextureName;
    std::vector<sf::IntRect> m_Frames;
    sf::Time m_Duration;
    bool m_Looping;

    Animation(std::string const& name, std::string const& textureName,
        sf::Time const& duration, bool looping)
        : m_Name(name), m_TextureName(textureName),
        m_Duration(duration), m_Looping(looping)
    { }

    //Adds frames horizontally
    void AddFrames(sf::Vector2i const& startFrom,
        sf::Vector2i const& frameSize, unsigned int frames)
    {
        sf::Vector2i current = startFrom;
        for (unsigned int i = 0; i < frames; i++)
        {
            //Add current frame from position and frame size
            m_Frames.push_back(sf::IntRect(current.x, current.y, frameSize.x, frameSize.y));
            //Advance current frame horizontally
            current.x += frameSize.x;
        }
    }
};
```


```cpp
private:
    //Returns the animation with the passed name
    //Returns nullptr if no such animation is found
    Animator::Animation* FindAnimation(std::string const& name);

    void SwitchAnimation(Animator::Animation* animation);

    //Reference to the sprite
    sf::Sprite& m_Sprite;
    sf::Time m_CurrentTime;
    std::list<Animator::Animation> m_Animations;
    Animator::Animation* m_CurrentAnimation;

};
```

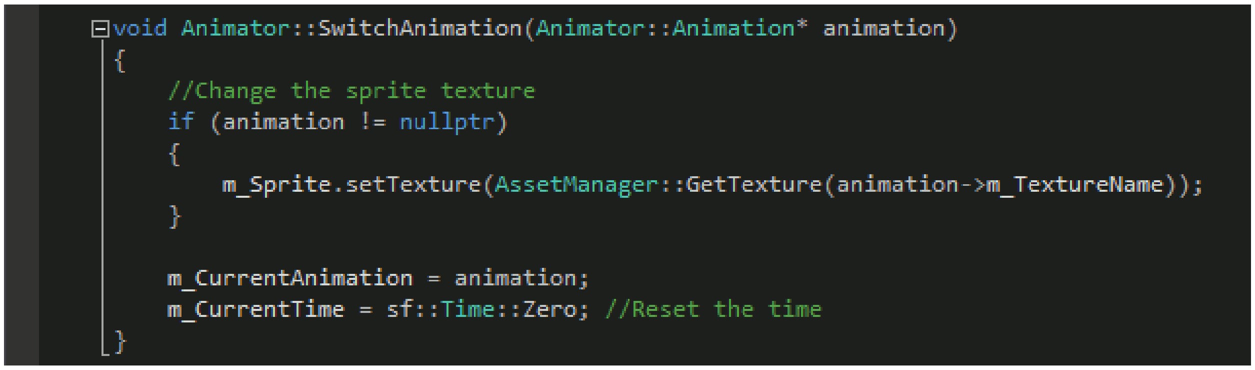

```cpp
public:
    struct Animation { ... };

    Animator(sf::Sprite& sprite);

    Animator::Animation& CreateAnimation(std::string const& name,
        std::string const& textureName, sf::Time const& duration, bool loop = false);

    void Update(sf::Time const& dt);

    //Returns if the switch was successful
    bool SwitchAnimation(std::string const& name);

    std::string GetCurrentAnimationName() const;
```



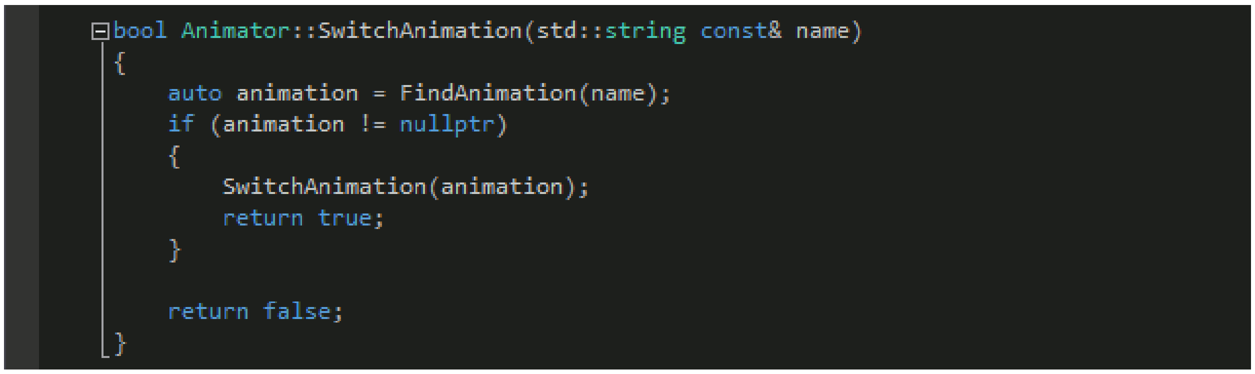
```cpp
Animator::Animator(sf::Sprite& sprite)
    :m_Sprite(sprite), m_CurrentTime(), m_CurrentAnimation(nullptr)
{
}
```



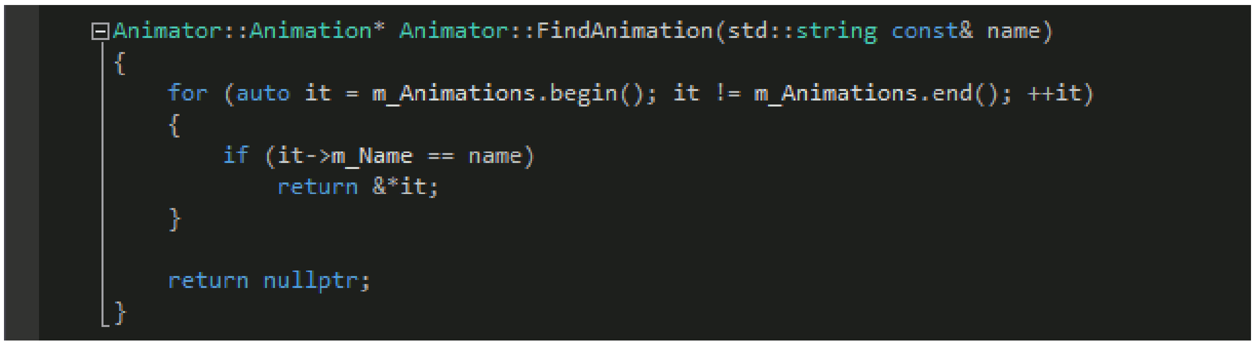
```cpp
Animator::Animation& Animator::CreateAnimation(std::string const& name,
    std::string const& textureName, sf::Time const& duration, bool loop)
{
    m_Animations.push_back(
        Animator::Animation(name, textureName, duration, loop));

    //If we don't have any other animations, use that as current animation
    if (m_CurrentAnimation == nullptr)
        SwitchAnimation(&m_Animations.back());

    return m_Animations.back();
}
```



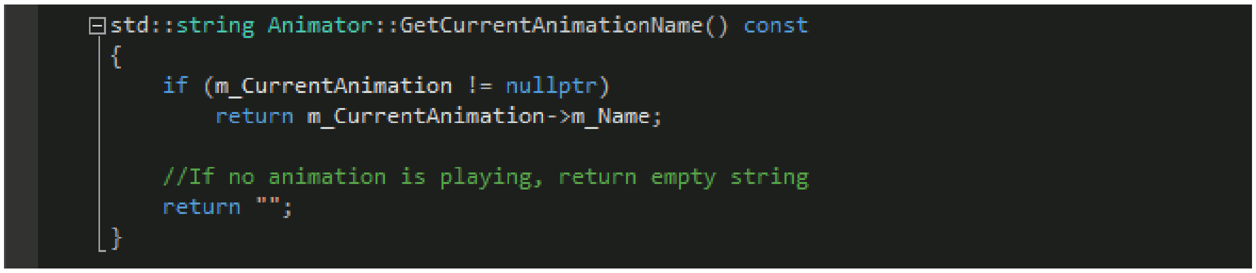
```cpp
void Animator::SwitchAnimation(Animator::Animation* animation)
{
    //Change the sprite texture
    if (animation != nullptr)
    {
        m_Sprite.setTexture(AssetManager::GetTexture(animation->m_TextureName));
    }

    m_CurrentAnimation = animation;
    m_CurrentTime = sf::Time::Zero; //Reset the time
}
```



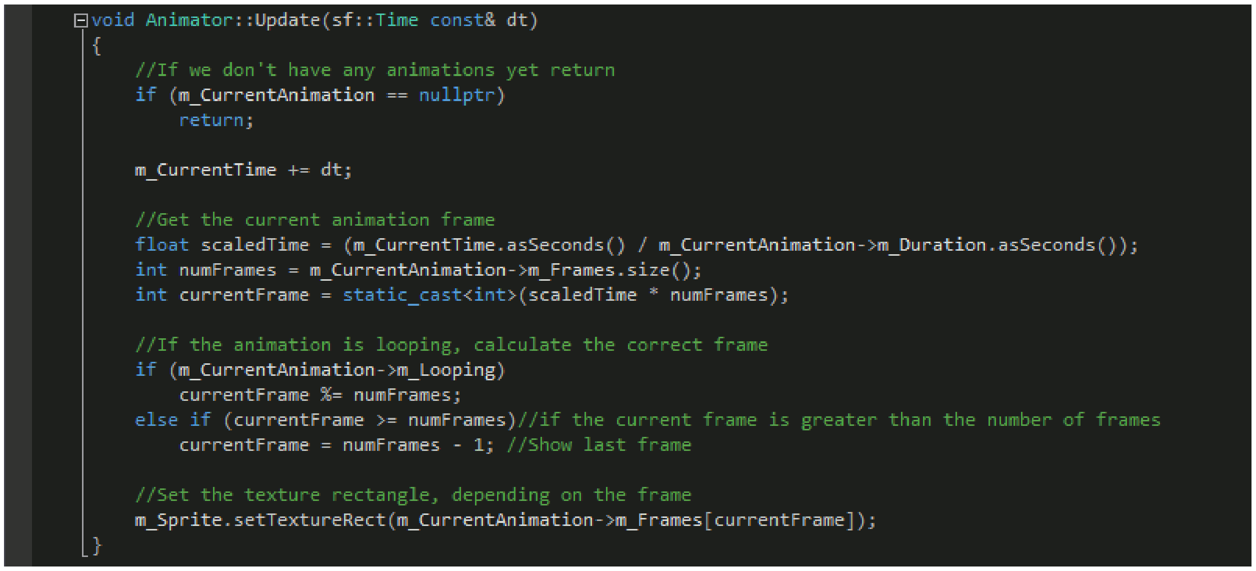
```cpp
bool Animator::SwitchAnimation(std::string const& name)
{
    auto animation = FindAnimation(name);
    if (animation != nullptr)
    {
        SwitchAnimation(animation);
        return true;
    }

    return false;
}
```



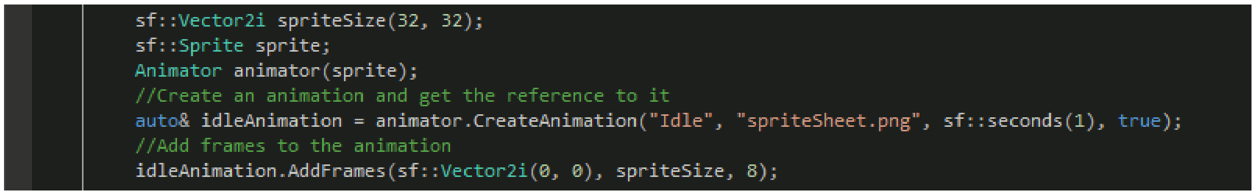
```cpp
Animator::Animation* Animator::FindAnimation(std::string const& name)
{
    for (auto it = m_Animations.begin(); it != m_Animations.end(); ++it)
    {
        if (it->m_Name == name)
            return &*it;
    }

    return nullptr;
}
```



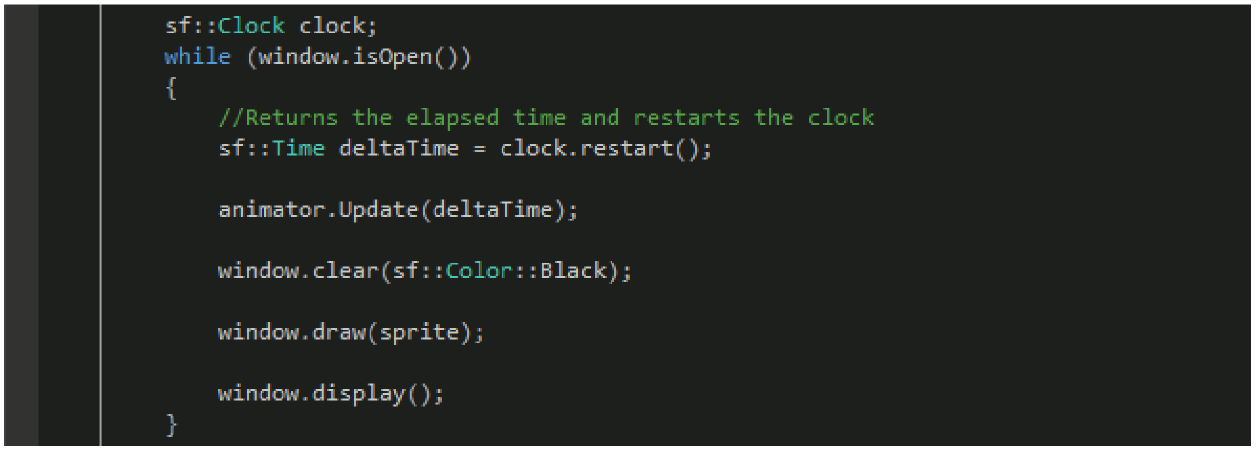
```cpp
std::string Animator::GetCurrentAnimationName() const
{
    if (m_CurrentAnimation != nullptr)
        return m_CurrentAnimation->m_Name;

    //If no animation is playing, return empty string
    return "";
}
```



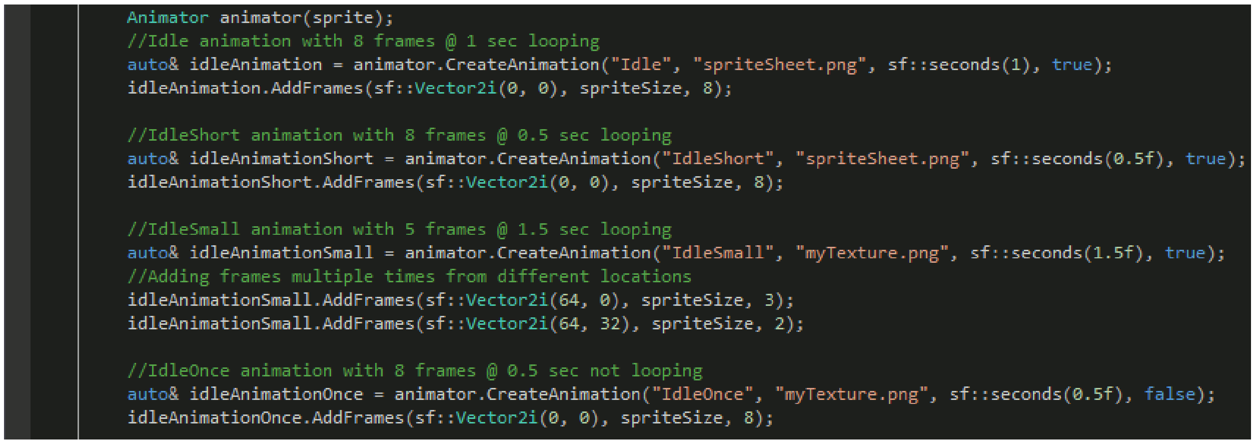
```cpp
void Animator::Update(sf::Time const& dt)
{
    //If we don't have any animations yet return
    if (m_CurrentAnimation == nullptr)
        return;

    m_CurrentTime += dt;

    //Get the current animation frame
    float scaledTime = (m_CurrentTime.asSeconds() / m_CurrentAnimation->m_Duration.asSeconds());
    int numFrames = m_CurrentAnimation->m_Frames.size();
    int currentFrame = static_cast<int>(scaledTime * numFrames);

    //If the animation is looping, calculate the correct frame
    if (m_CurrentAnimation->m_Looping)
        currentFrame %= numFrames;
    else if (currentFrame >= numFrames)//if the current frame is greater than the number of frames
        currentFrame = numFrames - 1; //Show last frame

    //Set the texture rectangle, depending on the frame
    m_Sprite.setTextureRect(m_CurrentAnimation->m_Frames[currentFrame]);
}
```



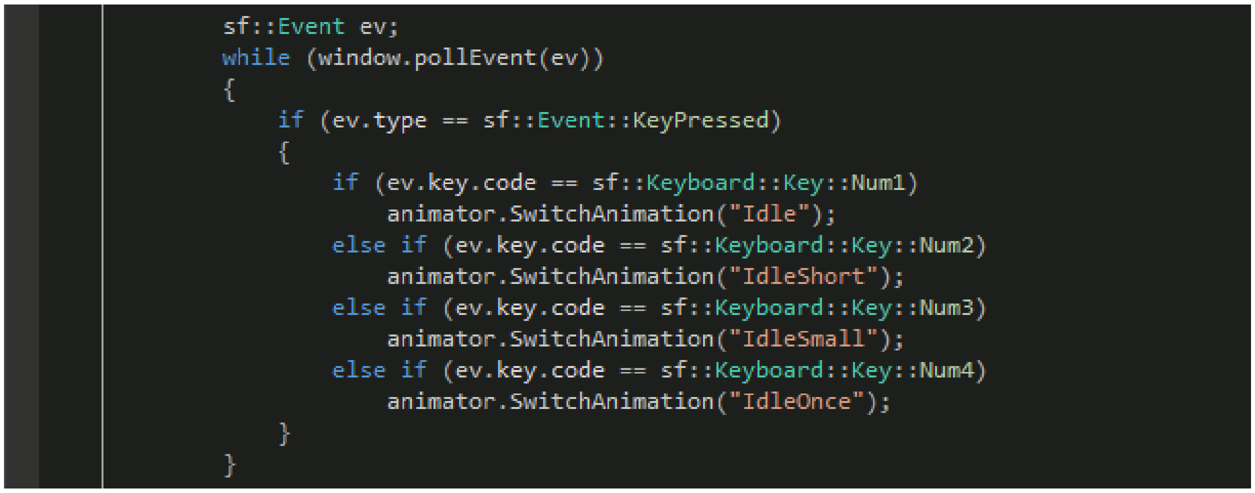
```cpp
    sf::Vector2i spriteSize(32, 32);
    sf::Sprite sprite;
    Animator animator(sprite);
    //Create an animation and get the reference to it
    auto& idleAnimation = animator.CreateAnimation("Idle", "spriteSheet.png", sf::seconds(1), true);
    //Add frames to the animation
    idleAnimation.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);
```



```cpp
    sf::Clock clock;
    while (window.isOpen())
    {
        //Returns the elapsed time and restarts the clock
        sf::Time deltaTime = clock.restart();

        animator.Update(deltaTime);

        window.clear(sf::Color::Black);

        window.draw(sprite);

        window.display();
    }
```

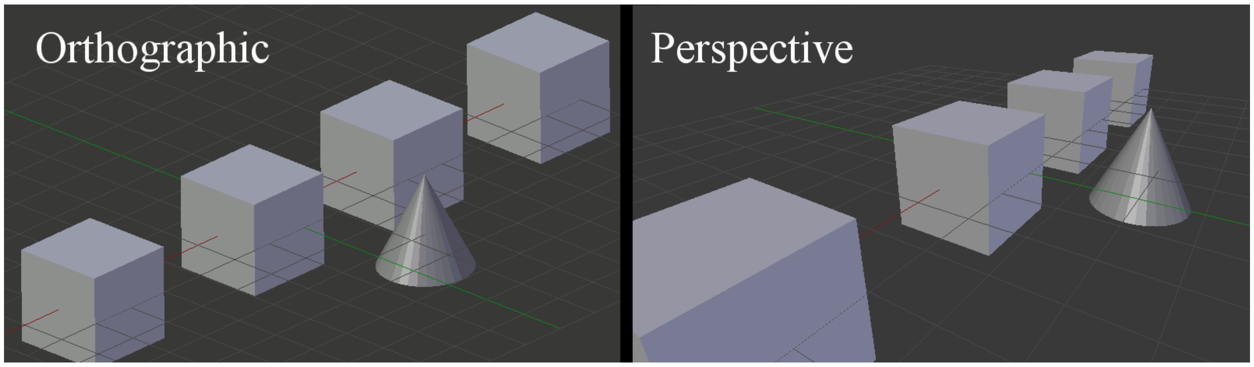


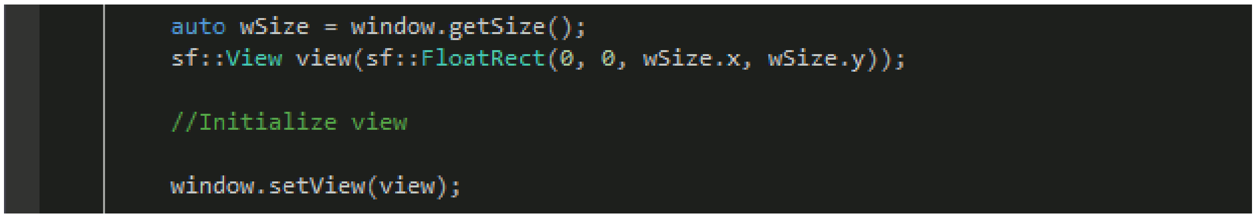
```cpp
Animator animator(sprite);
//Idle animation with 8 frames @ 1 sec looping
auto& idleAnimation = animator.CreateAnimation("Idle", "spriteSheet.png", sf::seconds(1), true);
idleAnimation.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);

//IdleShort animation with 8 frames @ 0.5 sec looping
auto& idleAnimationShort = animator.CreateAnimation("IdleShort", "spriteSheet.png", sf::seconds(0.5f), true);
idleAnimationShort.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);

//IdleSmall animation with 5 frames @ 1.5 sec looping
auto& idleAnimationSmall = animator.CreateAnimation("IdleSmall", "myTexture.png", sf::seconds(1.5f), true);
//Adding frames multiple times from different locations
idleAnimationSmall.AddFrames(sf::Vector2i(64, 0), spriteSize, 3);
idleAnimationSmall.AddFrames(sf::Vector2i(64, 32), spriteSize, 2);

//IdleOnce animation with 8 frames @ 0.5 sec not looping
auto& idleAnimationOnce = animator.CreateAnimation("IdleOnce", "myTexture.png", sf::seconds(0.5f), false);
idleAnimationOnce.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);
```



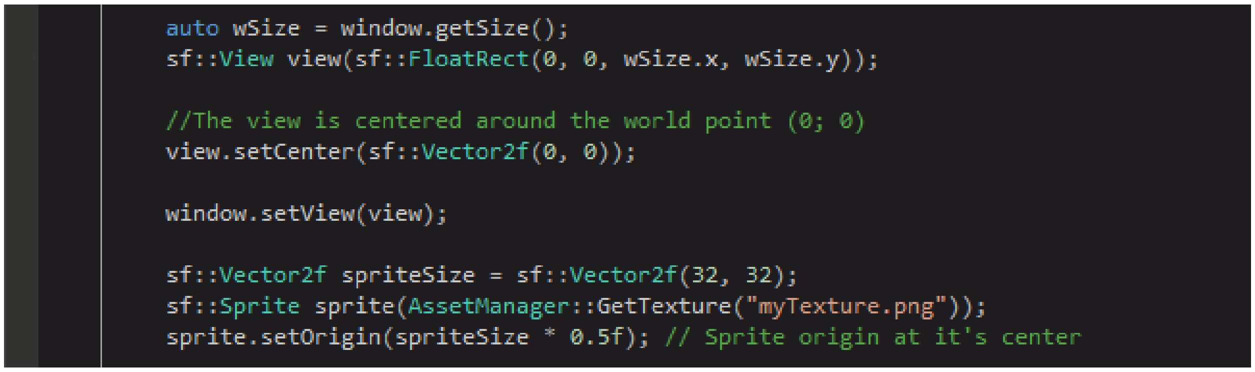
```cpp
sf::Event ev;
while (window.pollEvent(ev))
{
    if (ev.type == sf::Event::KeyPressed)
    {
        if (ev.key.code == sf::Keyboard::Key::Num1)
            animator.SwitchAnimation("Idle");
        else if (ev.key.code == sf::Keyboard::Key::Num2)
            animator.SwitchAnimation("IdleShort");
        else if (ev.key.code == sf::Keyboard::Key::Num3)
            animator.SwitchAnimation("IdleSmall");
        else if (ev.key.code == sf::Keyboard::Key::Num4)
            animator.SwitchAnimation("IdleOnce");
    }
}
```

Chapter 4: Manipulating a 2D Camera

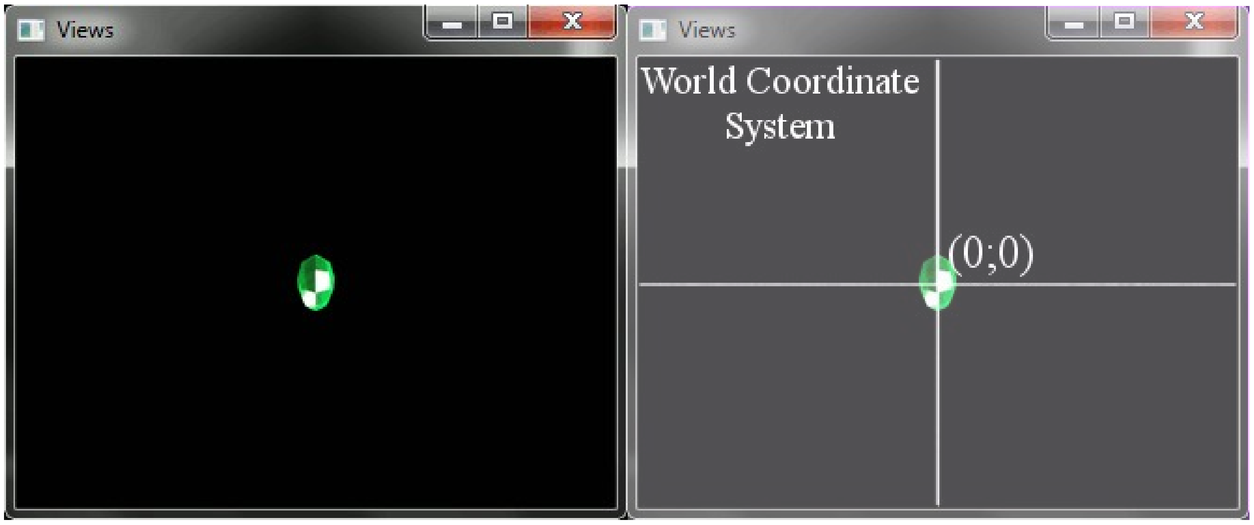


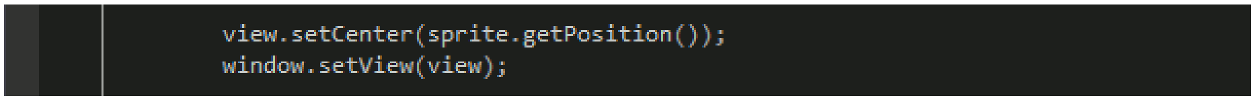


```
auto wSize = window.getSize();
sf::View view(sf::FloatRect(0, 0, wSize.x, wSize.y));

//Initialize view

window.setView(view);
```

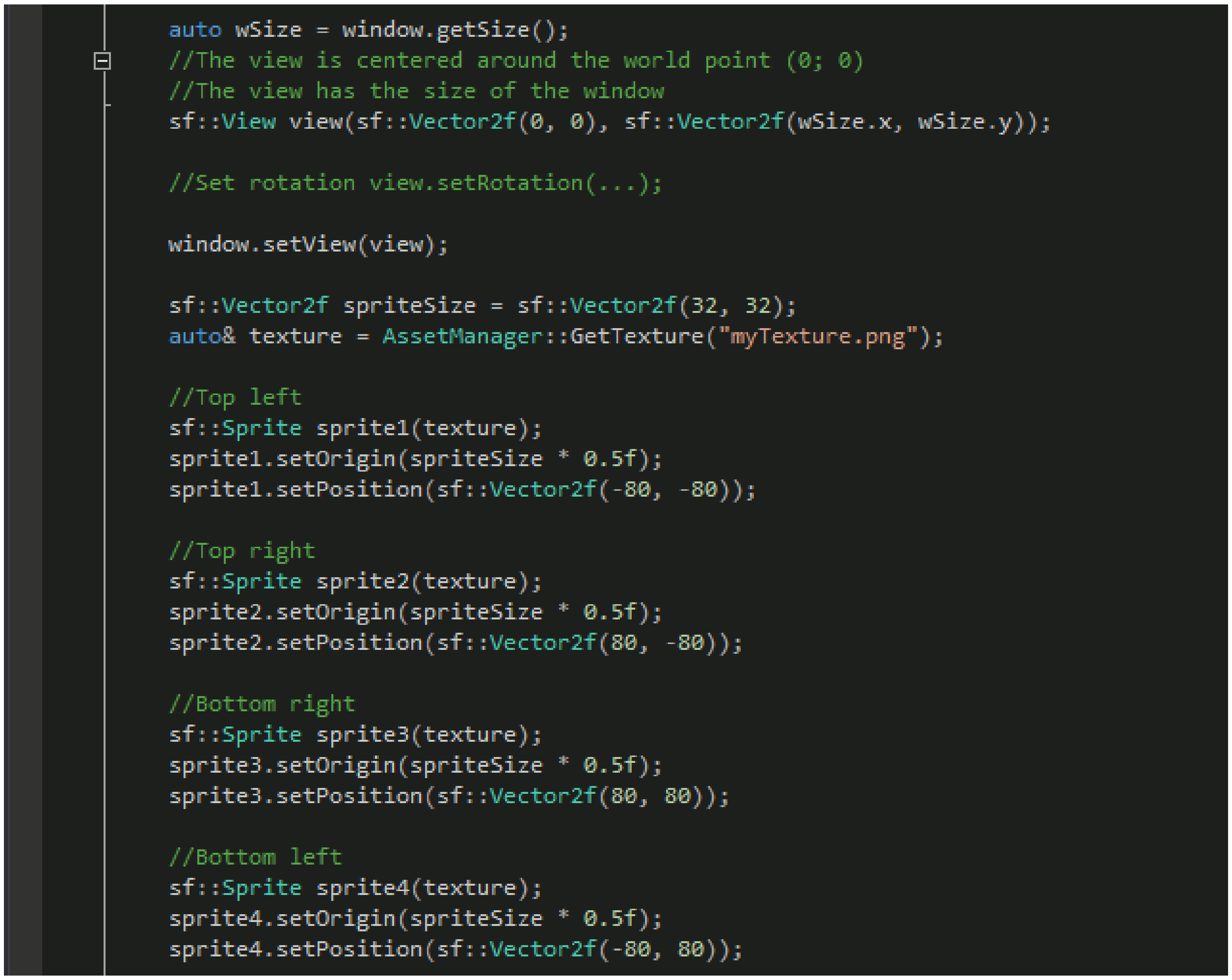


```
auto wSize = window.getSize();
sf::View view(sf::FloatRect(0, 0, wSize.x, wSize.y));

//The view is centered around the world point (0; 0)
view.setCenter(sf::Vector2f(0, 0));

window.setView(view);

sf::Vector2f spriteSize = sf::Vector2f(32, 32);
sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));
sprite.setOrigin(spriteSize * 0.5f); // Sprite origin at it's center
```

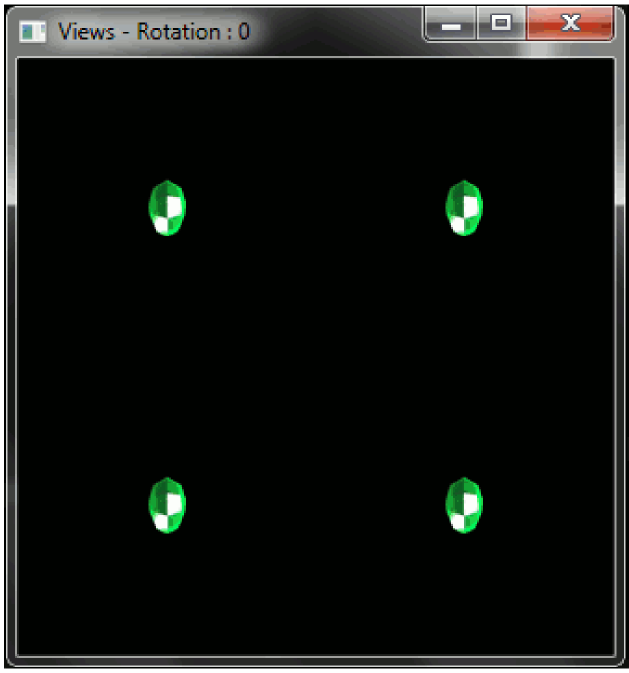


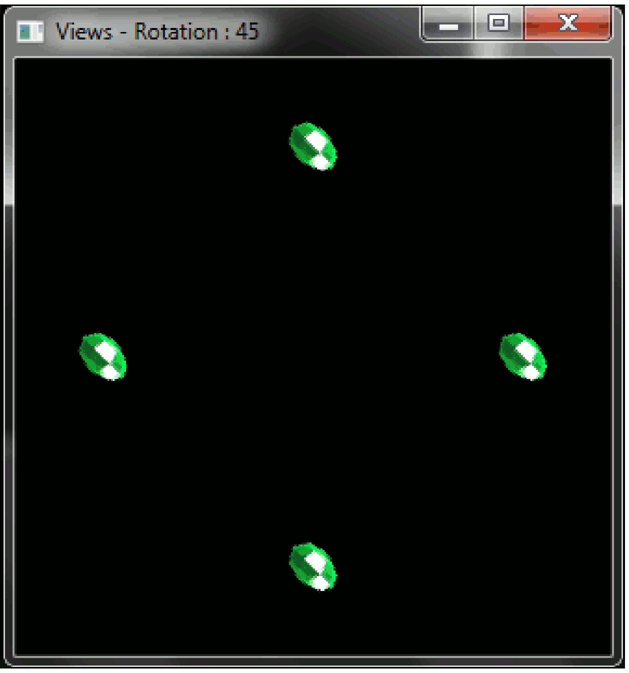


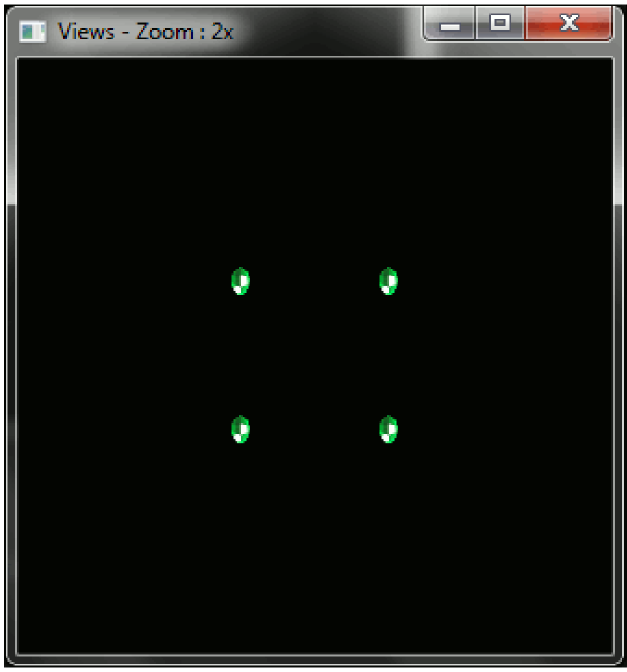
```
view.setCenter(sprite.getPosition());
window.setView(view);
```

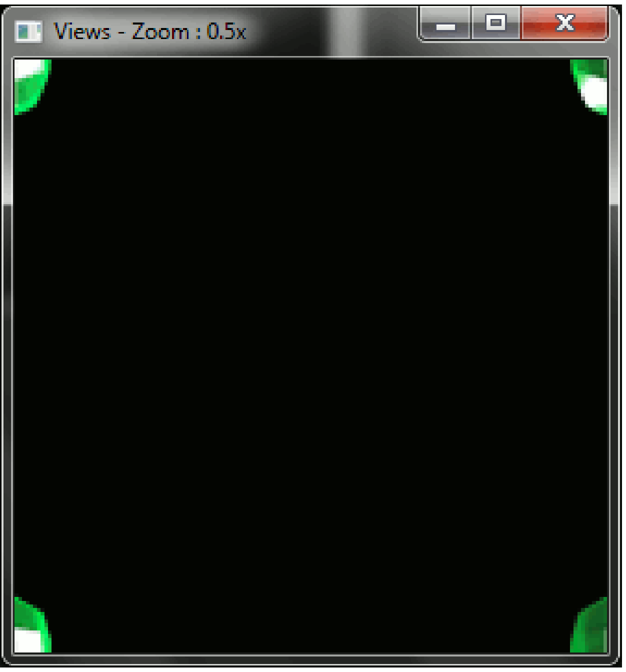


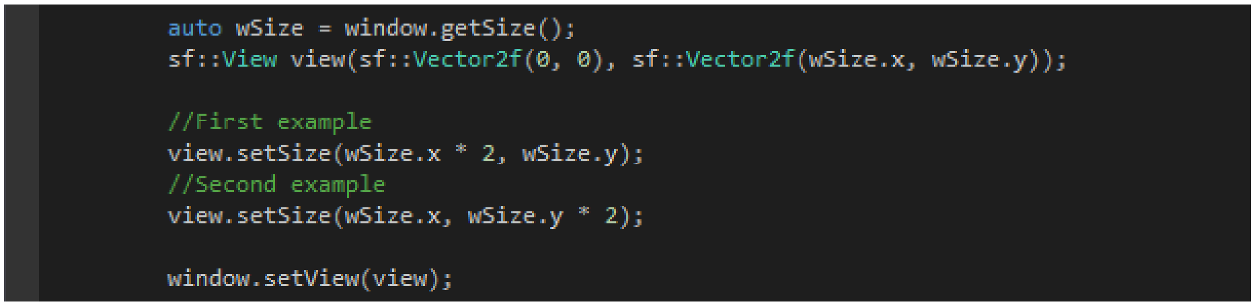
```
auto wSize = window.getSize();
//The view is centered around the world point (0; 0)
//The view has the size of the window
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

//Set rotation view.setRotation(...);

window.setView(view);

sf::Vector2f spriteSize = sf::Vector2f(32, 32);
auto& texture = AssetManager::GetTexture("myTexture.png");

//Top left
sf::Sprite sprite1(texture);
sprite1.setOrigin(spriteSize * 0.5f);
sprite1.setPosition(sf::Vector2f(-80, -80));

//Top right
sf::Sprite sprite2(texture);
sprite2.setOrigin(spriteSize * 0.5f);
sprite2.setPosition(sf::Vector2f(80, -80));

//Bottom right
sf::Sprite sprite3(texture);
sprite3.setOrigin(spriteSize * 0.5f);
sprite3.setPosition(sf::Vector2f(80, 80));

//Bottom left
sf::Sprite sprite4(texture);
sprite4.setOrigin(spriteSize * 0.5f);
sprite4.setPosition(sf::Vector2f(-80, 80));
```



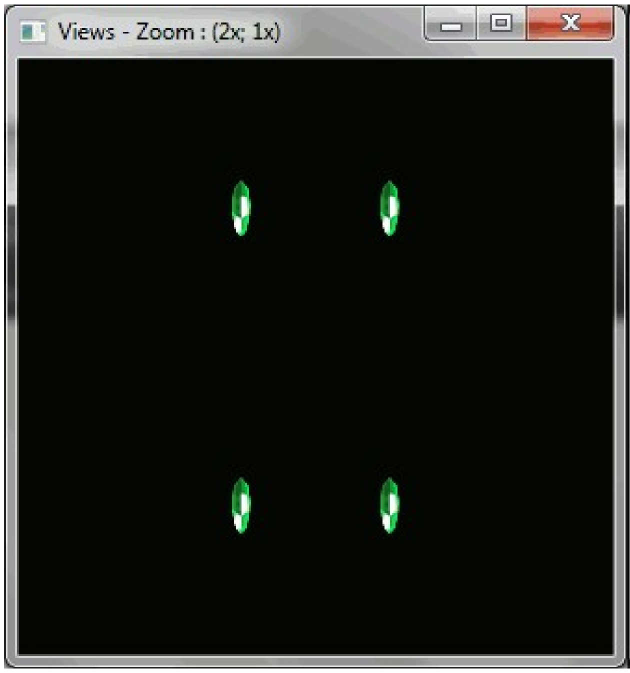


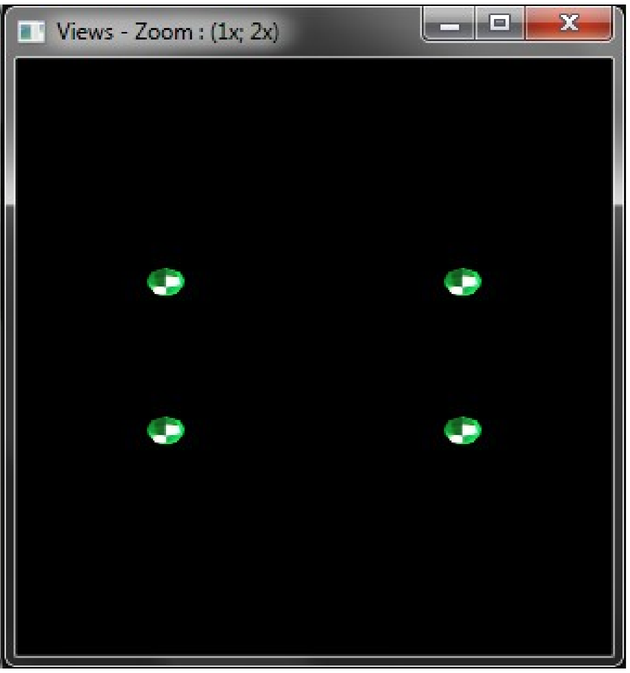


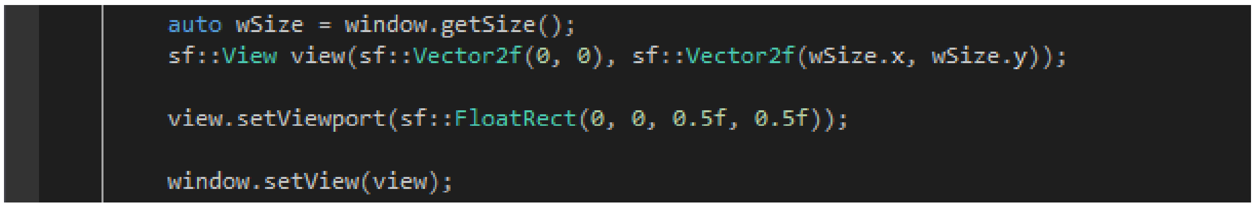




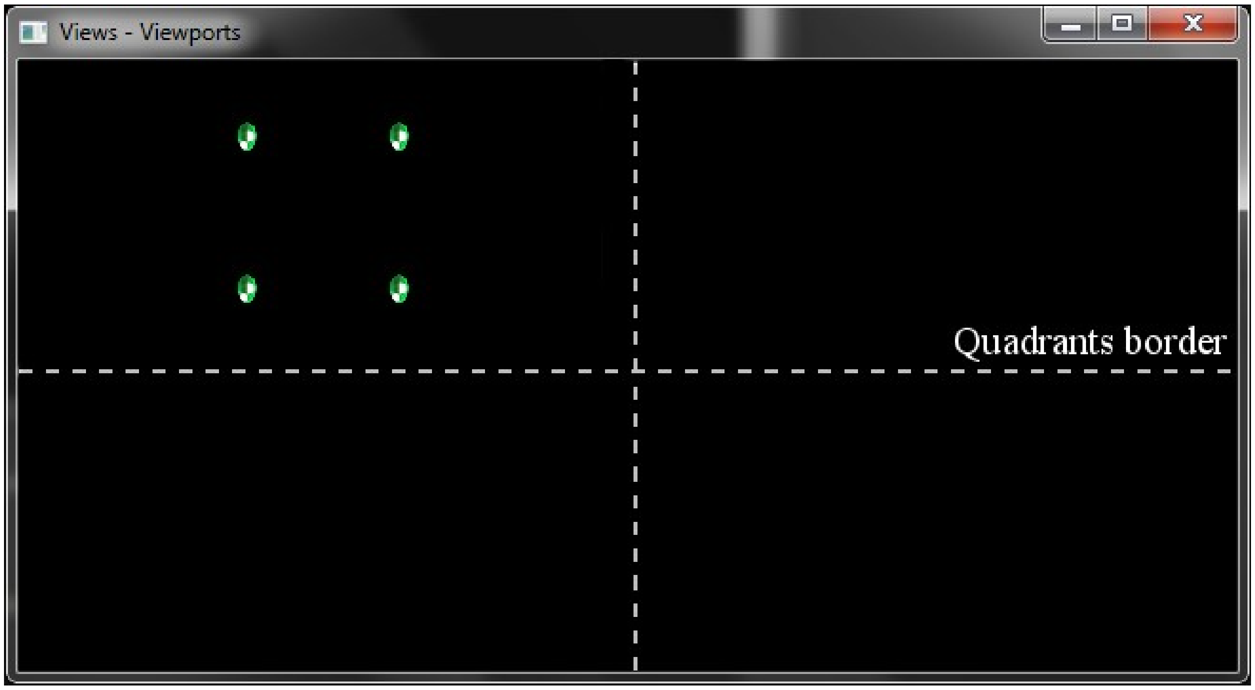
```cpp
auto wSize = window.getSize();
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

//First example
view.setSize(wSize.x * 2, wSize.y);
//Second example
view.setSize(wSize.x, wSize.y * 2);

window.setView(view);
```

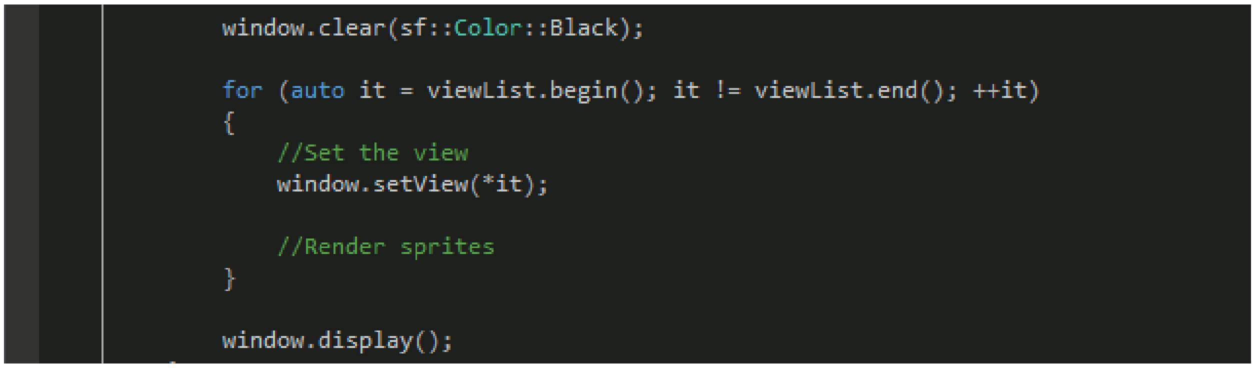




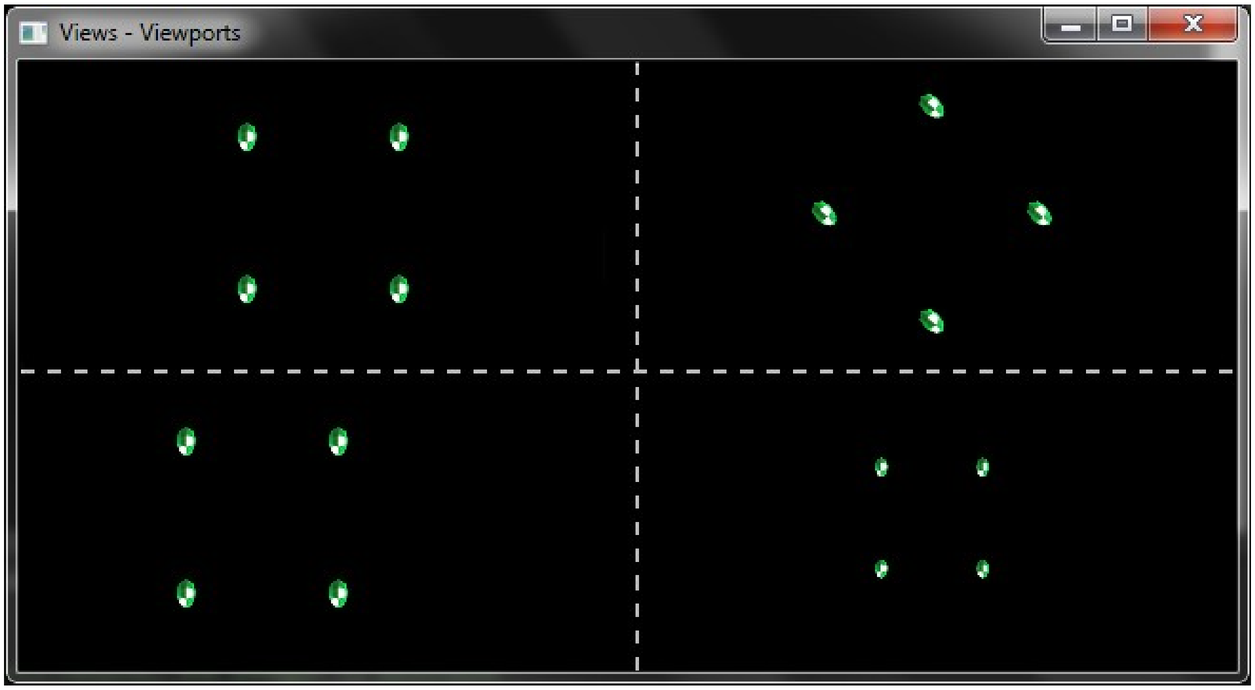


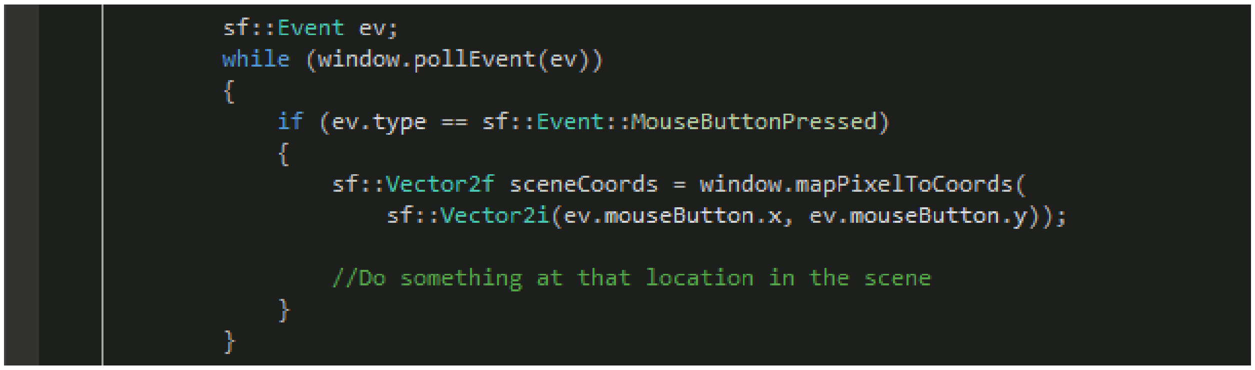
```
auto wSize = window.getSize();
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

view.setViewport(sf::FloatRect(0, 0, 0.5f, 0.5f));

window.setView(view);
```



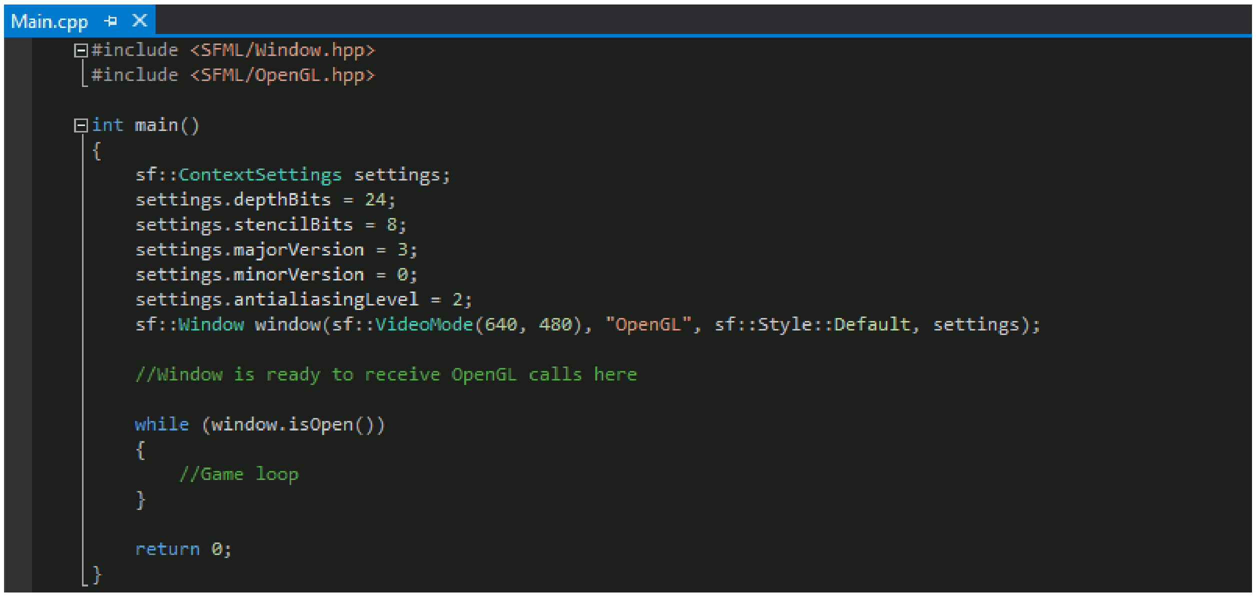


```
window.clear(sf::Color::Black);

for (auto it = viewList.begin(); it != viewList.end(); ++it)
{
    //Set the view
    window.setView(*it);

    //Render sprites
}

window.display();
```

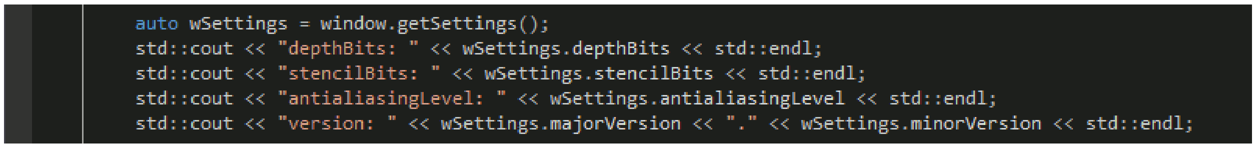




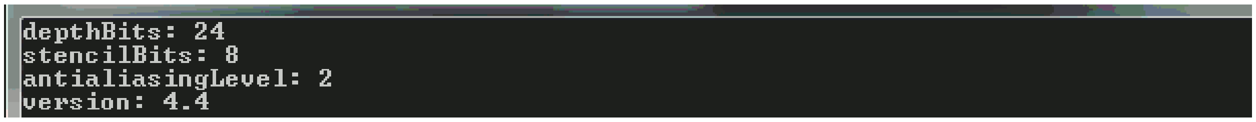
```
sf::Event ev;
while (window.pollEvent(ev))
{
    if (ev.type == sf::Event::MouseButtonPressed)
    {
        sf::Vector2f sceneCoords = window.mapPixelToCoords(
            sf::Vector2i(ev.mouseButton.x, ev.mouseButton.y));

        //Do something at that location in the scene
    }
}
```



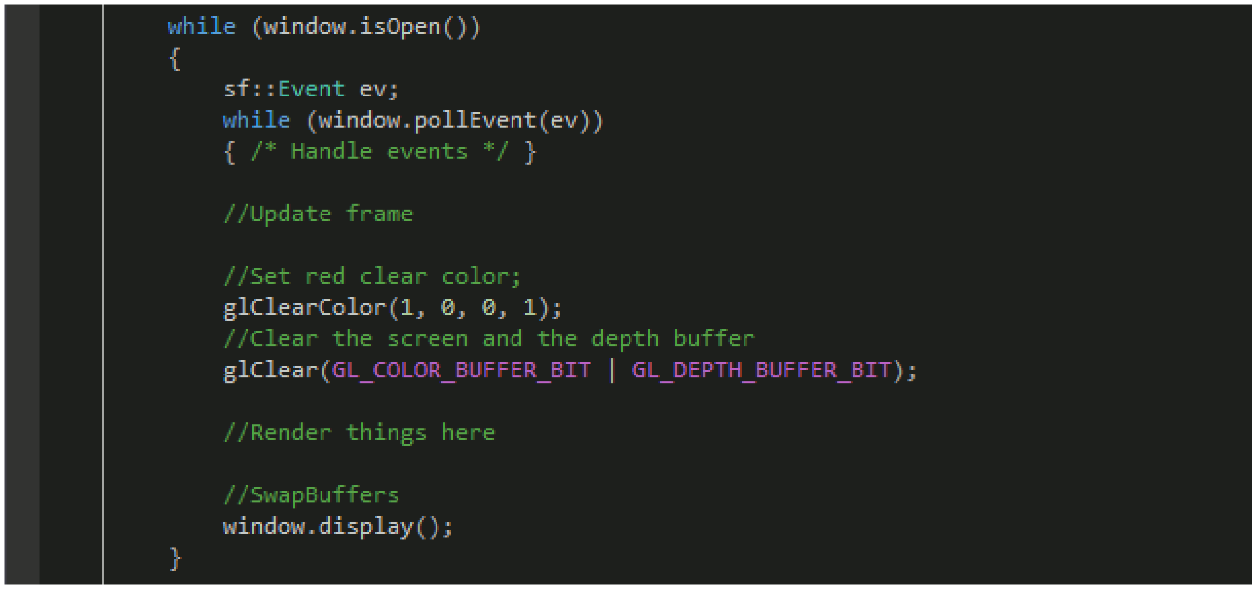
```cpp
#include <SFML/Window.hpp>
#include <SFML/OpenGL.hpp>

int main()
{
    sf::ContextSettings settings;
    settings.depthBits = 24;
    settings.stencilBits = 8;
    settings.majorVersion = 3;
    settings.minorVersion = 0;
    settings.antialiasingLevel = 2;
    sf::Window window(sf::VideoMode(640, 480), "OpenGL", sf::Style::Default, settings);

    //Window is ready to receive OpenGL calls here

    while (window.isOpen())
    {
        //Game loop
    }

    return 0;
}
```



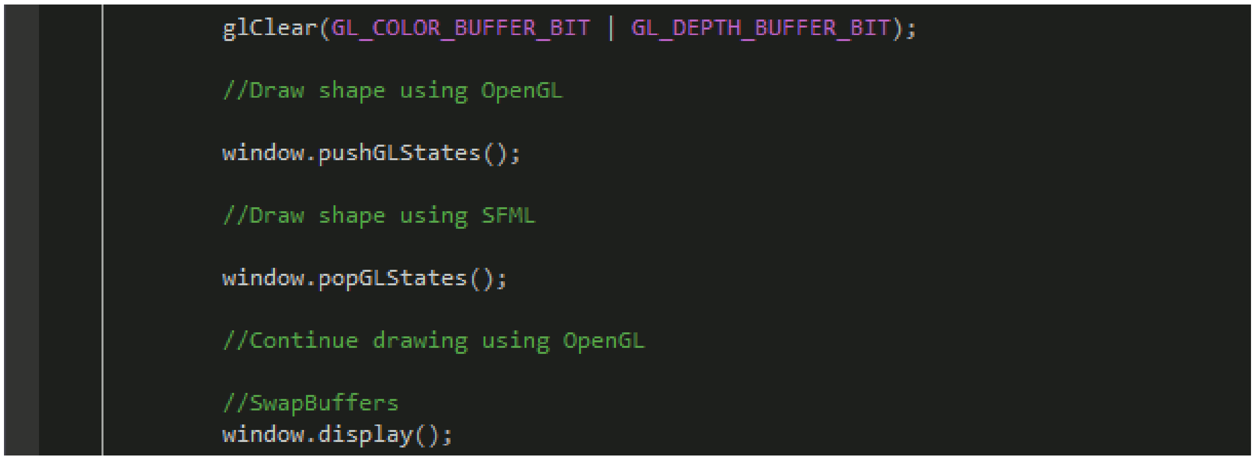
```cpp
    auto wSettings = window.getSettings();
    std::cout << "depthBits: " << wSettings.depthBits << std::endl;
    std::cout << "stencilBits: " << wSettings.stencilBits << std::endl;
    std::cout << "antialiasingLevel: " << wSettings.antialiasingLevel << std::endl;
    std::cout << "version: " << wSettings.majorVersion << "." << wSettings.minorVersion << std::endl;
```



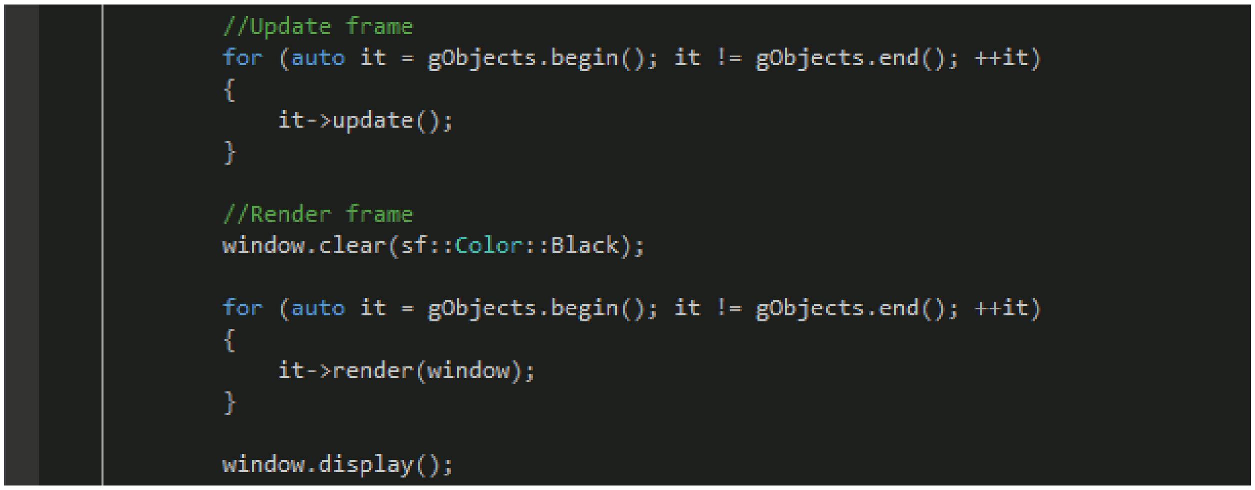
```
depthBits: 24
stencilBits: 8
antialiasingLevel: 2
version: 4.4
```



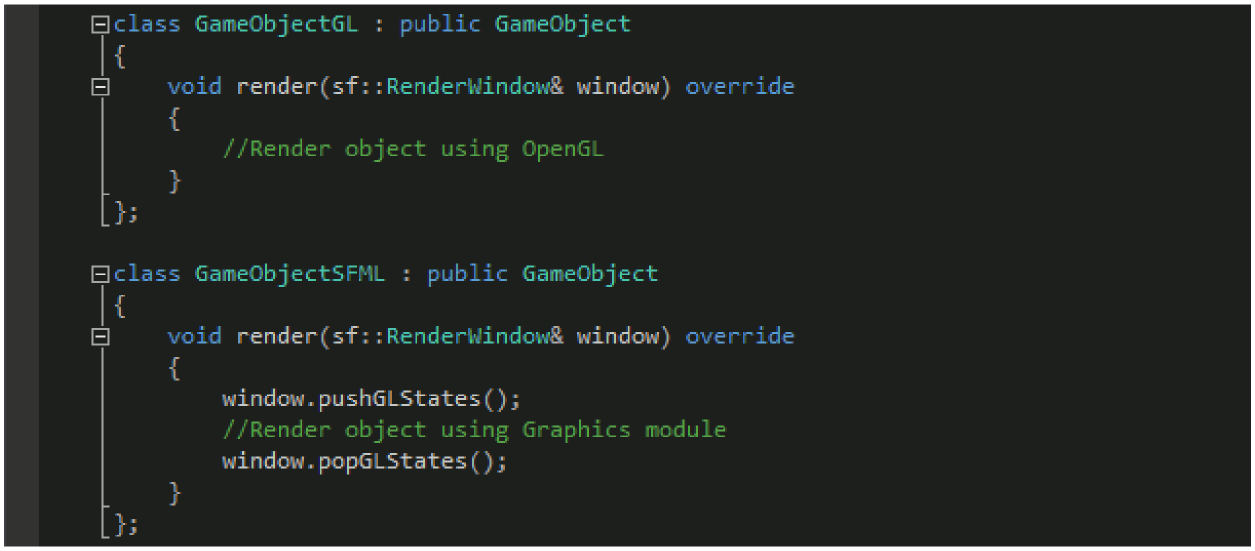
```cpp
    while (window.isOpen())
    {
        sf::Event ev;
        while (window.pollEvent(ev))
        { /* Handle events */ }

        //Update frame

        //Set red clear color;
        glClearColor(1, 0, 0, 1);
        //Clear the screen and the depth buffer
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        //Render things here

        //SwapBuffers
        window.display();
    }
```



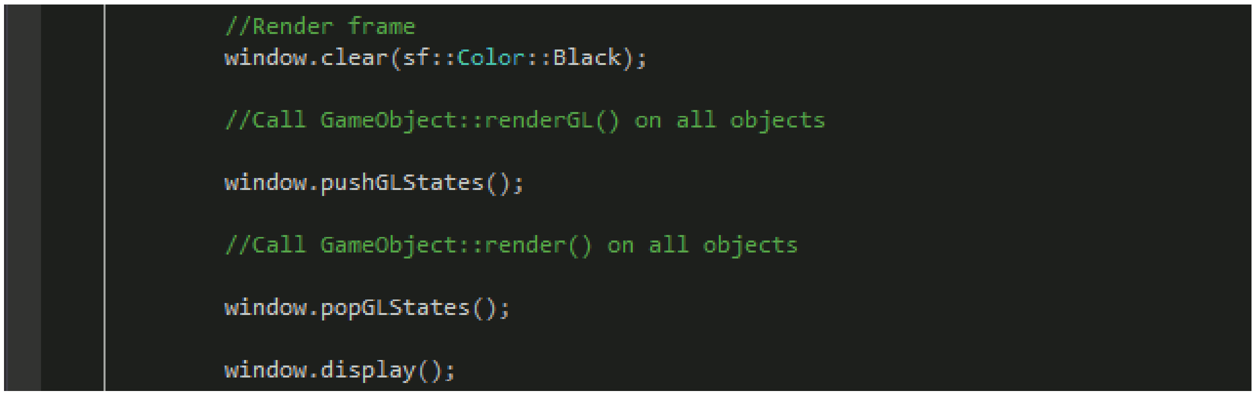
```cpp
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//Draw shape using OpenGL

window.pushGLStates();

//Draw shape using SFML

window.popGLStates();

//Continue drawing using OpenGL

//SwapBuffers
window.display();
```



```cpp
//Update frame
for (auto it = gObjects.begin(); it != gObjects.end(); ++it)
{
    it->update();
}

//Render frame
window.clear(sf::Color::Black);

for (auto it = gObjects.begin(); it != gObjects.end(); ++it)
{
    it->render(window);
}

window.display();
```

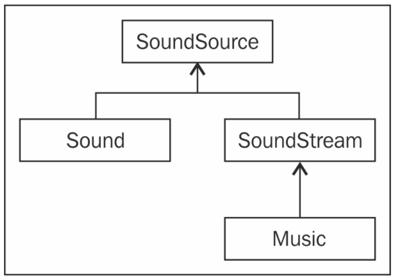


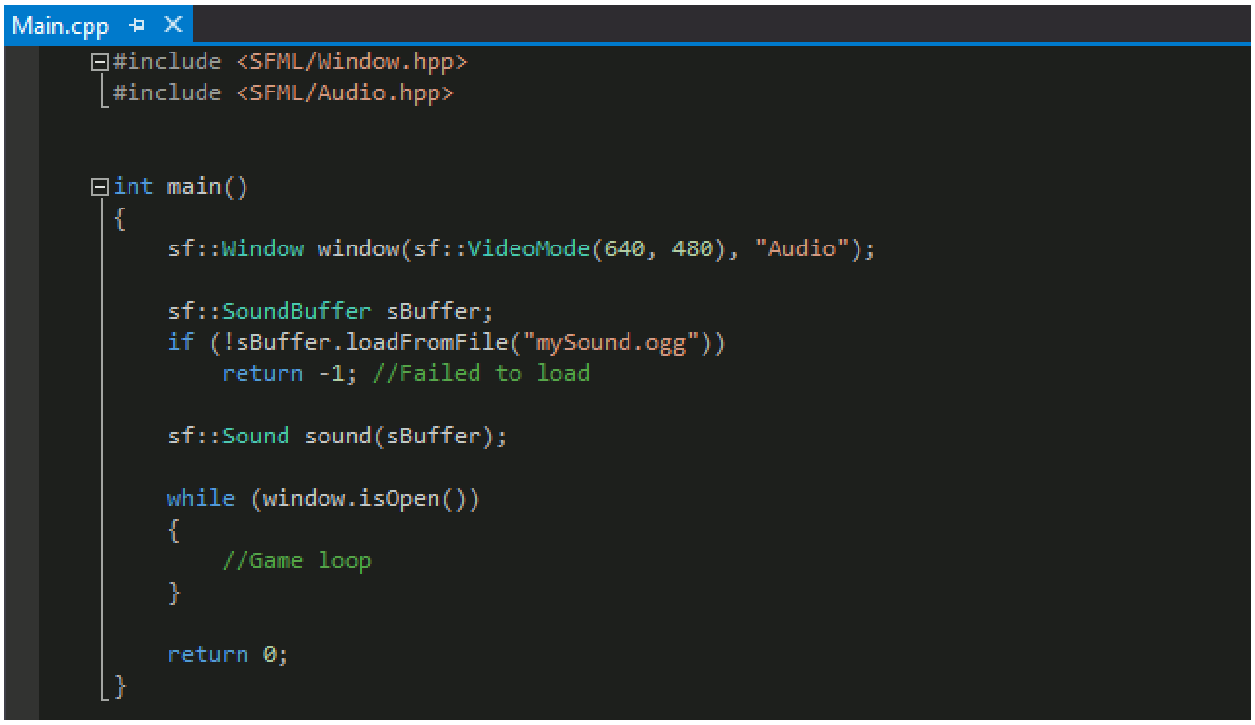
```cpp
class GameObjectGL : public GameObject
{
    void render(sf::RenderWindow& window) override
    {
        //Render object using OpenGL
    }
};

class GameObjectSFML : public GameObject
{
    void render(sf::RenderWindow& window) override
    {
        window.pushGLStates();
        //Render object using Graphics module
        window.popGLStates();
    }
};
```



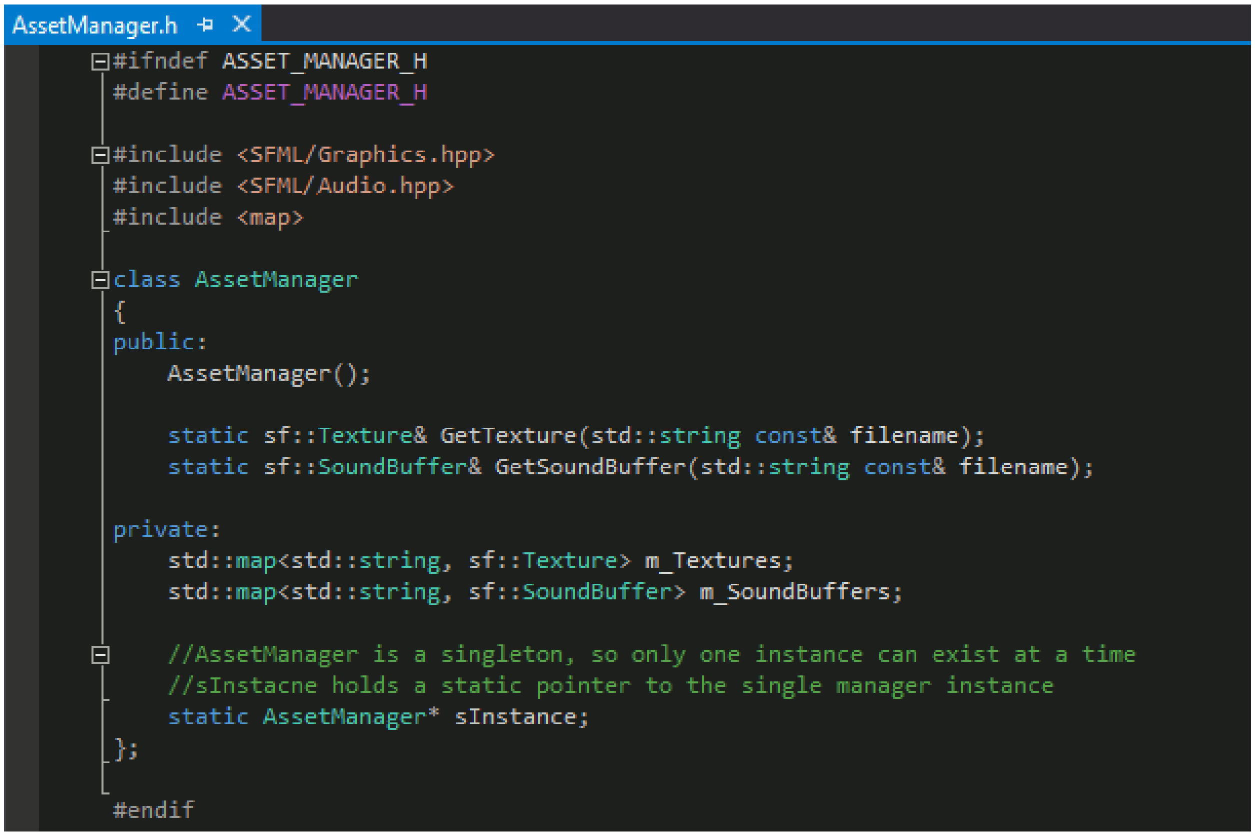
```
//Render frame
window.clear(sf::Color::Black);

//Call GameObject::renderGL() on all objects

window.pushGLStates();

//Call GameObject::render() on all objects

window.popGLStates();

window.display();
```

Chapter 5: Exploring a World of Sound and Text

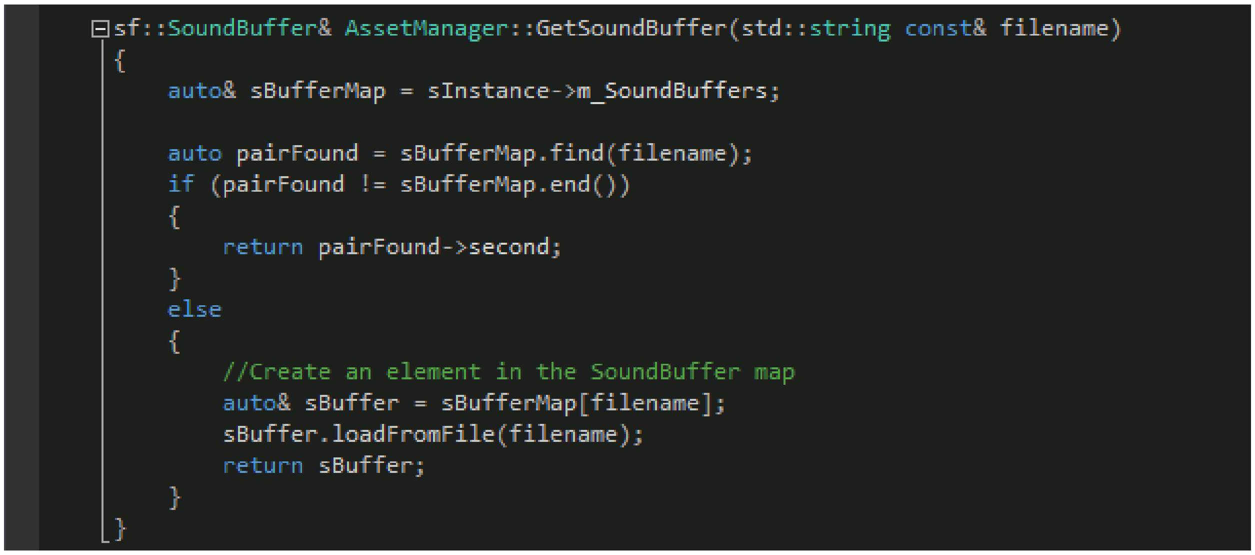




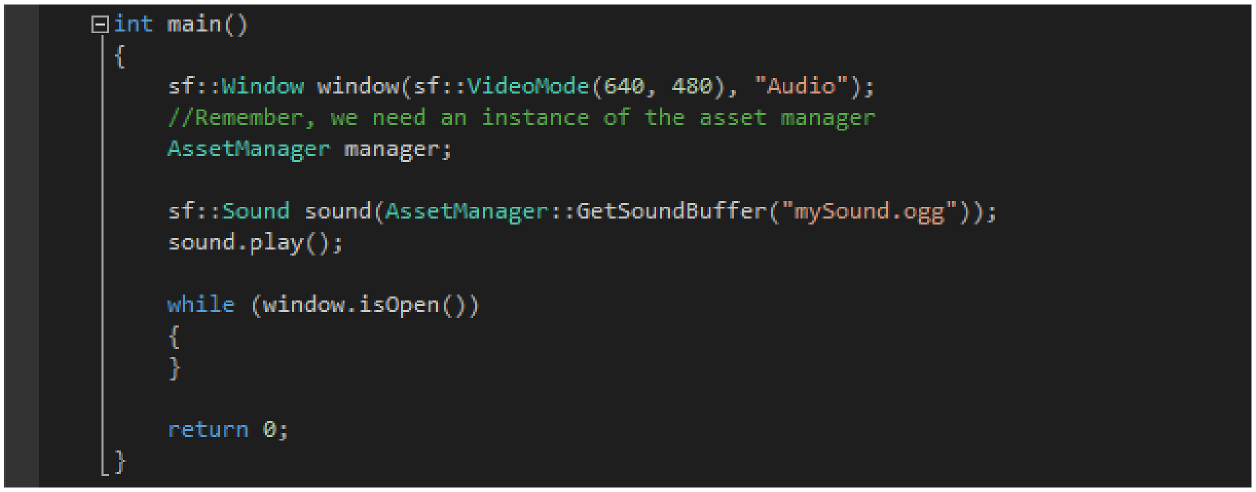
```cpp
#include <SFML/Window.hpp>
#include <SFML/Audio.hpp>


int main()
{
    sf::Window window(sf::VideoMode(640, 480), "Audio");

    sf::SoundBuffer sBuffer;
    if (!sBuffer.loadFromFile("mySound.ogg"))
        return -1; //Failed to load

    sf::Sound sound(sBuffer);

    while (window.isOpen())
    {
        //Game loop
    }

    return 0;
}
```



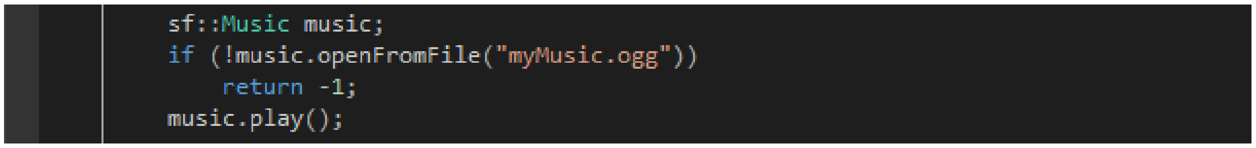
```
#ifndef ASSET_MANAGER_H
#define ASSET_MANAGER_H

#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include <map>

class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};

#endif
```



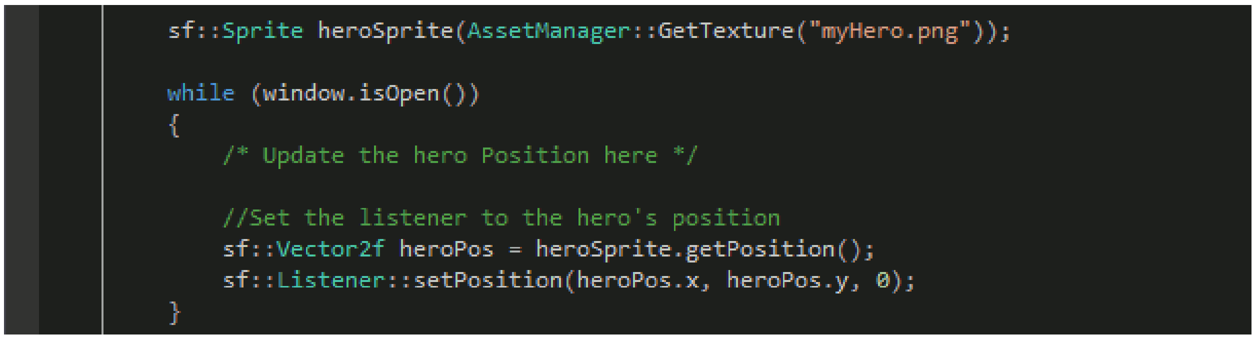
```
sf::SoundBuffer& AssetManager::GetSoundBuffer(std::string const& filename)
{
    auto& sBufferMap = sInstance->m_SoundBuffers;

    auto pairFound = sBufferMap.find(filename);
    if (pairFound != sBufferMap.end())
    {
        return pairFound->second;
    }
    else
    {
        //Create an element in the SoundBuffer map
        auto& sBuffer = sBufferMap[filename];
        sBuffer.loadFromFile(filename);
        return sBuffer;
    }
}
```



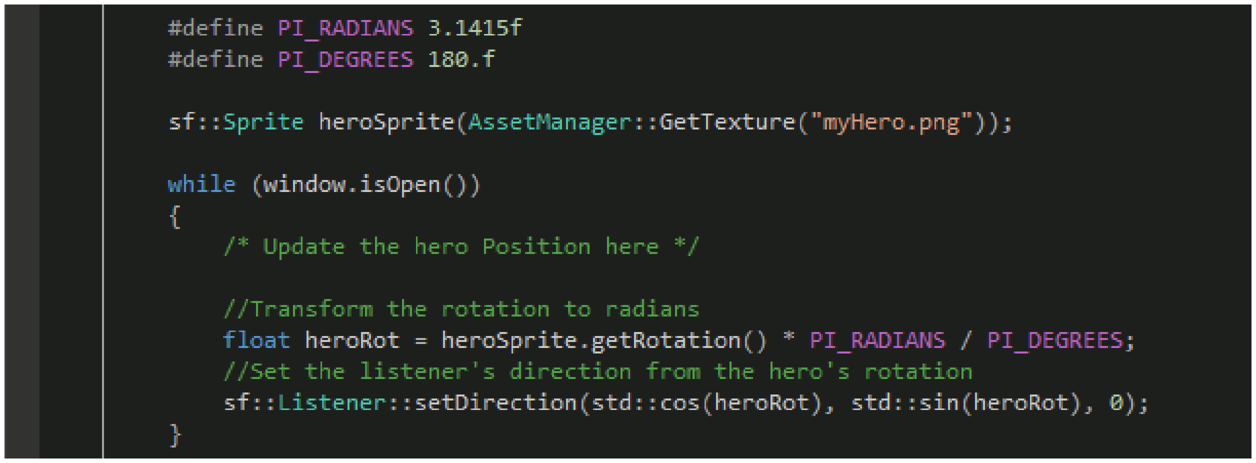
```cpp
int main()
{
    sf::Window window(sf::VideoMode(640, 480), "Audio");
    //Remember, we need an instance of the asset manager
    AssetManager manager;

    sf::Sound sound(AssetManager::GetSoundBuffer("mySound.ogg"));
    sound.play();

    while (window.isOpen())
    {
    }

    return 0;
}
```



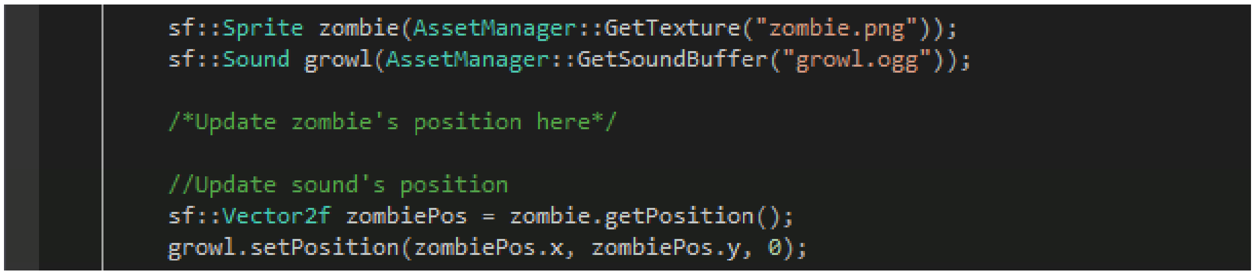
```cpp
    sf::Music music;
    if (!music.openFromFile("myMusic.ogg"))
        return -1;
    music.play();
```



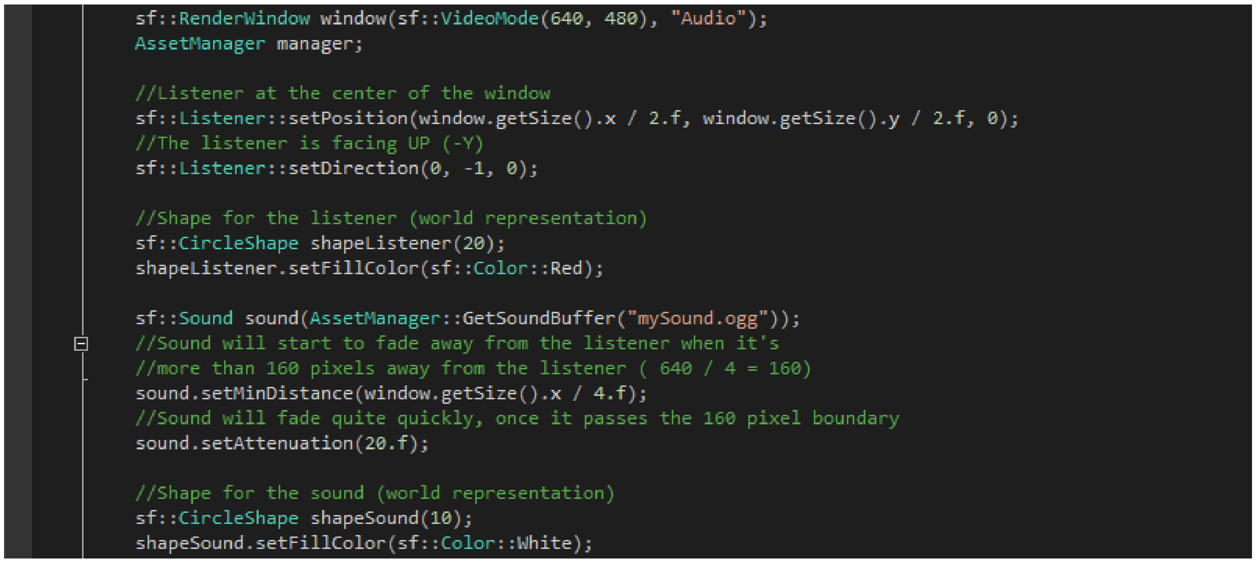
```cpp
    sf::Sprite heroSprite(AssetManager::GetTexture("myHero.png"));

    while (window.isOpen())
    {
        /* Update the hero Position here */

        //Set the listener to the hero's position
        sf::Vector2f heroPos = heroSprite.getPosition();
        sf::Listener::setPosition(heroPos.x, heroPos.y, 0);
    }
```



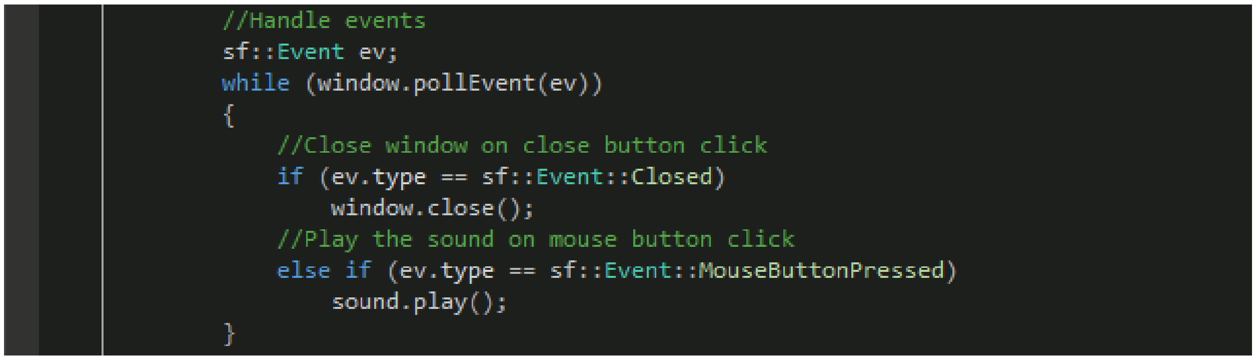
```cpp
    #define PI_RADIANS 3.1415f
    #define PI_DEGREES 180.f

    sf::Sprite heroSprite(AssetManager::GetTexture("myHero.png"));

    while (window.isOpen())
    {
        /* Update the hero Position here */

        //Transform the rotation to radians
        float heroRot = heroSprite.getRotation() * PI_RADIANS / PI_DEGREES;
        //Set the listener's direction from the hero's rotation
        sf::Listener::setDirection(std::cos(heroRot), std::sin(heroRot), 0);
    }
```



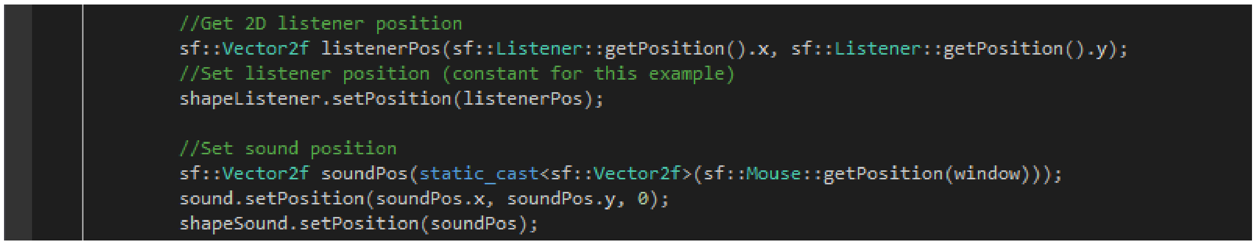
```
sf::Sprite zombie(AssetManager::GetTexture("zombie.png"));
sf::Sound growl(AssetManager::GetSoundBuffer("growl.ogg"));

/*Update zombie's position here*/

//Update sound's position
sf::Vector2f zombiePos = zombie.getPosition();
growl.setPosition(zombiePos.x, zombiePos.y, 0);
```



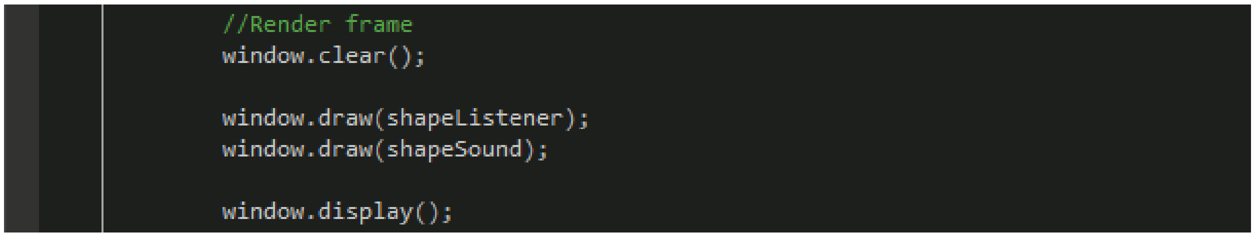
```
sf::RenderWindow window(sf::VideoMode(640, 480), "Audio");
AssetManager manager;

//Listener at the center of the window
sf::Listener::setPosition(window.getSize().x / 2.f, window.getSize().y / 2.f, 0);
//The listener is facing UP (-Y)
sf::Listener::setDirection(0, -1, 0);

//Shape for the listener (world representation)
sf::CircleShape shapeListener(20);
shapeListener.setFillColor(sf::Color::Red);

sf::Sound sound(AssetManager::GetSoundBuffer("mySound.ogg"));
//Sound will start to fade away from the listener when it's
//more than 160 pixels away from the listener ( 640 / 4 = 160)
sound.setMinDistance(window.getSize().x / 4.f);
//Sound will fade quite quickly, once it passes the 160 pixel boundary
sound.setAttenuation(20.f);

//Shape for the sound (world representation)
sf::CircleShape shapeSound(10);
shapeSound.setFillColor(sf::Color::White);
```



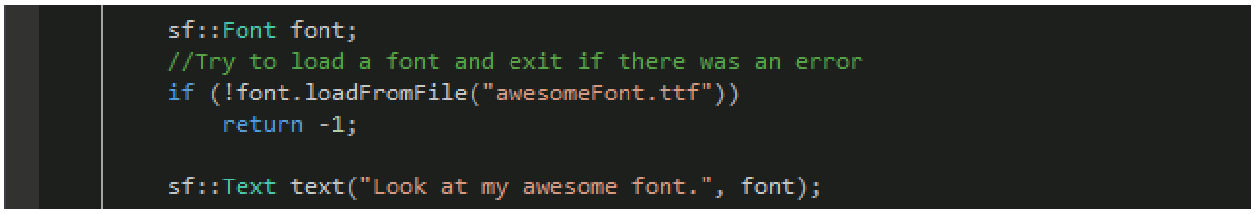
```
//Handle events
sf::Event ev;
while (window.pollEvent(ev))
{
    //Close window on close button click
    if (ev.type == sf::Event::Closed)
        window.close();
    //Play the sound on mouse button click
    else if (ev.type == sf::Event::MouseButtonPressed)
        sound.play();
}
```



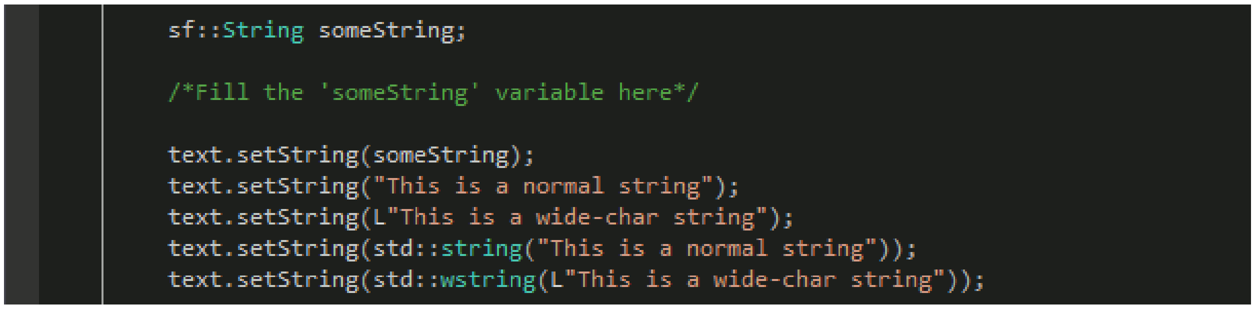
```
//Get 2D listener position
sf::Vector2f listenerPos(sf::Listener::getPosition().x, sf::Listener::getPosition().y);
//Set listener position (constant for this example)
shapeListener.setPosition(listenerPos);

//Set sound position
sf::Vector2f soundPos(static_cast<sf::Vector2f>(sf::Mouse::getPosition(window)));
sound.setPosition(soundPos.x, soundPos.y, 0);
shapeSound.setPosition(soundPos);
```



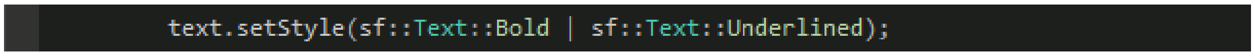
```
        //Render frame
        window.clear();

        window.draw(shapeListener);
        window.draw(shapeSound);

        window.display();
```



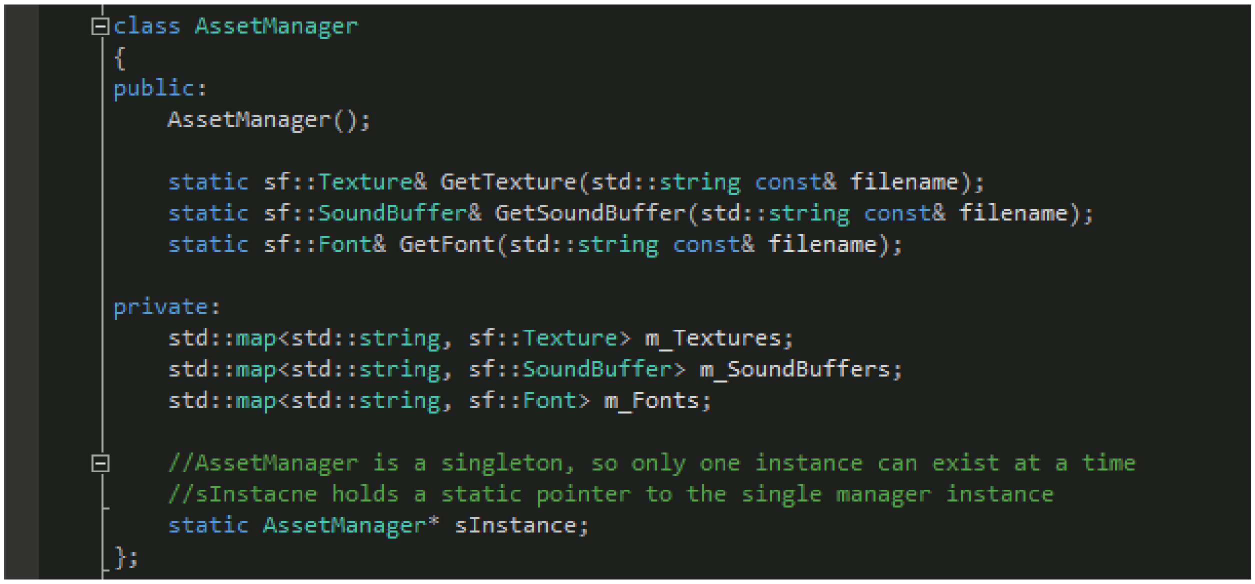
```
    sf::Font font;
    //Try to load a font and exit if there was an error
    if (!font.loadFromFile("awesomeFont.ttf"))
        return -1;

    sf::Text text("Look at my awesome font.", font);
```



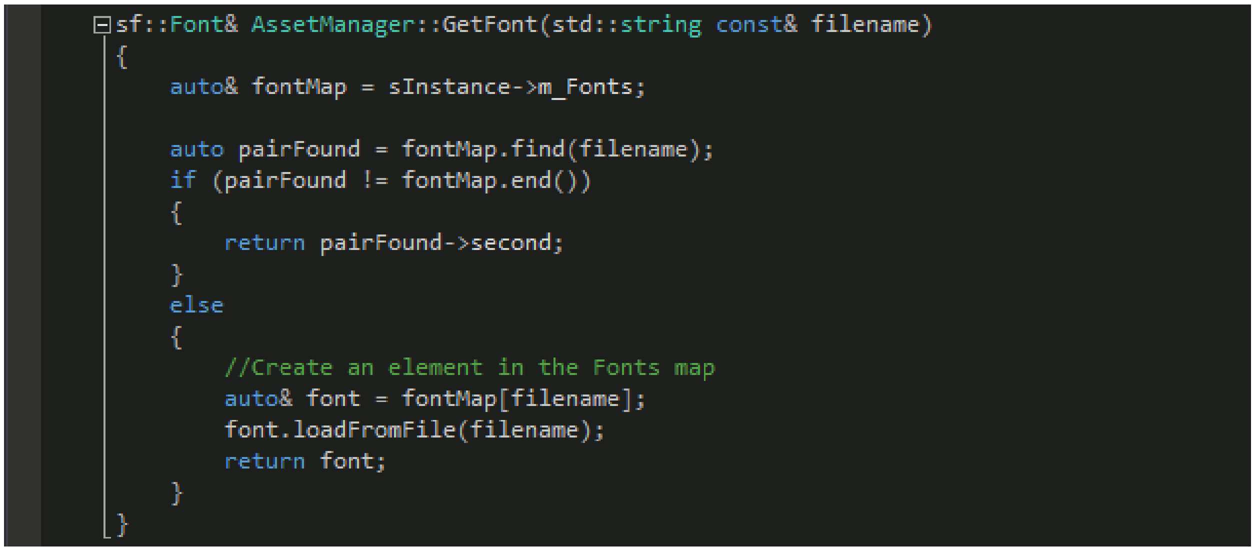
```
    sf::String someString;

    /*Fill the 'someString' variable here*/

    text.setString(someString);
    text.setString("This is a normal string");
    text.setString(L"This is a wide-char string");
    text.setString(std::string("This is a normal string"));
    text.setString(std::wstring(L"This is a wide-char string"));
```



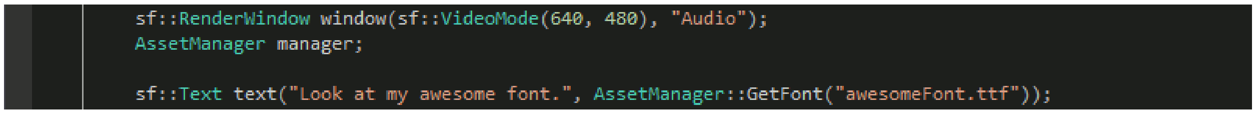
```
    text.setStyle(sf::Text::Bold | sf::Text::Underlined);
```



```
class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);
    static sf::Font& GetFont(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;
    std::map<std::string, sf::Font> m_Fonts;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};
```



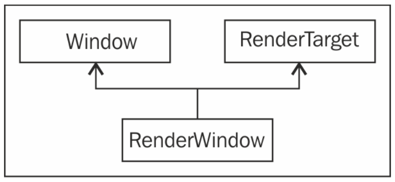
```
sf::Font& AssetManager::GetFont(std::string const& filename)
{
    auto& fontMap = sInstance->m_Fonts;

    auto pairFound = fontMap.find(filename);
    if (pairFound != fontMap.end())
    {
        return pairFound->second;
    }
    else
    {
        //Create an element in the Fonts map
        auto& font = fontMap[filename];
        font.loadFromFile(filename);
        return font;
    }
}
```



```
    sf::RenderWindow window(sf::VideoMode(640, 480), "Audio");
    AssetManager manager;

    sf::Text text("Look at my awesome font.", AssetManager::GetFont("awesomeFont.ttf"));
```

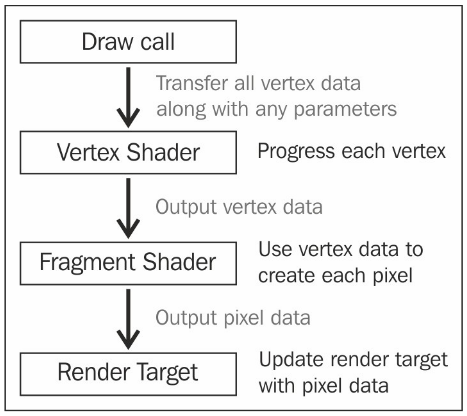
Angle_Radians = Angle_Degrees* PI_Radians / PI_Degrees

Chapter 6: Rendering Special Effects with Shaders



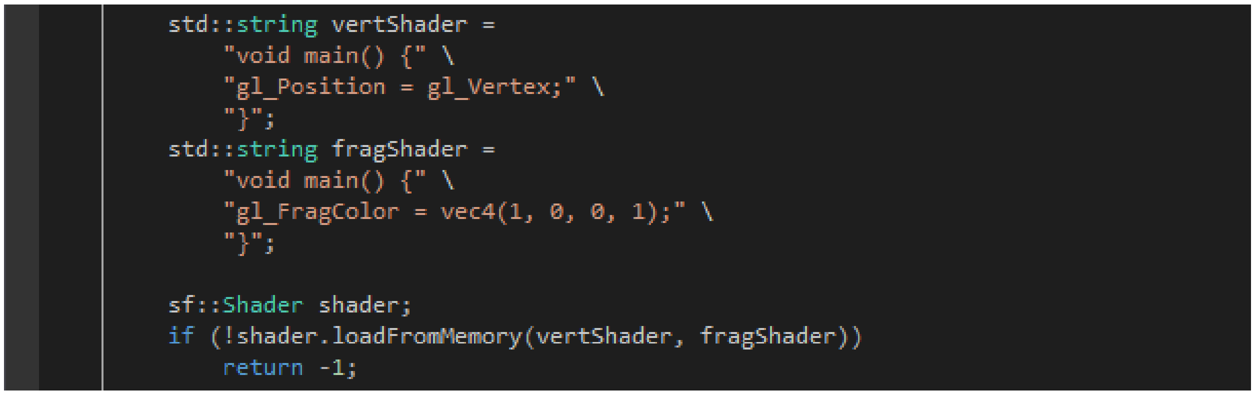
```
    window.draw(sprite, sf::BlendMode::BlendAdd);
```

```
sf::RenderTexture rTexture;
rTexture.create(32, 32, /*Depth buffer enabled = */ false);

sf::CircleShape circle(16); //Circle radius = 16

//Render routine - clear -> draw -> display
rTexture.clear();

rTexture.draw(circle);

rTexture.display();

//RenderTexture::getTexture() gets a ref to the Texture object
sf::Sprite sprite(rTexture.getTexture());

//Use the sprite in any way we like
```

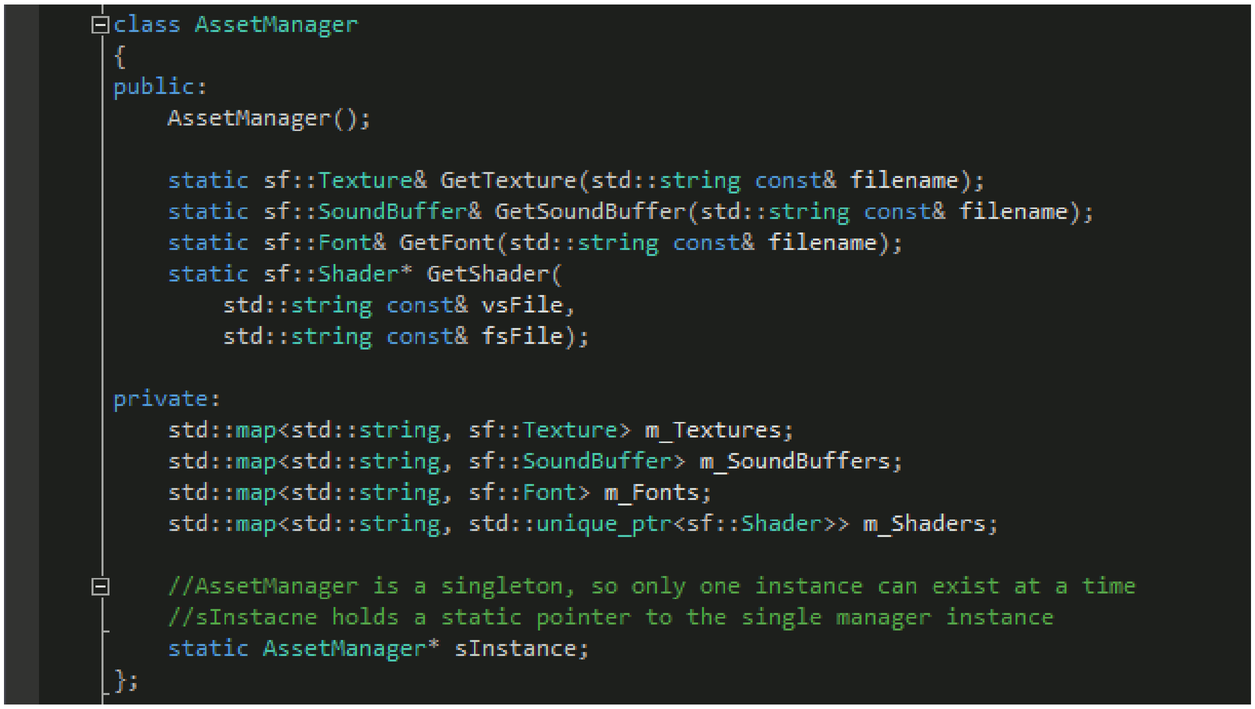

Draw call
Transfer all vertex data along with any parameters
Vertex Shader
Progress each vertex
Output vertex data
Fragment Shader
Use vertex data to create each pixel
Output pixel data
Render Target
Update render target with pixel data

```
if (!sf::Shader::isAvailable())
    return -1; //Shaders are not supported. Abort!
```

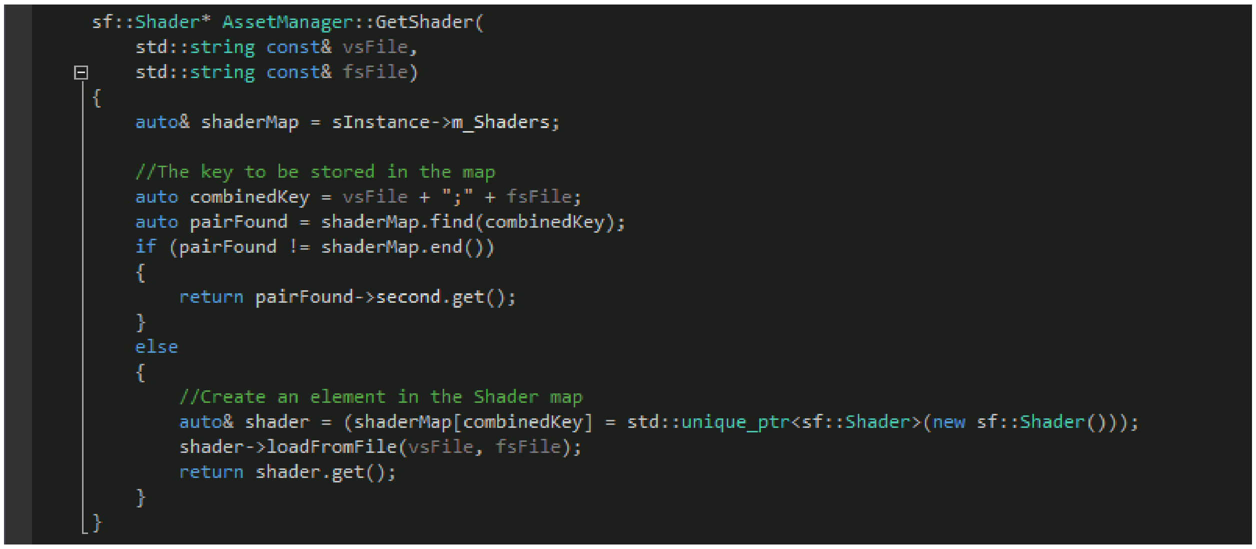
```
sf::Shader shader;
if (!shader.loadFromFile("vertShader.vert", "fragShader.frag"))
    return -1;

//Use the shader
```



```cpp
    std::string vertShader =
        "void main() {" \
        "gl_Position = gl_Vertex;" \
        "}";
    std::string fragShader =
        "void main() {" \
        "gl_FragColor = vec4(1, 0, 0, 1);" \
        "}";

    sf::Shader shader;
    if (!shader.loadFromMemory(vertShader, fragShader))
        return -1;
```



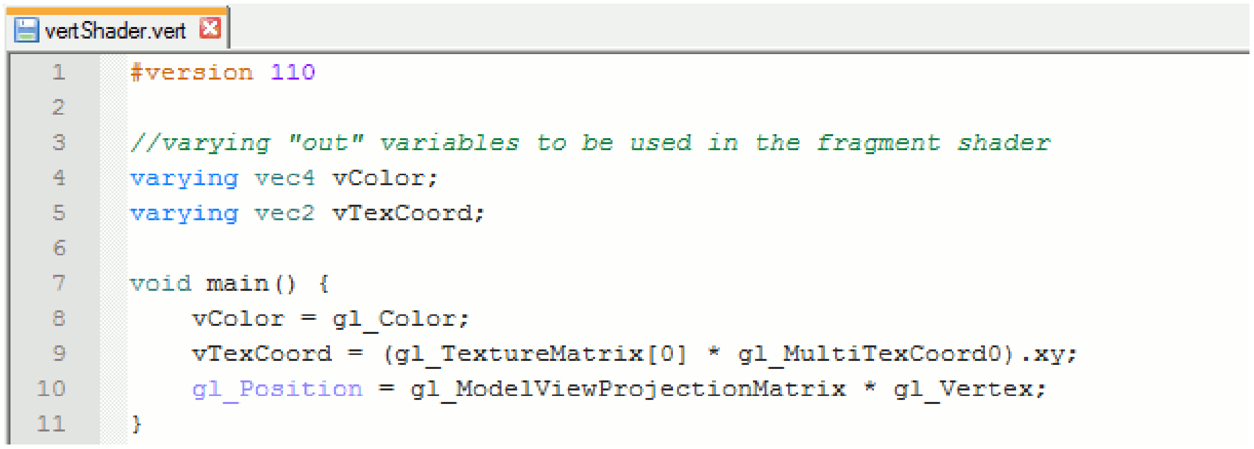
```cpp
class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);
    static sf::Font& GetFont(std::string const& filename);
    static sf::Shader* GetShader(
        std::string const& vsFile,
        std::string const& fsFile);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;
    std::map<std::string, sf::Font> m_Fonts;
    std::map<std::string, std::unique_ptr<sf::Shader>> m_Shaders;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};
```



```cpp
sf::Shader* AssetManager::GetShader(
    std::string const& vsFile,
    std::string const& fsFile)
{
    auto& shaderMap = sInstance->m_Shaders;

    //The key to be stored in the map
    auto combinedKey = vsFile + ";" + fsFile;
    auto pairFound = shaderMap.find(combinedKey);
    if (pairFound != shaderMap.end())
    {
        return pairFound->second.get();
    }
    else
    {
        //Create an element in the Shader map
        auto& shader = (shaderMap[combinedKey] = std::unique_ptr<sf::Shader>(new sf::Shader()));
        shader->loadFromFile(vsFile, fsFile);
        return shader.get();
    }
}
```

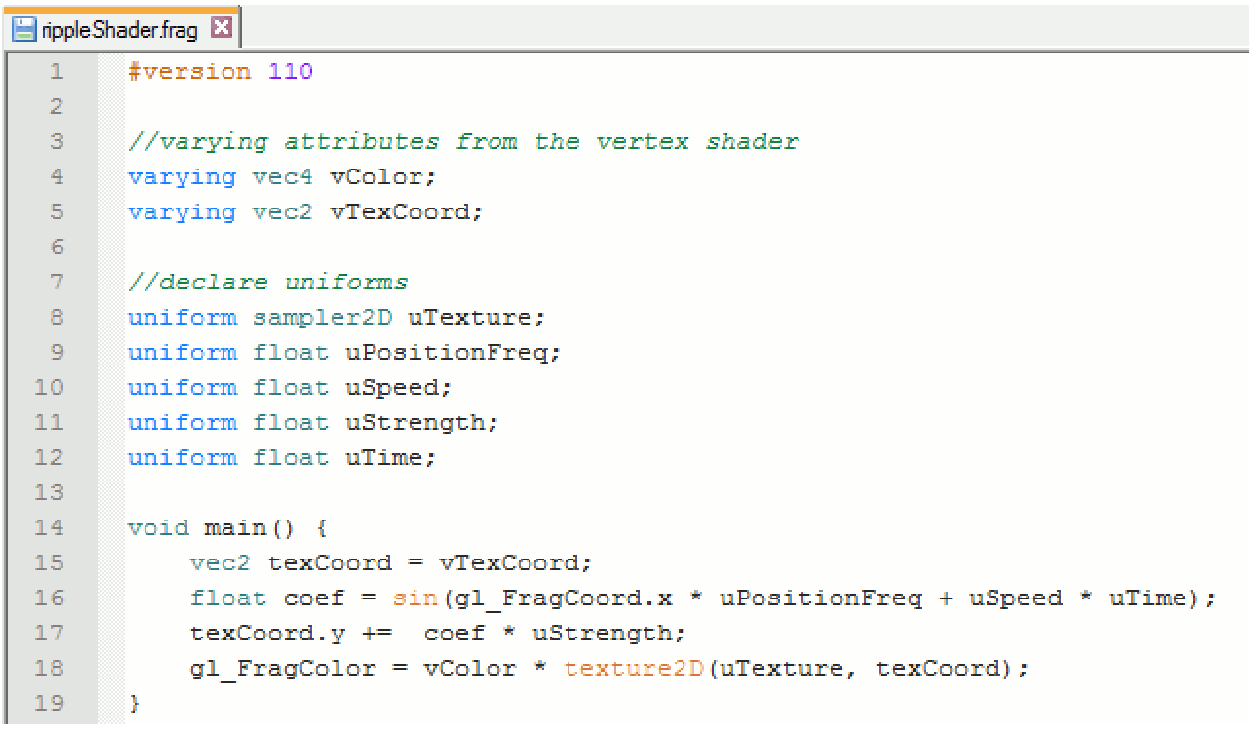
```
auto shader = AssetManager::GetShader("vertShader.vert", "fragShader.frag");

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));

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

    window.draw(sprite, shader);

    window.display();
}
```

```
shader->setParameter("uSpecialVector", sf::Vector2f(3, 3));
```



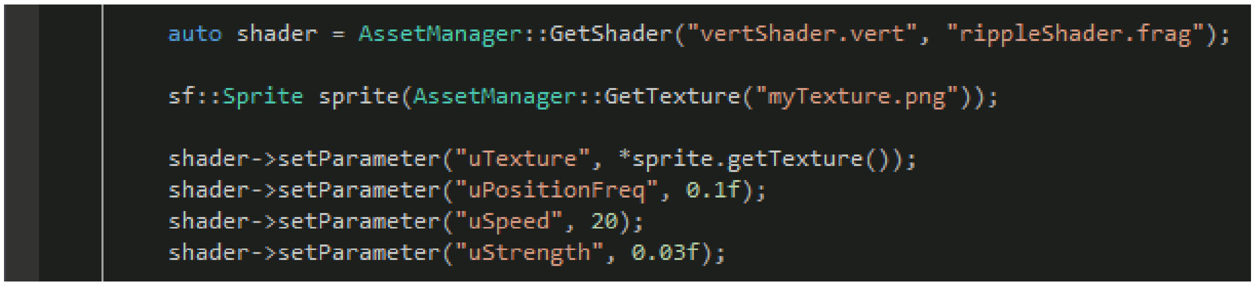
vertShader.vert

```
#version 110

//varying "out" variables to be used in the fragment shader
varying vec4 vColor;
varying vec2 vTexCoord;

void main() {
    vColor = gl_Color;
    vTexCoord = (gl_TextureMatrix[0] * gl_MultiTexCoord0).xy;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

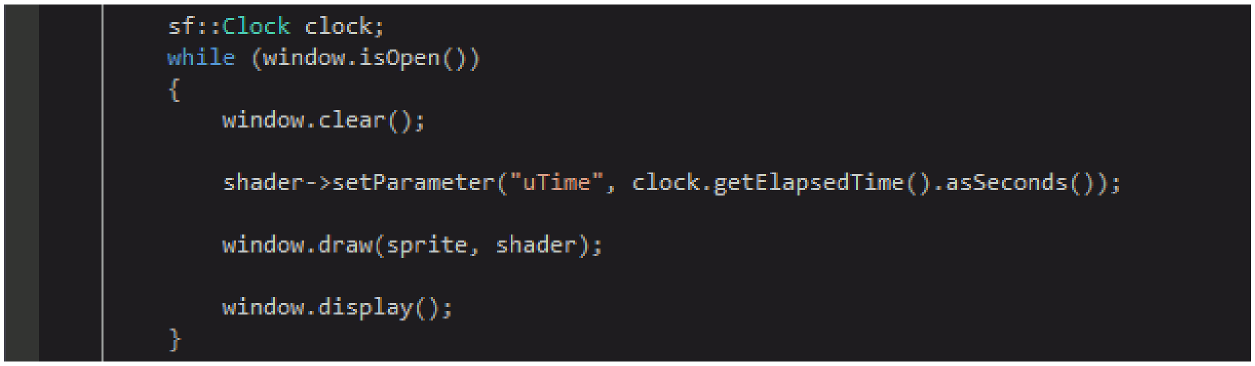


rippleShader.frag

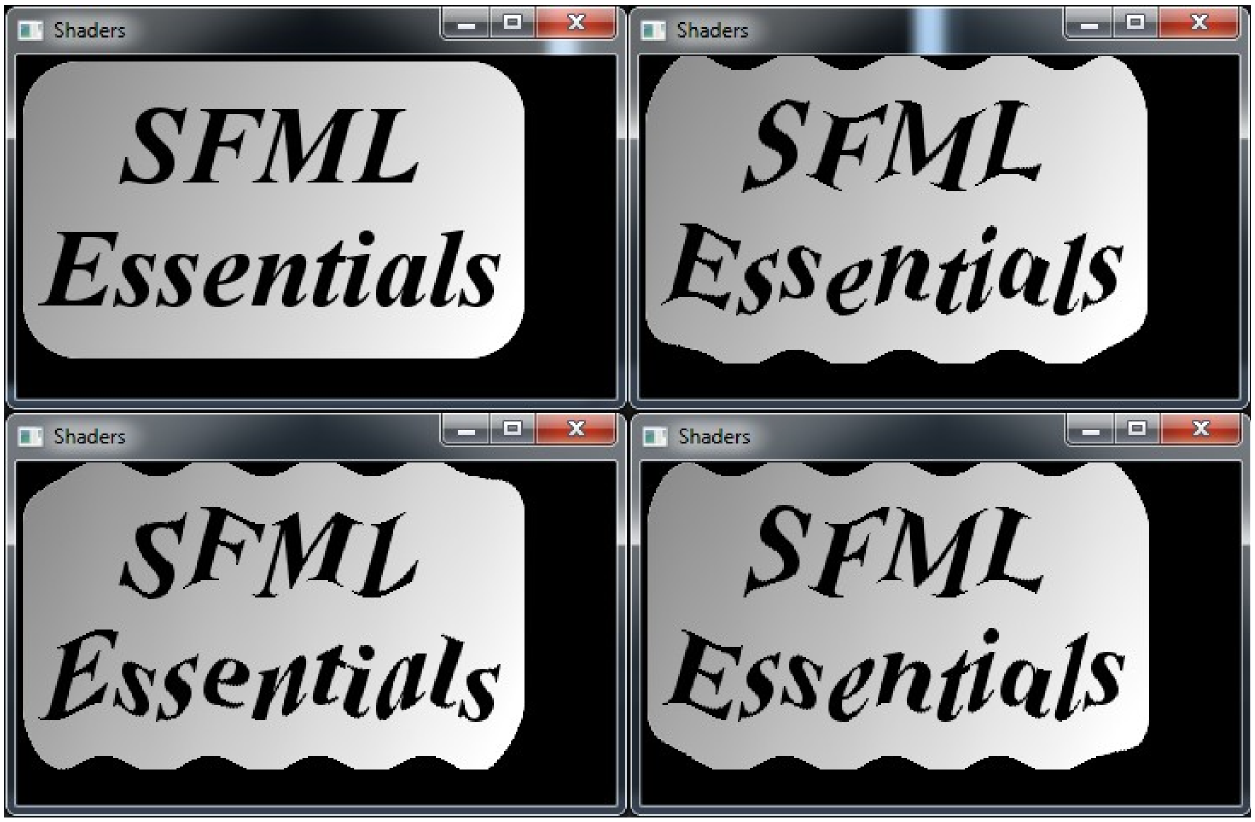
```
#version 110

//varying attributes from the vertex shader
varying vec4 vColor;
varying vec2 vTexCoord;

//declare uniforms
uniform sampler2D uTexture;
uniform float uPositionFreq;
uniform float uSpeed;
uniform float uStrength;
uniform float uTime;

void main() {
    vec2 texCoord = vTexCoord;
    float coef = sin(gl_FragCoord.x * uPositionFreq + uSpeed * uTime);
    texCoord.y +=  coef * uStrength;
    gl_FragColor = vColor * texture2D(uTexture, texCoord);
}
```

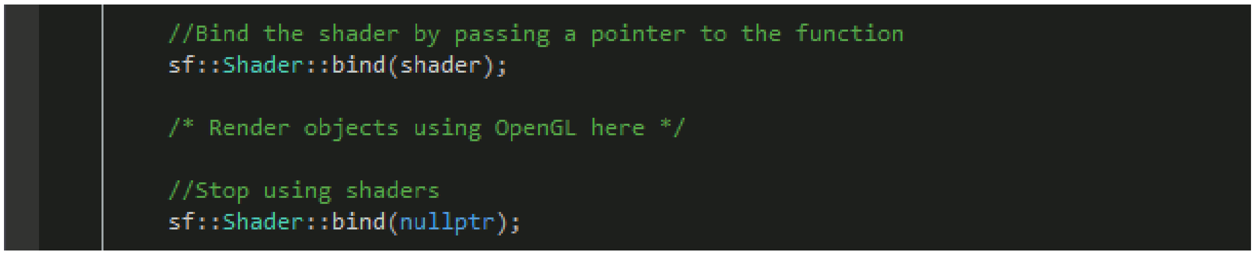


```
auto shader = AssetManager::GetShader("vertShader.vert", "rippleShader.frag");

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));

shader->setParameter("uTexture", *sprite.getTexture());
shader->setParameter("uPositionFreq", 0.1f);
shader->setParameter("uSpeed", 20);
shader->setParameter("uStrength", 0.03f);
```

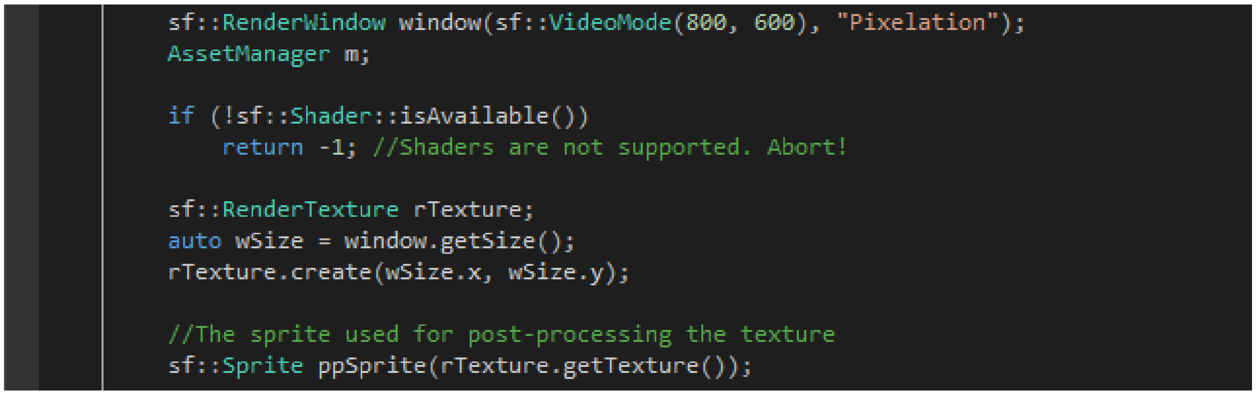


```
sf::Clock clock;
while (window.isOpen())
{
    window.clear();

    shader->setParameter("uTime", clock.getElapsedTime().asSeconds());

    window.draw(sprite, shader);

    window.display();
}
```

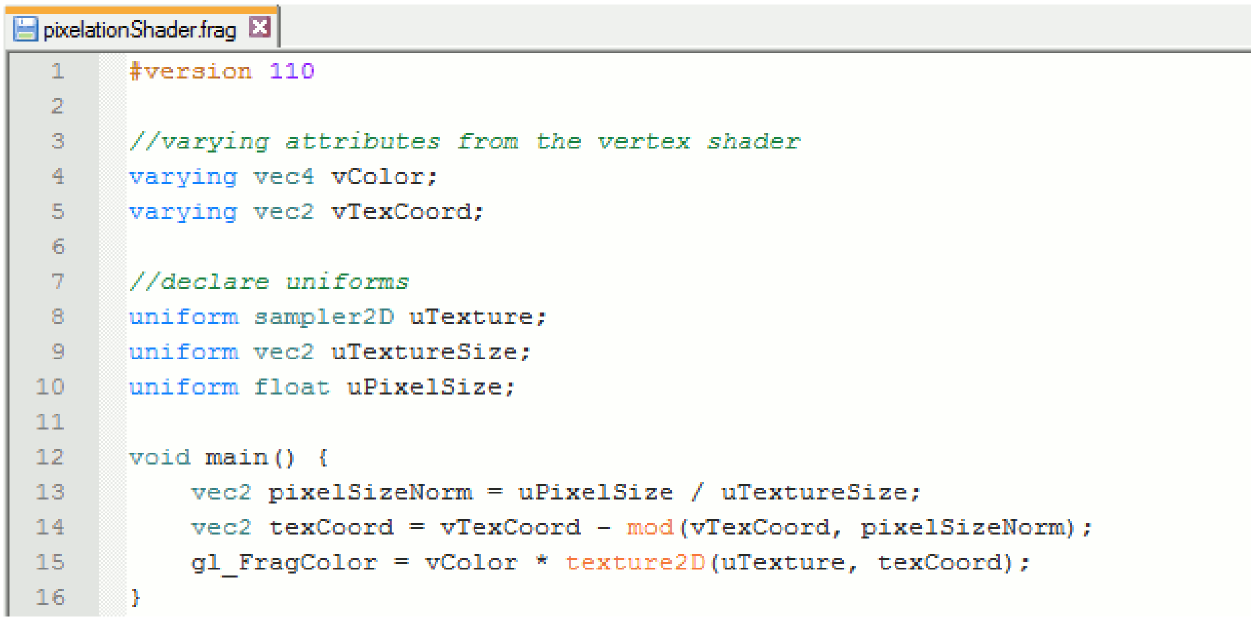




```
//Bind the shader by passing a pointer to the function
sf::Shader::bind(shader);

/* Render objects using OpenGL here */

//Stop using shaders
sf::Shader::bind(nullptr);
```

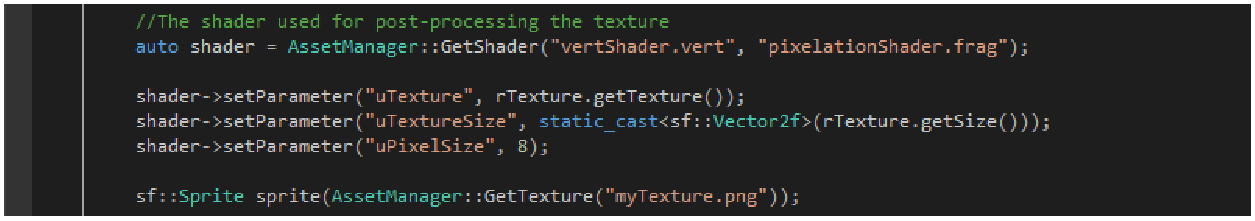


```
sf::RenderWindow window(sf::VideoMode(800, 600), "Pixelation");
AssetManager m;

if (!sf::Shader::isAvailable())
    return -1; //Shaders are not supported. Abort!

sf::RenderTexture rTexture;
auto wSize = window.getSize();
rTexture.create(wSize.x, wSize.y);

//The sprite used for post-processing the texture
sf::Sprite ppSprite(rTexture.getTexture());
```

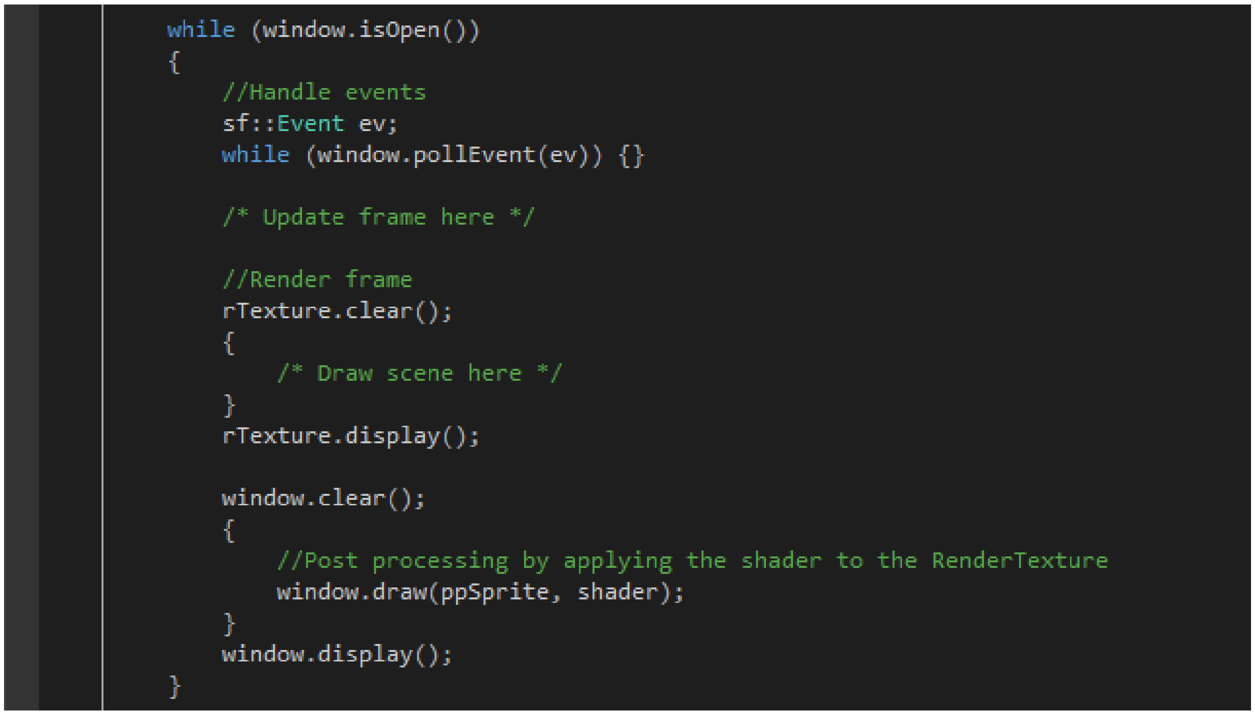


pixelationShader.frag

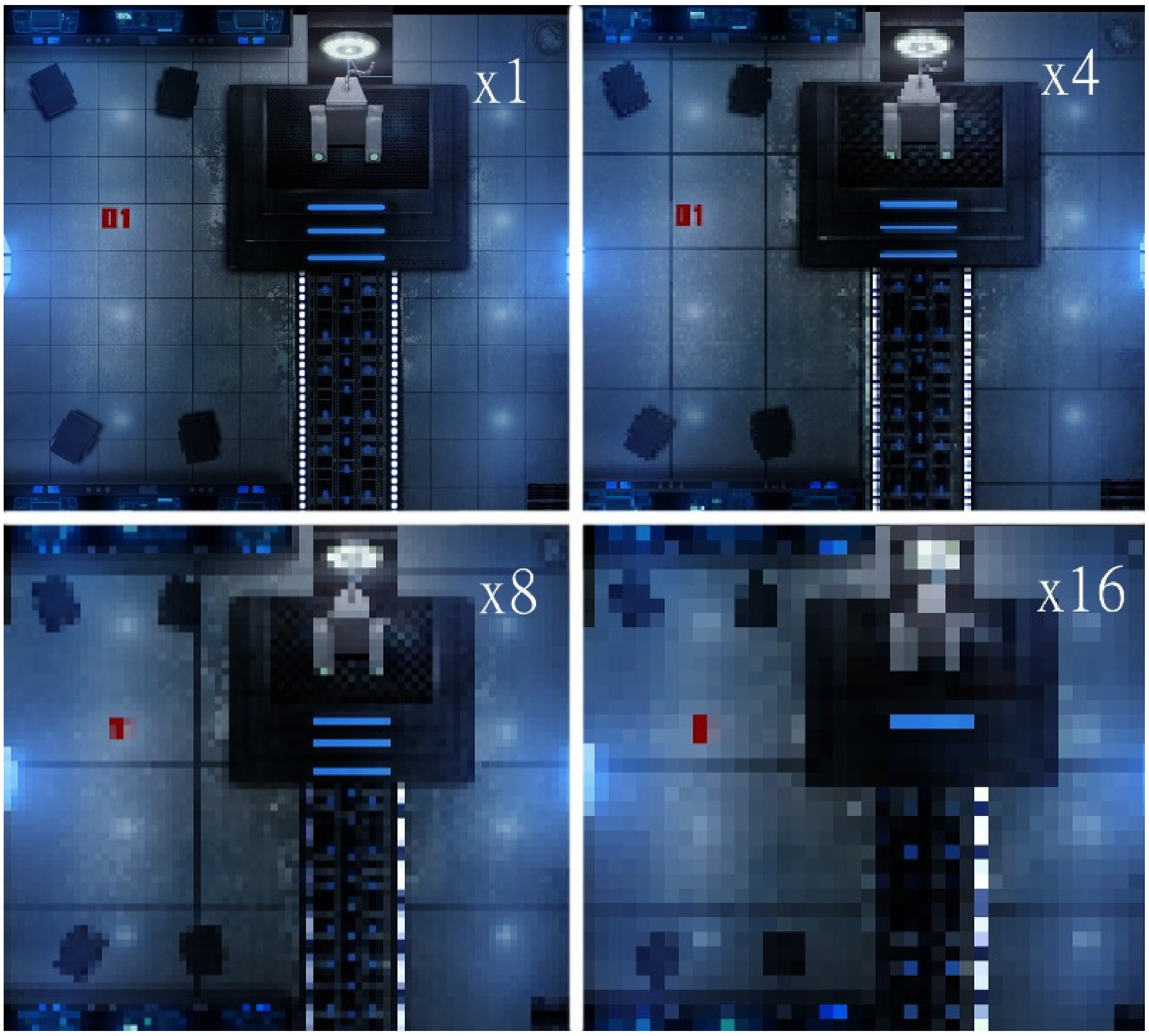
```
#version 110

//varying attributes from the vertex shader
varying vec4 vColor;
varying vec2 vTexCoord;

//declare uniforms
uniform sampler2D uTexture;
uniform vec2 uTextureSize;
uniform float uPixelSize;

void main() {
    vec2 pixelSizeNorm = uPixelSize / uTextureSize;
    vec2 texCoord = vTexCoord - mod(vTexCoord, pixelSizeNorm);
    gl_FragColor = vColor * texture2D(uTexture, texCoord);
}
```



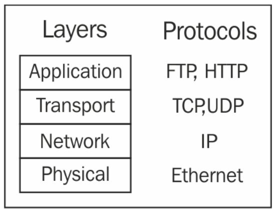
```
//The shader used for post-processing the texture
auto shader = AssetManager::GetShader("vertShader.vert", "pixelationShader.frag");

shader->setParameter("uTexture", rTexture.getTexture());
shader->setParameter("uTextureSize", static_cast<sf::Vector2f>(rTexture.getSize()));
shader->setParameter("uPixelSize", 8);

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));
```



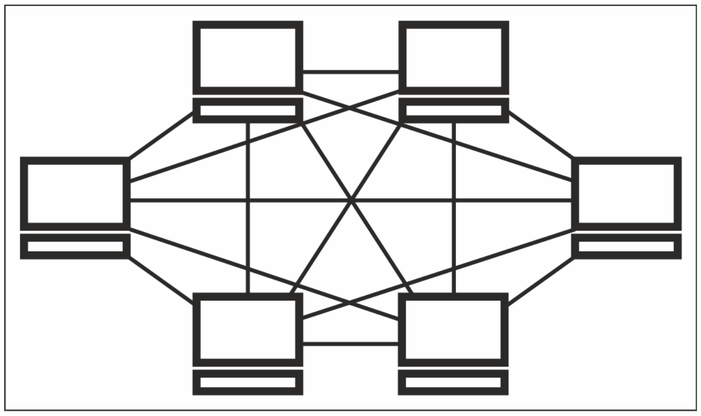
```
while (window.isOpen())
{
    //Handle events
    sf::Event ev;
    while (window.pollEvent(ev)) {}

    /* Update frame here */

    //Render frame
    rTexture.clear();
    {
        /* Draw scene here */
    }
    rTexture.display();

    window.clear();
    {
        //Post processing by applying the shader to the RenderTexture
        window.draw(ppSprite, shader);
    }
    window.display();
}
```

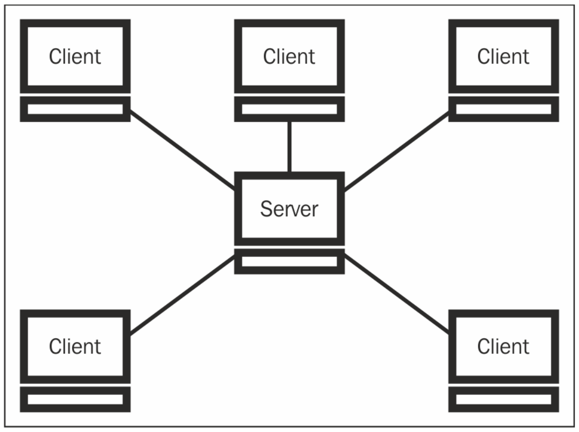

x1
x4
x8
x16
01

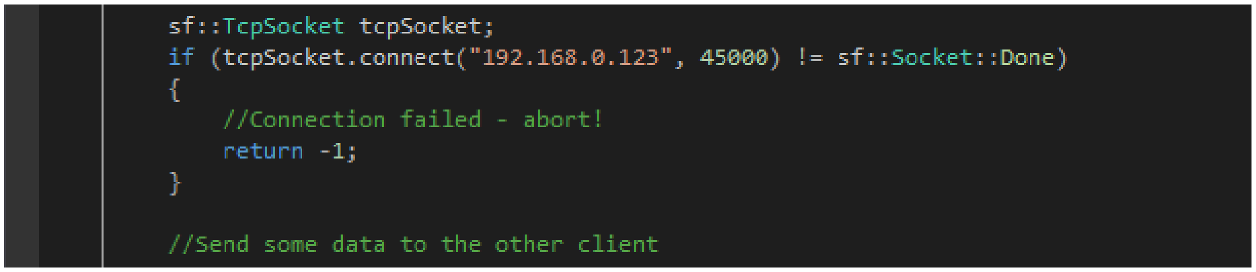
Chapter 7: Building Multiplayer Games



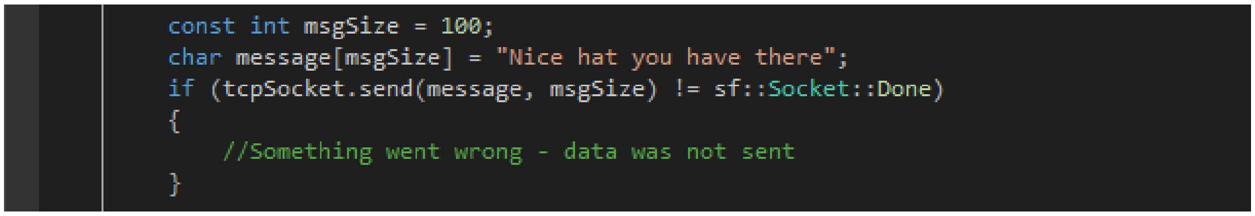
Layers	Protocols
Application	FTP, HTTP
Transport	TCP,UDP
Network	IP
Physical	Ethernet



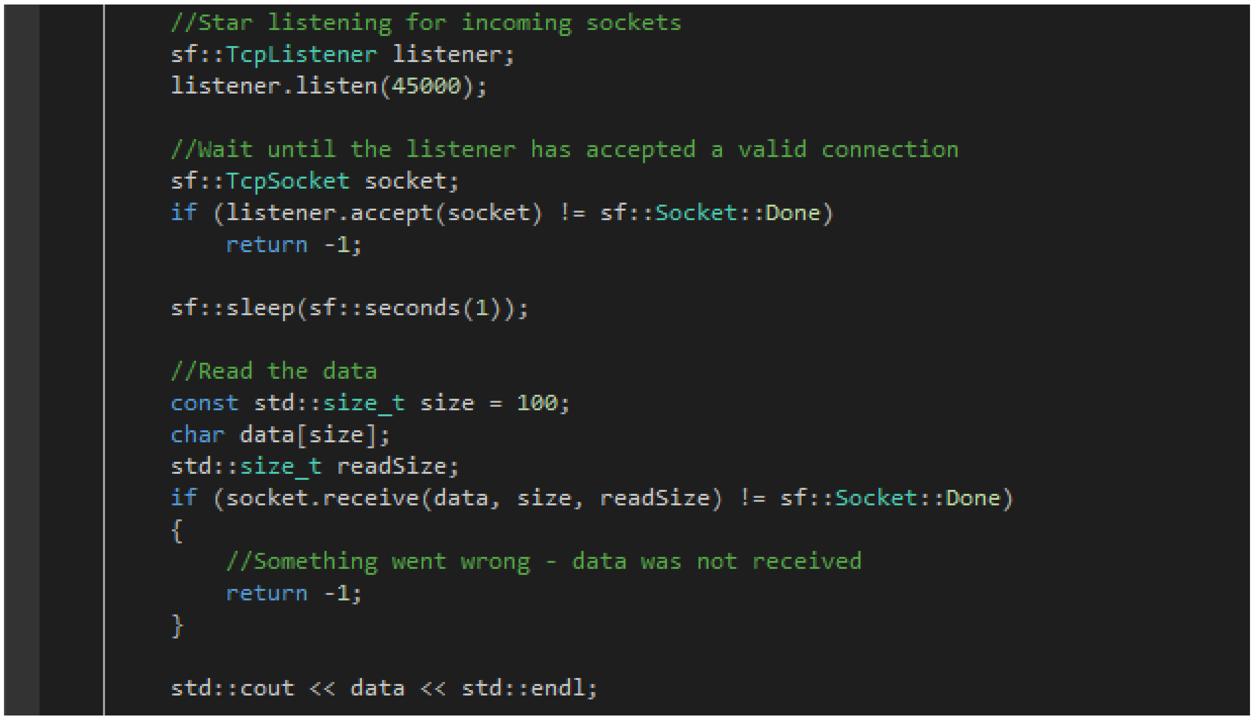




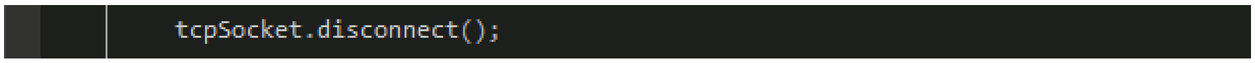
```
sf::TcpSocket tcpSocket;
if (tcpSocket.connect("192.168.0.123", 45000) != sf::Socket::Done)
{
    //Connection failed - abort!
    return -1;
}

//Send some data to the other client
```



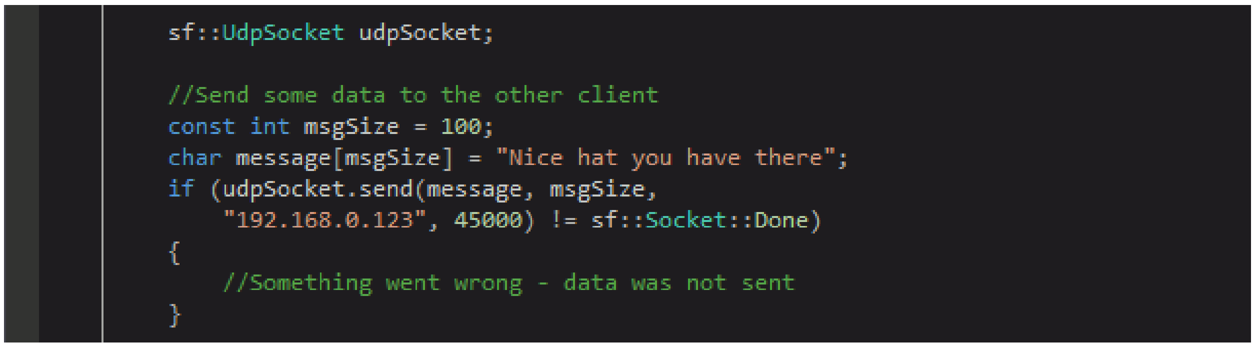
```
const int msgSize = 100;
char message[msgSize] = "Nice hat you have there";
if (tcpSocket.send(message, msgSize) != sf::Socket::Done)
{
    //Something went wrong - data was not sent
}
```



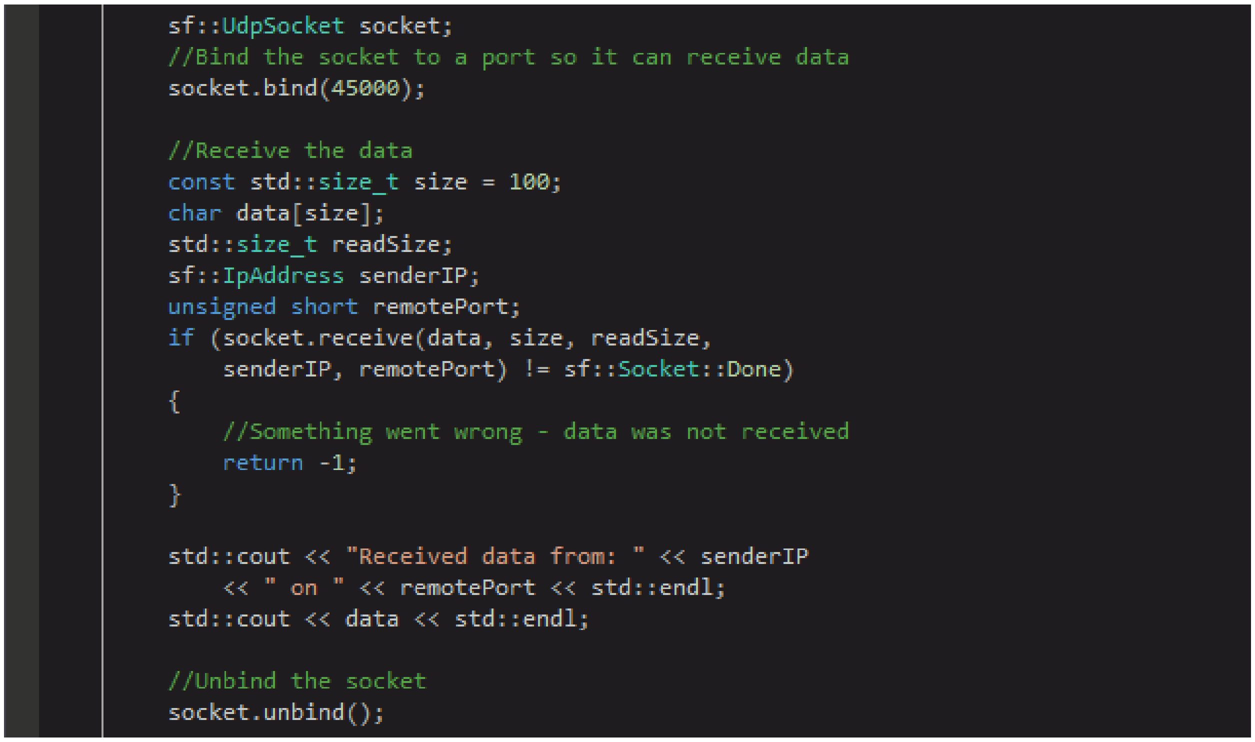
```
//Star listening for incoming sockets
sf::TcpListener listener;
listener.listen(45000);

//Wait until the listener has accepted a valid connection
sf::TcpSocket socket;
if (listener.accept(socket) != sf::Socket::Done)
    return -1;

sf::sleep(sf::seconds(1));

//Read the data
const std::size_t size = 100;
char data[size];
std::size_t readSize;
if (socket.receive(data, size, readSize) != sf::Socket::Done)
{
    //Something went wrong - data was not received
    return -1;
}

std::cout << data << std::endl;
```



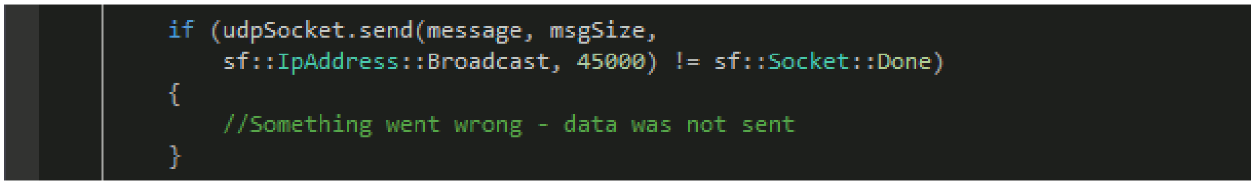
```
tcpSocket.disconnect();
```



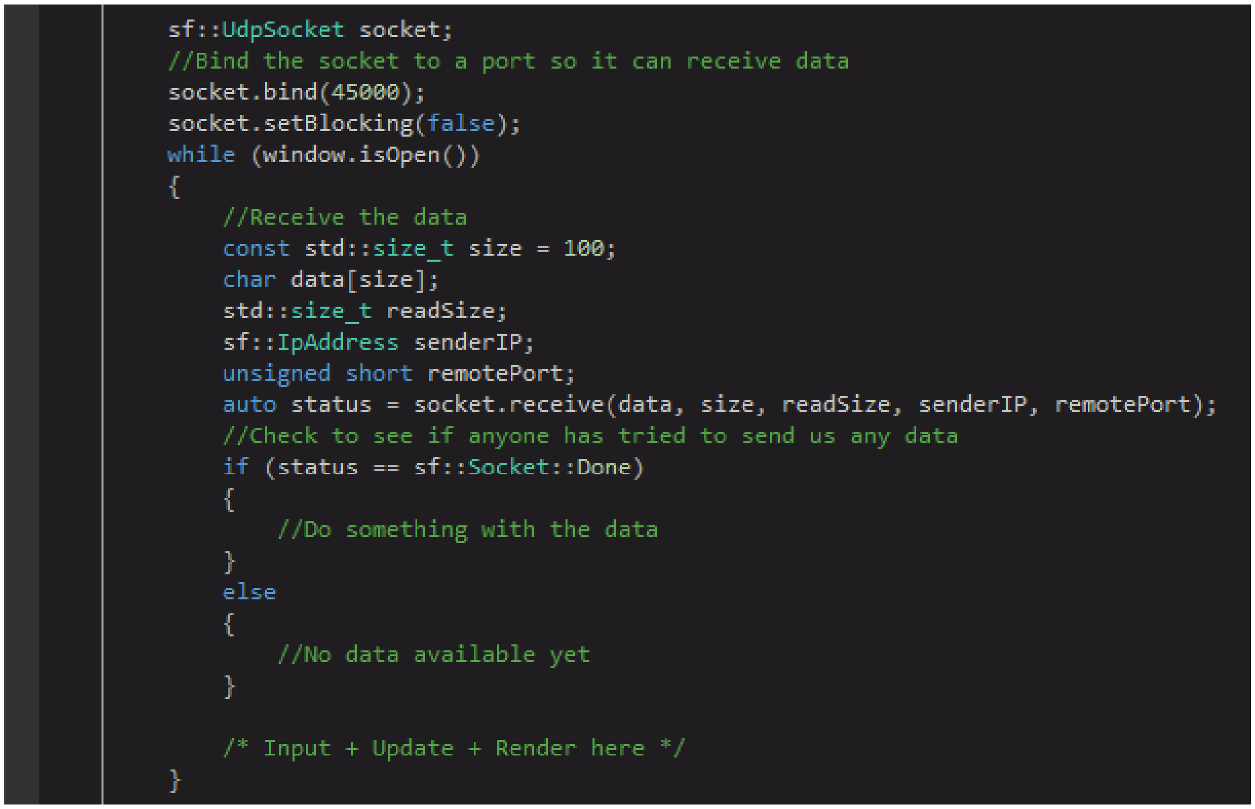
```
sf::UdpSocket udpSocket;

//Send some data to the other client
const int msgSize = 100;
char message[msgSize] = "Nice hat you have there";
if (udpSocket.send(message, msgSize,
    "192.168.0.123", 45000) != sf::Socket::Done)
{
    //Something went wrong - data was not sent
}
```



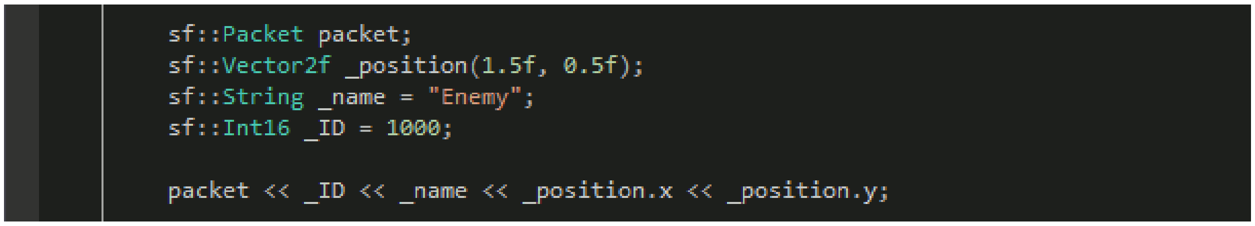
```cpp
sf::UdpSocket socket;
//Bind the socket to a port so it can receive data
socket.bind(45000);

//Receive the data
const std::size_t size = 100;
char data[size];
std::size_t readSize;
sf::IpAddress senderIP;
unsigned short remotePort;
if (socket.receive(data, size, readSize,
    senderIP, remotePort) != sf::Socket::Done)
{
    //Something went wrong - data was not received
    return -1;
}

std::cout << "Received data from: " << senderIP
    << " on " << remotePort << std::endl;
std::cout << data << std::endl;

//Unbind the socket
socket.unbind();
```



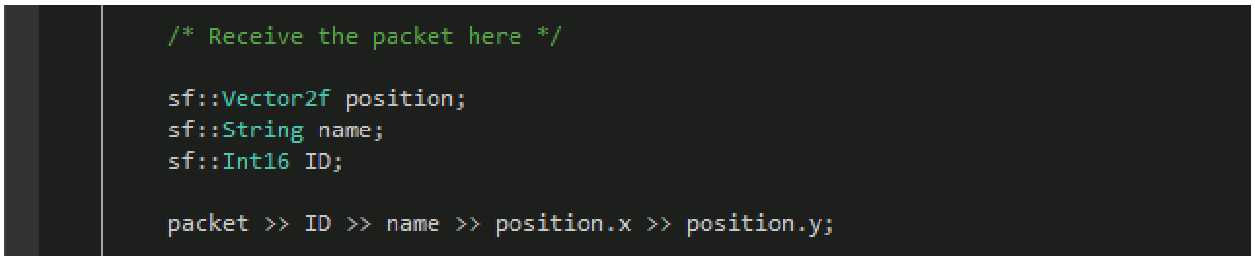
```cpp
if (udpSocket.send(message, msgSize,
    sf::IpAddress::Broadcast, 45000) != sf::Socket::Done)
{
    //Something went wrong - data was not sent
}
```



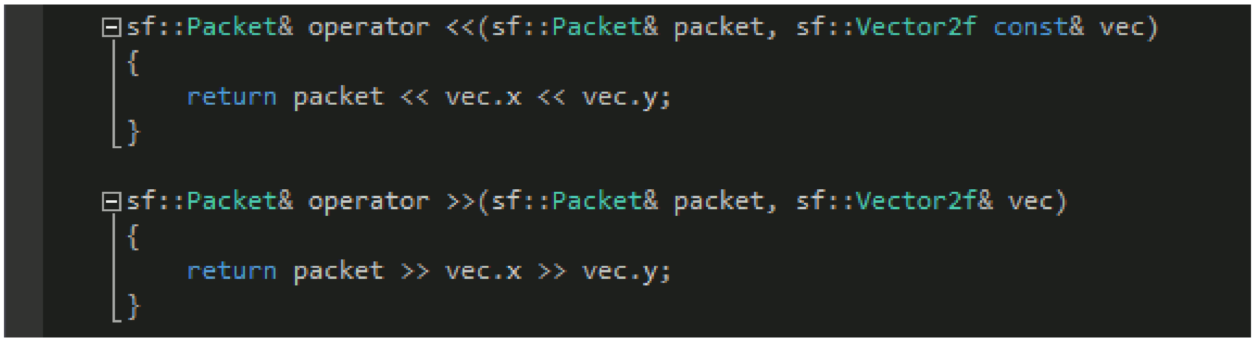
```cpp
sf::UdpSocket socket;
//Bind the socket to a port so it can receive data
socket.bind(45000);
socket.setBlocking(false);
while (window.isOpen())
{
    //Receive the data
    const std::size_t size = 100;
    char data[size];
    std::size_t readSize;
    sf::IpAddress senderIP;
    unsigned short remotePort;
    auto status = socket.receive(data, size, readSize, senderIP, remotePort);
    //Check to see if anyone has tried to send us any data
    if (status == sf::Socket::Done)
    {
        //Do something with the data
    }
    else
    {
        //No data available yet
    }

    /* Input + Update + Render here */
}
```



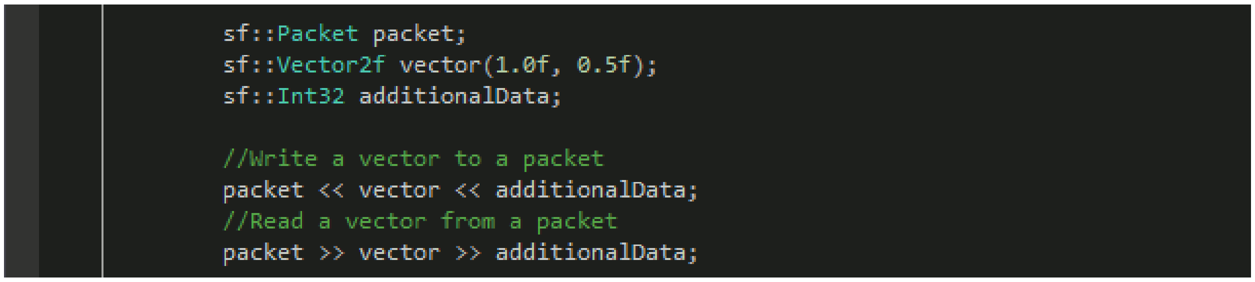
```cpp
sf::Packet packet;
sf::Vector2f _position(1.5f, 0.5f);
sf::String _name = "Enemy";
sf::Int16 _ID = 1000;

packet << _ID << _name << _position.x << _position.y;
```



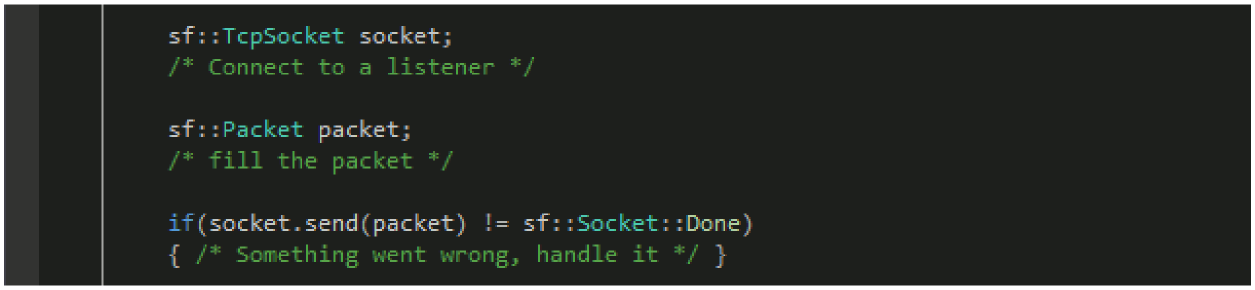
```cpp
/* Receive the packet here */

sf::Vector2f position;
sf::String name;
sf::Int16 ID;

packet >> ID >> name >> position.x >> position.y;
```



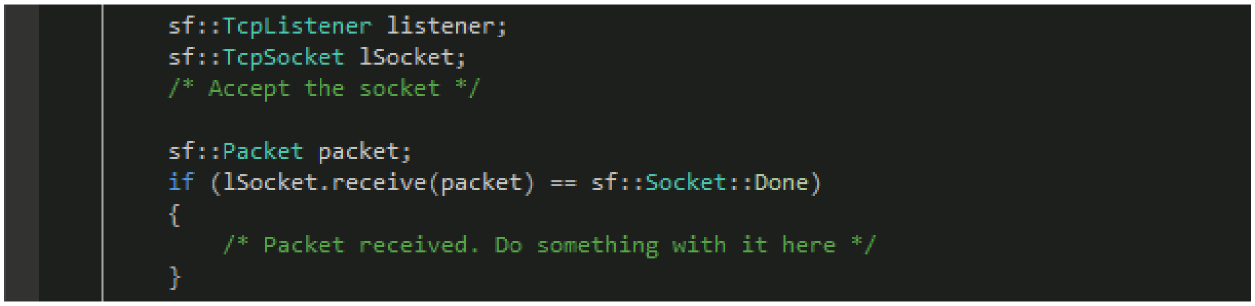
```cpp
sf::Packet& operator <<(sf::Packet& packet, sf::Vector2f const& vec)
{
    return packet << vec.x << vec.y;
}

sf::Packet& operator >>(sf::Packet& packet, sf::Vector2f& vec)
{
    return packet >> vec.x >> vec.y;
}
```



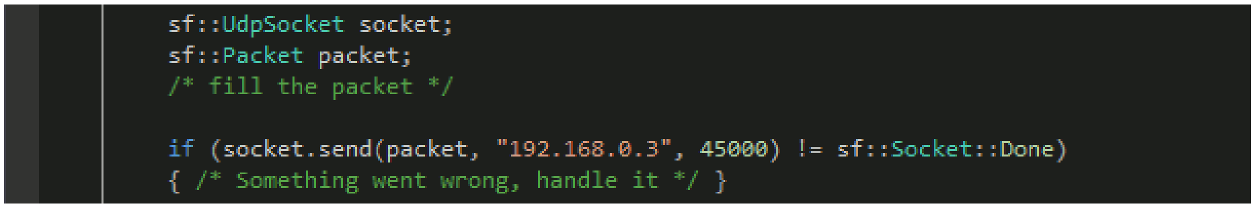
```cpp
    sf::Packet packet;
    sf::Vector2f vector(1.0f, 0.5f);
    sf::Int32 additionalData;

    //Write a vector to a packet
    packet << vector << additionalData;
    //Read a vector from a packet
    packet >> vector >> additionalData;
```



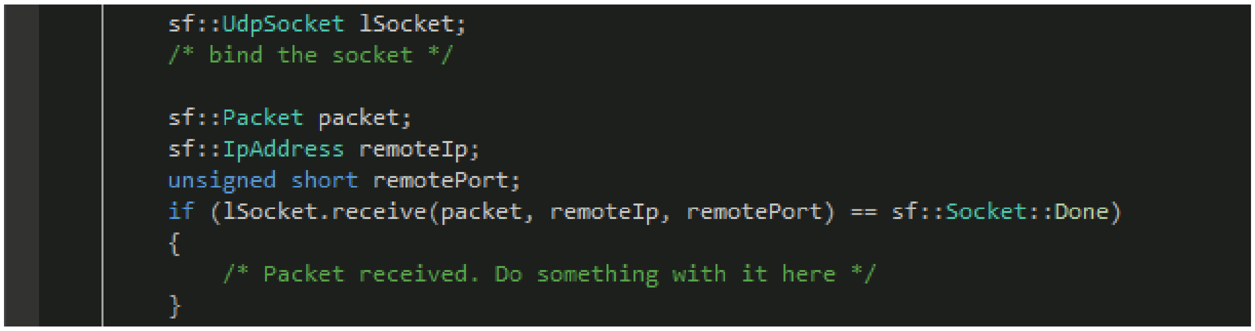
```cpp
sf::TcpSocket socket;
/* Connect to a listener */

sf::Packet packet;
/* fill the packet */

if(socket.send(packet) != sf::Socket::Done)
{ /* Something went wrong, handle it */ }
```



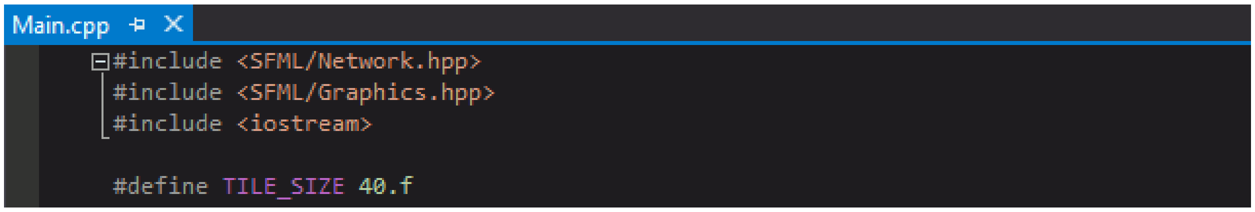
```cpp
sf::TcpListener listener;
sf::TcpSocket lSocket;
/* Accept the socket */

sf::Packet packet;
if (lSocket.receive(packet) == sf::Socket::Done)
{
    /* Packet received. Do something with it here */
}
```



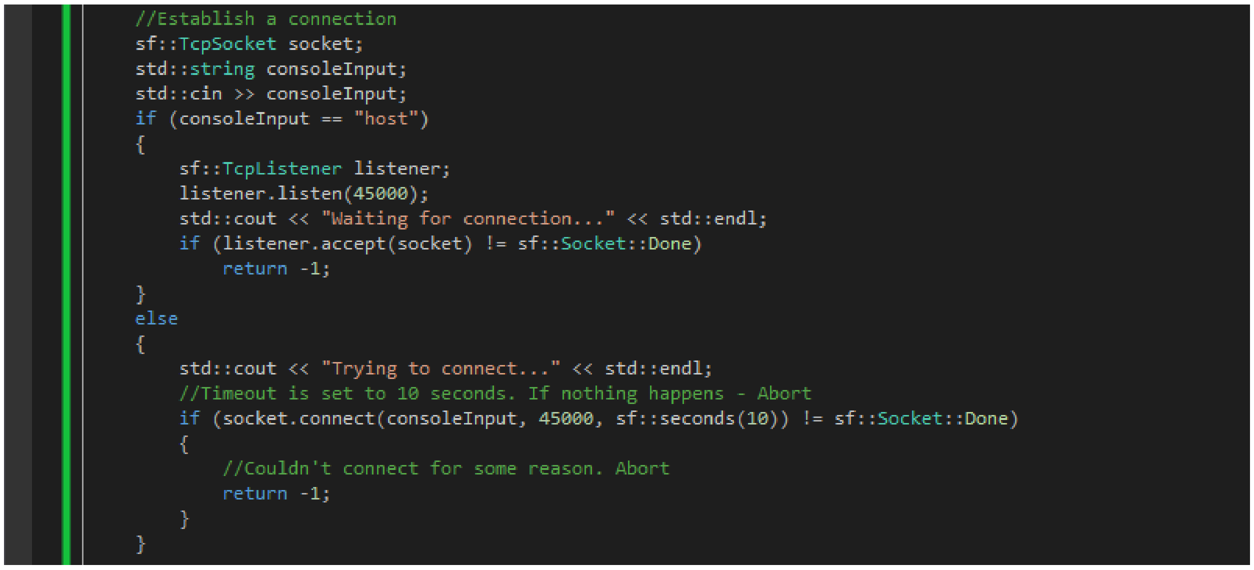
```cpp
sf::UdpSocket socket;
sf::Packet packet;
/* fill the packet */

if (socket.send(packet, "192.168.0.3", 45000) != sf::Socket::Done)
{ /* Something went wrong, handle it */ }
```



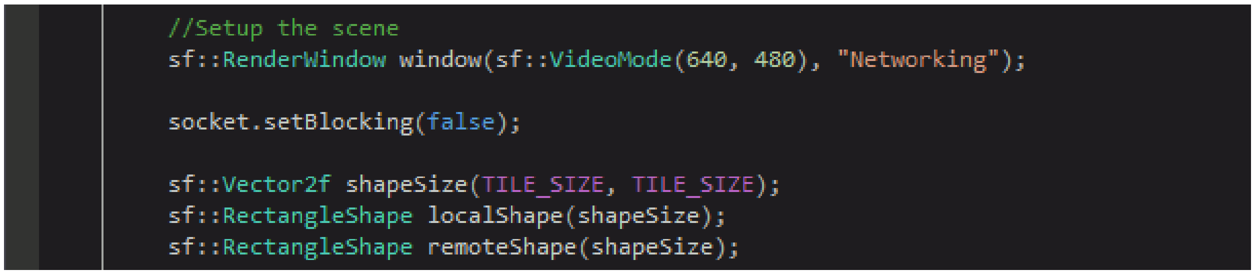
```cpp
sf::UdpSocket lSocket;
/* bind the socket */

sf::Packet packet;
sf::IpAddress remoteIp;
unsigned short remotePort;
if (lSocket.receive(packet, remoteIp, remotePort) == sf::Socket::Done)
{
    /* Packet received. Do something with it here */
}
```


Main.cpp

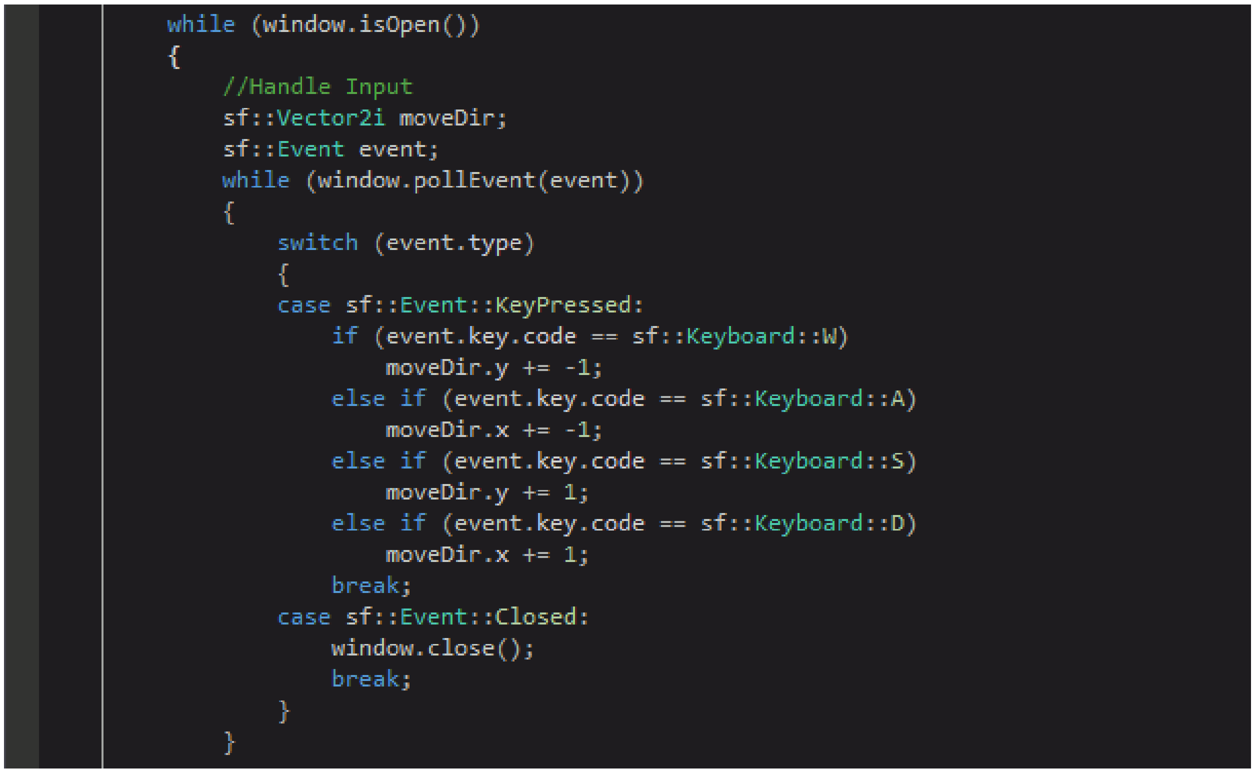
```
#include <SFML/Network.hpp>
#include <SFML/Graphics.hpp>
#include <iostream>

#define TILE_SIZE 40.f
```



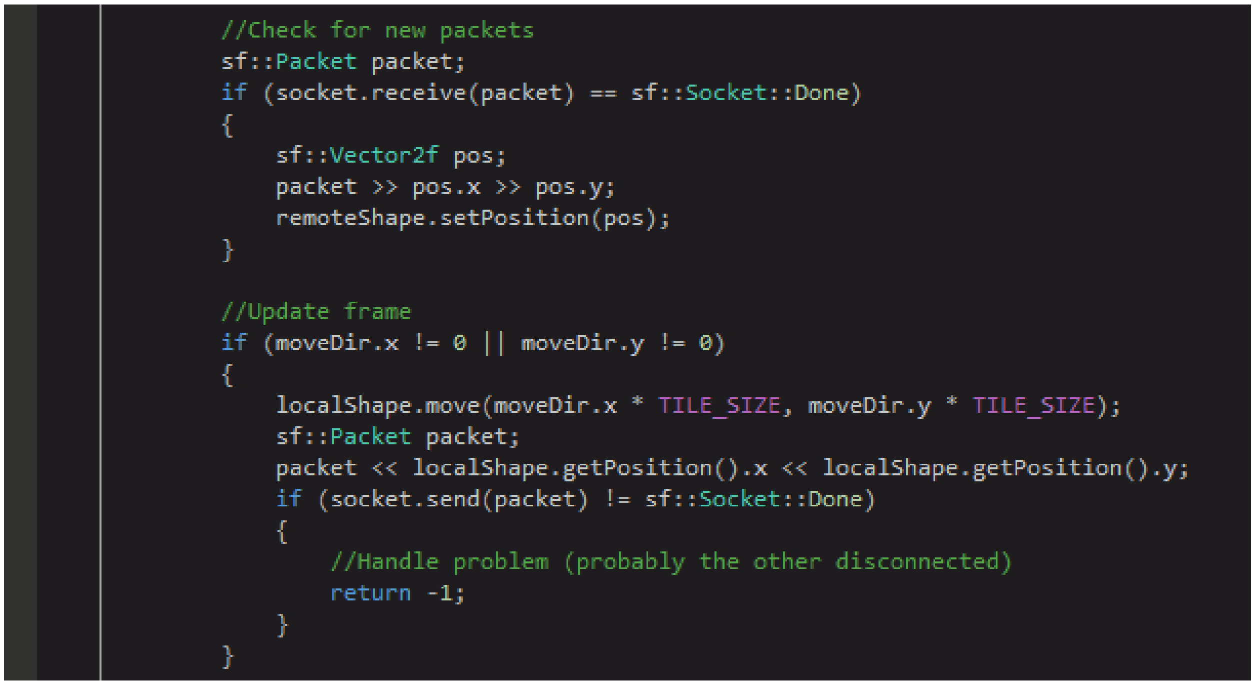
```
    //Establish a connection
    sf::TcpSocket socket;
    std::string consoleInput;
    std::cin >> consoleInput;
    if (consoleInput == "host")
    {
        sf::TcpListener listener;
        listener.listen(45000);
        std::cout << "Waiting for connection..." << std::endl;
        if (listener.accept(socket) != sf::Socket::Done)
            return -1;
    }
    else
    {
        std::cout << "Trying to connect..." << std::endl;
        //Timeout is set to 10 seconds. If nothing happens - Abort
        if (socket.connect(consoleInput, 45000, sf::seconds(10)) != sf::Socket::Done)
        {
            //Couldn't connect for some reason. Abort
            return -1;
        }
    }
```



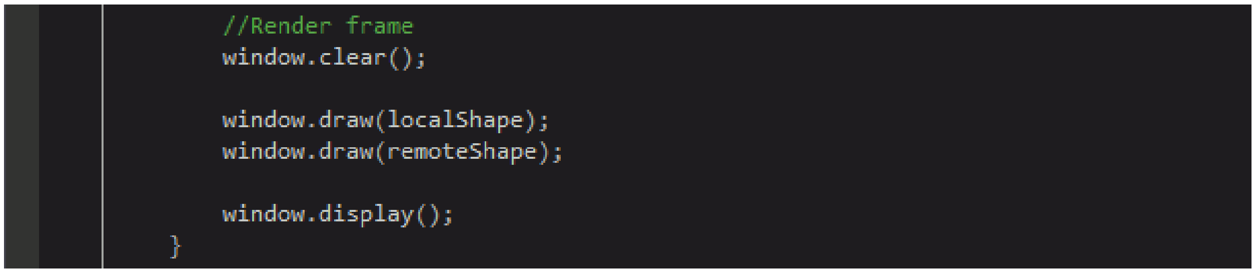
```
    //Setup the scene
    sf::RenderWindow window(sf::VideoMode(640, 480), "Networking");

    socket.setBlocking(false);

    sf::Vector2f shapeSize(TILE_SIZE, TILE_SIZE);
    sf::RectangleShape localShape(shapeSize);
    sf::RectangleShape remoteShape(shapeSize);
```



```
    while (window.isOpen())
    {
        //Handle Input
        sf::Vector2i moveDir;
        sf::Event event;
        while (window.pollEvent(event))
        {
            switch (event.type)
            {
            case sf::Event::KeyPressed:
                if (event.key.code == sf::Keyboard::W)
                    moveDir.y += -1;
                else if (event.key.code == sf::Keyboard::A)
                    moveDir.x += -1;
                else if (event.key.code == sf::Keyboard::S)
                    moveDir.y += 1;
                else if (event.key.code == sf::Keyboard::D)
                    moveDir.x += 1;
                break;
            case sf::Event::Closed:
                window.close();
                break;
            }
        }
```



```
        //Check for new packets
        sf::Packet packet;
        if (socket.receive(packet) == sf::Socket::Done)
        {
            sf::Vector2f pos;
            packet >> pos.x >> pos.y;
            remoteShape.setPosition(pos);
        }

        //Update frame
        if (moveDir.x != 0 || moveDir.y != 0)
        {
            localShape.move(moveDir.x * TILE_SIZE, moveDir.y * TILE_SIZE);
            sf::Packet packet;
            packet << localShape.getPosition().x << localShape.getPosition().y;
            if (socket.send(packet) != sf::Socket::Done)
            {
                //Handle problem (probably the other disconnected)
                return -1;
            }
        }
```



```
        //Render frame
        window.clear();

        window.draw(localShape);
        window.draw(remoteShape);

        window.display();
    }
```