Migration Guide

Welcome to SFML 3! The SFML Team has put a lot of effort into delivering a library that is both familiar to existing users while also making significant improvements. This document will walk you through how to upgrade your SFML 2 application to SFML 3.

One of the headline changes in SFML 3 is raising the C++ standard to C++17 thus bringing SFML into the world of modern C++! This change has enabled a vast number of internal improvements as well as new opportunities for improving the API that will be discussed in this document.

Compiler Requirements

Along with the upgrade from C++03 to C++17 you may need to upgrade your compiler. C++17 support has been widespread in all major compiler implementations for years prior to SFML 3's release so in all likelihood the compiler you're already using will work. In case you do need to upgrade, here are the minimum compiler versions.

Compiler	**Version**
MSVC	16 (VS 2019)
GCC	9
Clang	9
AppleClang	12

CMake Targets

SFML 3 uses modern CMake convention for library targets which entails having a namespace in front of the target name. These namespaces tell CMake "this is a target" whereas something like `sfml-graphics` might be a target or might be a precompiled library on disk like `libsfml-graphics.so`. Fixing this ambiguity leads to more useful error messages when a given target can't be found due to, for example, forgetting a `find_package` call. The component names used when calling `find_package` were also changed to capitalize the first letter.

v2 Target	v3 Target
`sfml-system`	`SFML::System`
`sfml-window`	`SFML::Window`
`sfml-graphics`	`SFML::Graphics`
`sfml-network`	`SFML::Network`
`sfml-audio`	`SFML::Audio`
`sfml-main`	`SFML::Main`

v2 Component	v3 Component
`system`	`System`
`window`	`Window`
`graphics`	`Graphics`
`network`	`Network`
`audio`	`Audio`
`main`	`Main`

v2:

```
find_package(SFML 2 REQUIRED COMPONENTS graphics audio network)
...
target_link_libraries(my_app PRIVATE sfml-graphics sfml-audio sfml-network)
```

v3:

```
find_package(SFML 3 REQUIRED COMPONENTS Graphics Audio Network)
...
target_link_libraries(my_app PRIVATE SFML::Graphics SFML::Audio SFML::Network)
```

Linux Dependencies

When using X11 as the backend on Linux, as opposed to DRM, `libxi-dev` is a newly required dependency. This was introduced with the raw mouse input support.

`sf::Vector2<T>` Parameters

A common pattern in SFML 2 was to use pairs of scalar parameters to represent concepts like sizes or positions. Take `sf::Transformable::setPosition(float, float)` for example. The two parameters combine to represent a position in world space.

SFML 3 takes all of the APIs with pairs of parameters like `(float, float)` or `(unsigned int, unsigned int)` and converts them to their corresponding `sf::Vector2<T>` type like `sf::Vector2f` or `sf::Vector2u` to make the interface more expressive and composable. This transition is often as simple as wrapping the two adjacent parameters with braces to construct the vector.

v2:

```
sf::VideoMode videoMode(480, 640, 24);
sf::CircleShape circle(10);
circle.setPosition(10, 20);
sf::IntRect rect(250, 400, 50, 100);
```

v3:

```
sf::VideoMode videoMode({480, 640}, 24);
sf::CircleShape circle(10);
circle.setPosition({10, 20});
sf::IntRect rect({250, 400}, {50, 100});
```

Fixed Width Integers

SFML 2 contained various typedefs for fixed width integers. Those are now replaced with the fixed width integers provided in the `<cstdint>` header from the standard library.

v2	v3
`sf::Int8`	`std::int8_t`
`sf::Uint8`	`std::uint8_t`
`sf::Int16`	`std::int16_t`
`sf::Uint16`	`std::uint16_t`
`sf::Int32`	`std::int32_t`
`sf::Uint32`	`std::uint32_t`
`sf::Int64`	`std::int64_t`
`sf::Uint64`	`std::uint64_t`

`sf::Event`

SFML 3 uses `std::variant` under the hood to implement a totally new, type-safe API for events. There are two main ways to use this new API. Check out the new EventHandling example program to see these methods in practice.

`sf::Event::getIf<T>`

The first option is based around `sf::Event::getIf<T>` and `sf::Event::is<T>`. `getIf<T>` works by providing a template parameter which must be an event subtype. Event subtypes are types like `sf::Event::Resized` or `sf::Event::MouseMoved`. If the template argument matches the active event subtype then a pointer to the data is returned. If that template argument is not the active event subtype then `nullptr` is returned. `sf::Event::is<T>` more simply returns `true` if `T` matches the active event subtype. `is<T>` is often used for subtypes like `sf::Event::Closed` which contain no data. Here's what that looks like:

```
while (window.isOpen())
{
    while (const std::optional event = window.pollEvent())
    {
        if (event->is<sf::Event::Closed>())
        {
            window.close();
        }
        else if (const auto* keyPressed = event-
>getIf<sf::Event::KeyPressed>())
        {
            if (keyPressed->scancode == sf::Keyboard::Scancode::Escape)
                window.close();
        }
    }

    // Remainder of main loop
}
```

Note how the API for getting events has changed slightly. `sf::WindowBase::pollEvent` and `sf::WindowBase::waitEvent` now return a `std::optional<sf::Event>`. These two functions *might* return an event but they might not. C++ lets you deduce the template parameter which is why you can write `const std::optional event` instead of `const std::optional<sf::Event> event`. `const auto event` is another valid choice if you prefer a more terse expression.

`sf::WindowBase::handleEvents`

The second option for processing events is via the new `sf::WindowBase::handleEvents` function. This functions performs event visitation. What this means is that you can provide callbacks which take different event subtypes as arguments. Alternatively you may provide an object (or objects) with `operator()` implementations which handle the event subtypes you want to process. Notably you do not have to provide callbacks for all possible event subtypes. Depending on the current active event subtype, the corresponding callback is called. Here's what that looks like:

```
const auto onClose = [&window](const sf::Event::Closed&)
{
    window.close();
};

const auto onKeyPressed = [&window](const sf::Event::KeyPressed& keyPressed)
{
    if (keyPressed.scancode == sf::Keyboard::Scancode::Escape)
        window.close();
};

while (window.isOpen())
{
    window.handleEvents(onClose, onKeyPressed);

    // Remainder of main loop
}
```

Window Styles and States

A new `sf::State` enumeration was added for specifying the state of the window which means whether the window is floating or fullscreen. Here's a before-and-after showing how this affects constructing a window.

v2:

```
sf::RenderWindow window(sf::VideoMode::getFullscreenModes().at(0), "Title",
sf::Style::Fullscreen);
```

v3:

```
sf::RenderWindow window(sf::VideoMode::getFullscreenModes().at(0), "Title",
sf::State::Fullscreen);
```

Scoped Enumerations

SFML 3 converts all old style unscoped enumerations to scoped enumerations. This improves the type safety of the interface. This means that the name of the enumeration is now part of the namespace required to access values of that enumeration.

For example, take the enumeration `sf::Keyboard::Key`. `sf::Keyboard::A` becomes `sf::Keyboard::Key::A`. The name of the enumeration now appears as a nested namespace when accessing one of the enumeration's values.

Here is a complete list of all enumerations which have undergone this change:

- `sf::BlendMode::Equation`
- `sf::BlendMode::Factor`
- `sf::Cursor::Type`
- `sf::Ftp::Response::Status`
- `sf::Ftp::TransferMode`
- `sf::Http::Request::Method`
- `sf::Http::Response::Status`
- `sf::Joystick::Axis`
- `sf::Keyboard::Key`
- `sf::Keyboard::Scan`
- `sf::Mouse::Button`
- `sf::Mouse::Wheel`
- `sf::PrimitiveType`
- `sf::Sensor::Type`
- `sf::Shader::Type`
- `sf::Socket::Status`
- `sf::Socket::Type`
- `sf::SoundSource::Status`
- `sf::VertexBuffer::Usage`

`sf::Rect<T>`

`sf::Rect<T>` has been refactored from the four scalar values `top`, `left`, `width`, and `height` into two `sf::Vector2<T>`s named `position` and `size`. This means that `sf::Rect<T>::getPosition()` and `sf::Rect<T>::getSize()` have been removed in favor of directly accessing the `position` and `size` data members. The 4-parameter constructor was also removed in favor of the constructor which takes two `sf::Vector2<T>`s.

v2	v3
`.left`	`.position.x`

v2	v3
`.top`	`.position.y`
`.width`	`.size.x`
`.height`	`.size.y`

v2:

```
sf::FloatRect rect(10, 20, 30, 40);
sf::Vector2f position = rect.getPosition();
sf::Vector2f size = rect.getSize();
```

v3:

```
sf::FloatRect rect({10, 20}, {30, 40});
sf::Vector2f position = rect.position;
sf::Vectro2f size = rect.size;
```

The two overloads of `sf::Rect<T>::intersects` have been replaced with one unified function called `sf::Rect<T>::findIntersection` which returns a `std::optional<Rect<T>>`. This optional contains the overlapping area if the rectangles overlap. Otherwise the optional is empty.

```
sf::IntRect rect1({0, 0}, {200, 200});
sf::IntRect rect2({100, 100}, {200, 200});
std::optional<sf::IntRect> intersection = rect1.findIntersection(rect2);
// position={100, 100} size={100, 100}
```

`sf::Angle`

All angles are now represented with a strong type named `sf::Angle`. This type provides two functions for creating angles called `sf::degrees(float)` and `sf::radians(float)` which construct an angle from either some value of degrees or radians. Operators (`+`, `-`, etc.) are provided to perform mathematical operations with angles. If you need access to the raw angle as a `float` then you can use either `sf::Angle::asDegrees()` or `sf::Angle::asRadians()`.

v2:

```
sf::RectangleShape shape(sf::Vector2f(50, 50));
shape.setRotation(90);
std::cout << "Rotation: " << shape.getRotation() << '\n';
```

v3:

```
sf::RectangleShape shape({50, 50});
shape.setRotation(sf::degrees(90));
std::cout << "Rotation: " << shape.getRotation().asDegrees() << '\n';
```

Renamed Functions

A number of functions have new names but otherwise have not changed their semantics.

v2	v3
`sf::Font::loadFromFile`	`sf::Font::openFromFile`
`sf::Socket::getHandle`	`sf::Socket::getNativeHandle`
`sf::WindowBase::getSystemHandle`	`sf::WindowBase::getNativeHandle`
`sf::Texture::create`	`sf::Texture::resize`
`sf::RenderTexture::create`	`sf::RenderTexture::resize`
`sf::Image::create`	`sf::Image::resize`
`sf::Sound::getLoop`	`sf::Sound::isLooping`
`sf::Sound::setLoop`	`sf::Sound::setLooping`
`sf::SoundStream::getLoop`	`sf::SoundStream::isLooping`
`sf::SoundStream::setLoop`	`sf::SoundStream::setLooping`

Removal of Default Constructors

The default constructors `sf::Sound::Sound()`, `sf::Text::Text()`, and `sf::Sprite::Sprite()` were removed. They can be replaced by the corresponding constructors which accept a resource type.

v2	v3
`sf::Sound::Sound()`	`sf::Sound::Sound(const sf::SoundBuffer&)`
`sf::Text::Text()`	`sf::Text::Text(const sf::Font&)`
`sf::Sprite::Sprite()`	`sf::Sprite::Sprite(const sf::Texture&)`

Now that these classes are guaranteed to be holding a reference to their corresponding resource type, the functions used to access to those resources can return a reference instead of a pointer. These functions are `sf::Sound::getBuffer()`, `sf::Text::getFont()`, and `sf::Sprite::getTexture()`.

v2:

```
const sf::SoundBuffer soundBuffer("sound.flac");
sf::Sound sound;
sound.setBuffer(soundBuffer);
```

v3:

```
const sf::SoundBuffer soundBuffer("sound.flac");
sf::Sound sound(soundBuffer);
```

`std::optional` Usage

SFML 3 makes liberal use of `std::optional` to express when a given function may not return a value. Some of these usages have already been mentioned like `sf::WindowBase::pollEvent`. Here are some more places where SFML 3 makes use of `std::optional`.

- `sf::IpAddress` uses `std::optional` to express how resolving an address from a string may not yield a usable IP address.
- `sf::Image::saveToMemory` returns a `std::optional` because the `sf::Image` may be empty or the underlying implementation may fail.
- `sf::SoundFileReader::open` returns a `std::optional` because the stream being opened

may not be valid.

- `sf::Music::onLoop` and `sf::SoundStream::onLoop` returns a `std::optional` because if the objects are not in a looping state then there is nothing to return.
- `sf::InputStream` uses `std::optional` in various places. Instead of returning `-1` to signal an error, `std::nullopt` can be returned.

LearnCpp.com is a great place to learn more about using `std::optional`. Read more about that here.

New Constructors for Loading Resources

The following classes gained constructors that allow for loading/opening resources in a single expression. Upon failure they throw an `sf::Exception`.

- `sf::InputSoundFile`
- `sf::OutputSoundFile`
- `sf::Music`
- `sf::SoundBuffer`
- `sf::Font`
- `sf::Image`
- `sf::RenderTexture`
- `sf::Shader`
- `sf::Texture`
- `sf::FileInputStream`
- `sf::Cursor`

SFML 3 still supports the SFML 2 style of error handling in addition to these new constructors.

v2:

```
sf::SoundBuffer soundBuffer;
if (!soundBuffer.loadFromFile("sound.wav"))
{
    // Handle error
}
```

v3:

```
sf::SoundBuffer soundBuffer;
if (!soundBuffer.loadFromFile("sound.wav"))
{
    // Handle error
}

// OR

const sf::SoundBuffer soundBuffer("sound.wav");
```

`sf::Vector2<T>` and `sf::Vector3<T>` Utility Functions

`sf::Vector2<T>` and `sf::Vector3<T>` gained a number of new functions for performing common mathematic operations on vectors.

`sf::Vector2<T>` Function	Description
`Vector2(T, sf::Angle)`	Construct from polar coordinates
`length()`	Get length
`lengthSquared()`	Get length squared
`normalized()`	Get vector normalized to unit circle
`angleTo(sf::Vector2)`	Get angle to another vector
`angle()`	Get angle from X axis
`rotatedBy(sf::Angle)`	Get vector rotated by a given angle
`projectedOnto(sf::Vector2)`	Get vector projected onto another vector
`perpendicular()`	Get perpendicular vector
`dot(sf::Vector2)`	Get dot product
`cross(sf::Vector2)`	Get Z component of cross product

`sf::Vector2<T>` Function	Description
`componentWiseMul(sf::Vector2)`	Get component-wise multiple
`componentWiseDiv(sf::Vector2)`	Get component-wise divisor

`sf::Vector3<T>` Function	Description
`length()`	Get length
`lengthSquared()`	Get length squared
`normalized()`	Get vector normalized to unit circle
`dot(sf::Vector3)`	Get dot product
`cross(sf::Vector3)`	Get cross product
`componentWiseMul(sf::Vector3)`	Get component-wise multiple
`componentWiseDiv(sf::Vector3)`	Get component-wise divisor

Threading Primitives

`sf::Lock`, `sf::Mutex`, `sf::Thread`, `sf::ThreadLocal`, and `sf::ThreadLocalPtr` were removed and replaced with their equivalents from the standard library. The standard library provides multiple options for threads, locks, and mutexes among other threading primitives.

v2	v3
`sf::Lock`	`std::lock_guard` or `std::unique_lock`
`sf::Mutex`	`std::mutex` or `std::recursive_mutex`
`sf::Thread`	`std::thread` or `std::jthread` (requires C++20)

v2	v3
`sf::ThreadLocal`	`thread_local`
`sf::ThreadLocalPtr`	`thread_local`

Sound Samples and Channel Map

SFML 3 introduces the concept of a Channel Map which defines the mapping of the position in sample frame to the sound channel. For example, if you have a sound frame with six different samples for a 5.1 sound system, the Channel Map defines how each of those samples map to which speaker channel.

SFML 2 always assumed the order as specified by OpenAL.

```
auto samples = std::vector<std::int16_t>();
// ...

auto channelMap = std::vector<sf::SoundChannel>{
    sf::SoundChannel::FrontLeft,
    sf::SoundChannel::FrontCenter,
    sf::SoundChannel::FrontRight,
    sf::SoundChannel::BackRight,
    sf::SoundChannel::BackLeft,
    sf::SoundChannel::LowFrequencyEffects
};
auto soundBuffer = sf::SoundBuffer(samples.data(), samples.size(),
channelMap.size(), 44100, channelMap);
auto sound = sf::Sound(soundBuffer);
```

This a breaking change for the following APIs:

- `bool sf::SoundBuffer::loadFromSamples(...)`
- `bool sf::SoundBuffer::update(...)`
- `void sf::SoundStream::initialize(...)`
- `bool sf::OutputSoundFile::openFromFile(...)`
- `bool sf::SoundFileWriter::open(...)`

Deprecated APIs

SFML 3 removes all of the deprecated APIs in SFML 2.

Deprecated API	Replacement
`sf::Event::MouseWheelEvent`	`sf::Event::MouseWheelScrolled`
`sf::RenderWindow::capture`	See 1
`sf::RenderTexture::create`	`sf::RenderTexture::resize`
`sf::Shader::setParameter`	`sf::Shader::setUniform`
`sf::Text::setColor`	`sf::Text::setFillColor`
`sf::Text::getColor`	`sf::Text::getFillColor`
`sf::PrimitiveType::LinesStrip`	`sf::PrimitiveType::LineStrip`
`sf::PrimitiveType::TrianglesStrip`	`sf::PrimitiveType::TriangleStrip`
`sf::PrimitiveType::TrianglesFan`	`sf::PrimitiveType::TriangleFan`
`sf::PrimitiveType::Quads`	See 2
`sf::Keyboard::BackSlash`	`sf::Keyboard::Key::Backslash`
`sf::Keyboard::BackSpace`	`sf::Keyboard::Key::Backspace`
`sf::Keyboard::Dash`	`sf::Keyboard::Key::Dash`
`sf::Keyboard::Quote`	`sf::Keyboard::Key::Hyphen`
`sf::Keyboard::Return`	`sf::Keyboard::Key::Enter`
`sf::Keyboard::SemiColon`	`sf::Keyboard::Key::Semicolon`
`sf::Keyboard::Tilde`	`sf::Keyboard::Key::Grave`

1. `sf::RenderWindow::capture` can be recreated by using an `sf::Texture` and its `sf::Texture::update(const Window&)` function to copy its contents into an `sf::Image` instead.
2. `sf::PrimitiveType::Quads` can be replaced by another primitive type. This is not a drop-in replacement but rather will require refactoring your code to work with a new geometry. One viable option is to use `sf::PrimitiveType::Triangles` where two adjacent triangles join to form what was previously one quad.

Anti-Aliasing Renamed

SFML 3 capitalizes the `A` of `aliasing` for all the APIs.

- `sf::RenderTexture::getMaximumAntialiasingLevel()` becomes `sf::RenderTexture::getMaximumAntiAliasingLevel()`
- `sf::ContextSettings::antialiasingLevel` becomes `sf::ContextSettings::antiAliasingLevel`

CoordinateType for RenderStates

The enum `sf::CoordinateType` was moved from the `sf::Texture` into its own dedicated enum class.

The `sf::RenderStates` class got a new member of type `sf::CoordinateType` to control how the texture coordinates will be interpreted. By default SFML uses `sf::CoordinateType::Pixels`, while `sf::CoordinateType::Normalized` is the default for OpenGL. Using `sf::CoordinateType::Normalized` with `sf::RenderStates` allows for using normalized textures with `sf::VertexArray` and `sf::VertexBuffer`.

The constructor for `sf::RenderStates` has changed.

v2:

```
auto renderStates = sf::RenderStates(sf::BlendAlpha,
                                     transform,
                                     texture,
                                     nullptr);
```

v3:

```
auto renderStates = sf::RenderStates(sf::BlendMode::BlendAlpha,
                                     transform,
                                     sf::CoordinateTye::Pixels,
                                     texture,
                                     nullptr);
```

Other Minor Changes

SFML 3 includes various smaller changes that ought to be mentioned.

- Changed the parameter order of the `sf::Text` constructor, so that the provided font is always the first parameter
- Reverted to default value of CMake's `BUILD_SHARED_LIBS` which means SFML now builds static libraries by default
- Changed `sf::String` interface to use `std::u16string` and `std::u32string`
- Removed `sf::ContextSettings` constructor in favor of aggregate initialization
- Removed `sf::View::reset` in favor of assigning from a new `sf::View` object
- Removed `sf::Vertex` constructors in favor of aggregate initialization
- Renamed `sf::Mouse::Button::XButton1` and `sf::Mouse::Button::XButton2` enumerators to `sf::Mouse::Button::Extra1` and `sf::Mouse::Button::Extra2`
- Removed NonCopyable.hpp header in favor of using built-in language features for disabling copy operators
- Converted the following classes to namespaces: `sf::Clipboard`, `sf::Keyboard`, `sf::Joystick`, `sf::Listener`, `sf::Mouse`, `sf::Sensor`, `sf::Touch`, `sf::Vulkan`
- Removed `sf::SoundStream::setProcessingInterval` as miniaudio matches the internal processing rate to the underlying backend