Events explained

Introduction

This tutorial is a detailed list of window events. It describes them, and shows how to (and how not to) use them.

The sf::Event type

SFML uses a type-safe API for events. There are two main ways to use this API. For a comprehensive example of both approaches, check out the EventHandling example program.

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

The first option is based around `sf::Event`'s member functions `getIf<T>` and `is<T>`. The template argument `T` must be an event subtype such as `sf::Event::Resized` or `sf::Event::MouseMoved`. `getIf<T>` returns a pointer to the specific event subtype if the template argument matches the active event subtype. Otherwise it returns `nullptr`.

`is<T>` returns `true` if the template argument matches the active event subtype. Otherwise it returns `false`. This is especially useful for subtypes like `sf::Event::Closed` which does not contain any data.

Note how the API for getting events has changed slightly from SFML 2. `sf::WindowBase::pollEvent` and `sf::WindowBase::waitEvent` returns a `std::optional<sf::Event>`. These two functions *might* return an event but they might not.

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();
        }
    }
    // Rest of the main loop
}
```

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 shorter expression.

`sf::WindowBase::handleEvents`

The second option for processing events is via the `sf::WindowBase::handleEvents` function. This functions performs event visitation. What this means is that you can provide lambdas or other callables 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. Depending on the current active event subtype, the corresponding callback is called.

Note

You do not have to provide callbacks for all possible event subtypes.

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);

    // Rest of the main loop
}
```

The Closed event

The `sf::Event::Closed` event is triggered when the user wants to close the window through any of the possible methods the window manager provides ("close" button, keyboard shortcut, etc.). This event only represents a close request, meaning the window is not yet closed when the event is received.

Typically, one will just call `window.close()` in reaction to this event to actually close the window. However, you may also want to do something else first, like saving the current application state or asking the user what to do. If you don't do anything, the window remains open.

There's no event data associated with this event.

```
if (event->is<sf::Event::Closed>())
    window.close();
```

The Resized event

The `sf::Event::Resized` event is triggered when the window is resized, either through user action or programmatically by calling `window.setSize`. You can use this event to adjust the rendering settings: the viewport if you use OpenGL directly, or the current view if you use sfml-graphics.

The data associated with this event contains the new size of the window.

```
if (const auto* resized = event->getIf<sf::Event::Resized>())
{
    std::cout << "new width: " << resized->size.x << std::endl;
    std::cout << "new height: " << resized->size.y << std::endl;
}
```

The FocusLost and FocusGained events

The `sf::Event::FocusLost` and `sf::Event::FocusGained` events are triggered when the window loses/gains focus, which happens when the user switches the currently active window. When the window is out of focus, it doesn't receive keyboard events. This event can be used e.g. if you want to pause your game when the window is inactive.

There's no event data associated with this event.

```
if (event->is<sf::Event::FocusLost>())
    myGame.pause();

if (event->is<sf::Event::FocusGained>())
    myGame.resume();
```

The TextEntered event

The `sf::Event::TextEntered` event is triggered when a character is typed. This must not be confused with the `KeyPressed` event. `TextEntered` interprets the user input and produces the appropriate printable character. For example, pressing '^' then 'e' on a

French keyboard will produce two `KeyPressed` events, but a single `TextEntered` event containing the 'ê' character. It works with all the input methods provided by the operating system, even the most specific or complex ones.

This event is typically used to catch user input in a text field.

The data associated with this event contains the Unicode value of the entered character. You can either put this data directly in a `sf::String`, or cast it to a `char` after making sure that it is in the ASCII range (0 - 127).

```
if (const auto* textEntered = event->getIf<sf::Event::TextEntered>())
{
    if (textEntered->unicode < 128)
        std::cout << "ASCII character typed: " << static_cast<char>
(textEntered->unicode) << std::endl;
}
```

Note that since they are part of the Unicode standard, some non-printable characters such as *backspace* are generated by this event. In most cases you'll need to filter them out.

⚠ **Warning**

Many programmers use the `KeyPressed` event to get user input and start to implement crazy algorithms that try to interpret all the possible key combinations to produce correct characters. Don't do that!

The KeyPressed and KeyReleased events

The `sf::Event::KeyPressed` and `sf::Event::KeyReleased` events are triggered when a keyboard key is pressed/released.

If a key is held, multiple `KeyPressed` events will be generated, at the default operating system delay (i.e. the same delay that applies when you hold a letter in a text editor). To disable repeated `KeyPressed` events, you can call `window.setKeyRepeatEnabled(false)`. On the flip side, it is obvious that `KeyReleased` events can never be repeated.

This event is the one to use if you want to trigger an action exactly once when a key is pressed or released, like making a character jump with space, or exiting something with escape.

Sometimes, people try to react to `KeyPressed` events directly to implement smooth movement. Doing so will *not* produce the expected effect, because when you hold a key you only get a few events (remember, the repeat delay). To achieve smooth movement with events, you must use a boolean that you set on `KeyPressed` and clear on `KeyReleased`; you can then move (independently of events) as long as the boolean is set.

The other (easier) solution to produce smooth movement is to use real-time keyboard input with `sf::Keyboard` (see the dedicated tutorial).

The data associated with these events contains the scancode and key code of the pressed/released key, as well as the current state of the modifier keys (alt, control, shift, system).

Scancodes are unique values for each physical key on a keyboard, regardless of the language or layout, whereas key codes represent keys based on the user's chosen layout. For instance, the Z key is in the bottom row to the left of the X key on a US layout. Referring to the scancode for Z would identify this physical key location on any keyboard. However, on a German layout, the same physical key is labeled Y. Thus, using the key code for Y can lead to different physical key locations, depending on the chosen layout.

It is strongly recommended to use scancodes over key codes if the physical location of the keys is more important than the key values that depend on the keyboard layout, such as using WASD keys for movement.

```
if (const auto* keyPressed = event->getIf<sf::Event::KeyPressed>())
{
    if (keyPressed->scancode == sf::Keyboard::Scan::Escape)
    {
        std::cout << "the escape key was pressed" << std::endl;
        std::cout << "scancode: " << static_cast<int>(keyPressed->scancode)
<< std::endl;
        std::cout << "code: " << static_cast<int>(keyPressed->code) <<
std::endl;
        std::cout << "control: " << keyPressed->control << std::endl;
        std::cout << "alt: " << keyPressed->alt << std::endl;
        std::cout << "shift: " << keyPressed->shift << std::endl;
        std::cout << "system: " << keyPressed->system << std::endl;
        std::cout << "description: " <<
sf::Keyboard::getDescription(keyPressed->scancode).toAnsiString() <<
std::endl;
        std::cout << "localize: " << static_cast<int>
(sf::Keyboard::localize(keyPressed->scancode)) << std::endl;
        std::cout << "delocalize: " << static_cast<int>
(sf::Keyboard::delocalize(keyPressed->code)) << std::endl;
    }
}
```

Note

Note that some keys have a special meaning for the operating system, and will lead to unexpected behavior. An example is the F10 key on Windows, which "steals" the focus, or the F12 key which starts the debugger when using Visual Studio.

The MouseWheelScrolled event

The `sf::Event::MouseWheelScrolled` event is triggered when a mouse wheel moves up or down, but also laterally if the mouse supports it.

The data associated with this event contains the number of ticks the wheel has moved, what the orientation of the wheel is and the current position of the mouse cursor.

```
if (const auto* mouseWheelScrolled = event-
>getIf<sf::Event::MouseWheelScrolled>())
{
    switch (mouseWheelScrolled->wheel)
    {
        case sf::Mouse::Wheel::Vertical:
            std::cout << "wheel type: vertical" << std::endl;
            break;
        case sf::Mouse::Wheel::Horizontal:
            std::cout << "wheel type: horizontal" << std::endl;
            break;
    }
    std::cout << "wheel movement: " << mouseWheelScrolled->delta <<
std::endl;
    std::cout << "mouse x: " << mouseWheelScrolled->position.x << std::endl;
    std::cout << "mouse y: " << mouseWheelScrolled->position.y << std::endl;
}
```

The MouseButtonPressed and MouseButtonReleased events

The `sf::Event::MouseButtonPressed` and `sf::Event::MouseButtonReleased` events are triggered when a mouse button is pressed/released. SFML supports 5 mouse buttons: left, right, middle (wheel), extra #1 and extra #2 (side buttons).

The data associated with these events contains the code of the pressed/released button, as well as the current position of the mouse cursor.

```
if (const auto* mouseButtonPressed = event-
>getIf<sf::Event::MouseButtonPressed>())
{
    if (mouseButtonPressed->button == sf::Mouse::Button::Right)
    {
        std::cout << "the right button was pressed" << std::endl;
        std::cout << "mouse x: " << mouseButtonPressed->position.x <<
std::endl;
        std::cout << "mouse y: " << mouseButtonPressed->position.y <<
std::endl;
    }
}
```

The MouseMoved event

The `sf::Event::MouseMoved` event is triggered when the mouse moves within the window.

Note

This event is triggered even if the window isn't focused but only when the mouse moves within the inner area of the window, not when it moves over the title bar or window borders.

The data associated with this event contains the current position of the mouse cursor relative to the window.

```
if (const auto* mouseMoved = event->getIf<sf::Event::MouseMoved>())
{
    std::cout << "new mouse x: " << mouseMoved->position.x << std::endl;
    std::cout << "new mouse y: " << mouseMoved->position.y << std::endl;
}
```

The MouseMovedRaw event

The `sf::Event::MouseMovedRaw` event is triggered when the mouse moved within the window even if the mouse is moved a distance too small to perceive.

While the `sf::Event::MouseMoved` position value is dependent on the screen resolution, the raw data of this event is not. If the physical mouse is moved too little to cause the cursor on the screen to move at least a single pixel, no `sf::Event::MouseMoved` event will be generated. In contrast, any movement information generated by the mouse independent of its sensor resolution will always generate an `sf::Event::MouseMovedRaw` event.

The data associated with this event contains the change in position of the mouse cursor relative to the window.

```
if (const auto* mouseMovedRaw = event->getIf<sf::Event::MouseMovedRaw>())
{
    std::cout << "new mouse x: " << mouseMoved->delta.x << std::endl;
    std::cout << "new mouse y: " << mouseMoved->delta.y << std::endl;
}
```

Note

Currently, raw mouse input events will only be generated on Windows and Linux.

The MouseEntered and MouseLeft event

The `sf::Event::MouseEntered` and `sf::Event::MouseLeft` events are triggered when the mouse cursor enters/leaves the window. There's no event data associated with these events.

```
if (event->is<sf::Event::MouseEntered>())
    std::cout << "the mouse cursor has entered the window" << std::endl;

if (event->is<sf::Event::MouseLeft>())
    std::cout << "the mouse cursor has left the window" << std::endl;
```

The JoystickButtonPressed and JoystickButtonReleased events

The `sf::Event::JoystickButtonPressed` and `sf::Event::JoystickButtonReleased` events are triggered when a joystick button is pressed/released.

SFML supports up to 8 joysticks and 32 buttons.

The data associated with these events contains the identifier of the joystick and the index of the pressed/released button.

```
if (const auto* joystickButtonPressed = event-
>getIf<sf::Event::JoystickButtonPressed>())
{
    std::cout << "joystick button pressed!" << std::endl;
    std::cout << "joystick id: " << joystickButtonPressed->joystickId <<
std::endl;
    std::cout << "button: " << joystickButtonPressed->button << std::endl;
}
```

The JoystickMoved event

The `sf::Event::JoystickMoved` event is triggered when a joystick axis moves. Joystick axes are typically very sensitive. That's why SFML uses a detection threshold to avoid spamming your event loop with tons of `JoystickMoved` events. This threshold can be changed with the `sf::Window::setJoystickThreshold` function, in case you want to receive more or less joystick move events.

SFML supports 8 joystick axes: X, Y, Z, R, U, V, POV X and POV Y. How they map to your joystick depends on its driver.

The data associated with this event contains the identifier of the joystick, the name of the axis, and its current position (in the range [-100, 100]).

```
if (const auto* joystickMoved = event->getIf<sf::Event::JoystickMoved>())
{
    if (joystickMoved->axis == sf::Joystick::Axis::X)
    {
        std::cout << "X axis moved!" << std::endl;
        std::cout << "joystick id: " << joystickMoved->joystickId <<
std::endl;
        std::cout << "new position: " << joystickMoved->position <<
std::endl;
    }
}
```

The JoystickConnected and JoystickDisconnected events

The `sf::Event::JoystickConnected` and `sf::Event::JoystickDisconnected` events are triggered when a joystick is connected/disconnected.

The data associated with this event contains the identifier of the connected/disconnected joystick.

```
if (const auto* joystickConnected = event-
>getIf<sf::Event::JoystickConnected>())
    std::cout << "joystick connected: " << joystickConnected->joystickId <<
std::endl;

if (const auto* joystickDisconnected = event-
>getIf<sf::Event::JoystickDisconnected>())
    std::cout << "joystick disconnected: " << joystickDisconnected-
>joystickId << std::endl;
```