SFML

Modules

SFML consists of 5 modules:

- system
- window
- graphics
- audio
- network

Each module has its own purpose, but some may be dependent on oneanother.
The modules are reccognizable with their `sf::` namespace.
For CPSE2, system, window, and graphics are applicable.

System

Time

Handling time in SFML is done through the `sf::Time` class.
With the sf::Time class you can also perform arithmetic on different values respective to their scale.
So for example:

```
sf::Time time = sf::seconds(5);
sf::Time millis = sf::milliseconds(500);
sf::Time total = time + millis; // 5500 milliseconds

sf::Int64 total_int = total.asMilliseconds(); // convert to integer value for cout
float total_float = total.asSeconds();
std::cout << total_int << std::endl; // outputs : 5500
std::cout << total_float << std::endl; // output : 5.5
```

Creating

Time can be constructed from many different units with their corresponding `sf::<unit_here>`
For example:

```
sf::Time t1 = sf::milliseconds(50); // 50 milliseconds
sf::Time t2 = sf::seconds(0.5f); // 500 milliseconds
```

Conversion between unit type and sf::Time class

```
sf::Time time = sf::seconds(5);

sf::Int64 sec = time.asSeconds(); // int = 5
sf::Int54 msec = time.asMilliseconds(); // int = 5000
```

Measuring

Elapsed time can be measured with the `sf::Clock` class.
The class has two functions : `getElapsedTime()` and `restart()`. The clock starts at construction.

```
sf::Clock clock; // start clock

sf::Time elapsed = clock.getElapsedTime(); // get elapsed time
std::cout << elapsed.asSeconds() << std::endl;
clock.restart(); // restart clock
elapsed = clock.getElapsedTime(); // get new elapsed time since restart
```

Data-/Filestreams

FOR LATER

Window

The window class is the container for everything to be drawn in.

Creating

Creating a window can be done like follows:

```
// construction and initialization
sf::Window window(sf::VideoMode(x_res, y_res), "Window Title", Style_Parameters);

// separate construction and initialization
sf::Window window;
window.create(sf::VideoMode(x_res, yres), "Window Title", Style_Parameters);
```

`x_res, y_res` should be the amount of pixels horizontal, and vertical respectively.
`"Window Title"` can be any string you want, it'll show in the titlebar above your window
Style paramters dictate a few features your window has, they need to be bitwise-or'd together like this : `sf::Style::Close | sf::Style::Titlebar`
Valid styles are:

Style	Description
`sf::Style::None`	No decoration at all (useful for splash screens, for example); this style cannot be combined with others
`sf::Style::Titlebar`	The window has a titlebar
`sf::Style::Resize`	The window can be resized and has a maximize button
`sf::Style::Close`	The window has a close button
`sf::Style::Fullscreen`	The window is shown in fullscreen mode; this style cannot be combined with others, and requires a valid video mode
`sf::Style::Default`	The default style, which is a shortcut for Titlebar | Resize | Close

tidbits

You can set a max framerate and enable vSync in SFML windows

```
window.setFrameLimit(60);
window.setVerticalSyncEnables(true);
```

Events

To actually keep the window open you'll have to use events and a while loop.
An event is something that happens to the window, be it internally or externally.
Events can be handled in a while loop that polls an event queue with `pollEvent()`
For a list of possible events see the enum under Public Types at SFML Event Documentation

Closed event

The closed event is generated when the window is closed through the X-button provided by the OS, or any other means really.
Pressing the close button does not acutally close the window it just sends an event, this even can then be handled.

```
while(window.isOpen()){ // Main Game Loop
        sf::Event event; // construct empty event
        while( window.pollEvent(event)){ //  read incoming event from event queue
                if(event.type == sf::Event::Closed){ // EventType Closed
                        // do whatever store the state of things etc.
                        window.close(); // close the window
                }
        }
}
```

KeyPressed and KeyReleased event

A KeyPressed event is generated when any button on your keyboard is pressed.
If a key is held down, a KeyPressed event is generated every Operating System keyboard

delay, not neces
While you can react to this event and conditionally see what key was pressed, it's recommended that you directly use the `Keyboard::isKeyPressed()` (see User Input) method for smooth user input.
However if you do want to use the KeyPressed event:

```
while(window.isOpen()){ // Main Game Loop
        sf::Event event; // construct empty event
        while( window.pollEvent(event)){ //  read incoming event from event queue
                if(event.type == sf::Event::KeyPressed){
                        if(event.key.scancode == sf::Keyboard::Scan::A){
                                std::cout << "Key 'A' Pressed" << std::endl;
                        }
                }
        }
}
```

TextEntered event

The textentered event interprets keyboard presses into their corresponding ASCII characters. This means with certain locale settings pressing `"` and then `e` will produce two KeyPressed events, but only one `ë` TextEntered event.

```
while(window.isOpen()){ // Main Game Loop
        sf::Event event; // construct empty event
        while( window.pollEvent(event)){ //  read incoming event from event queue
                if(event.type == sf::Event::TextEntered){
                        if(event.text.unicode < 128){ // check if unicode value is
less than 128, which is ascii value range
                                std::cout << "ASCII character typed : " <<
static_cast<char>(event.text.unicode) << std::endl;
                        }
                }
        }
}
```

MouseButtonPressed, MouseWheelScrolled, MouseMoved

MouseButtonPressed and MouseWheelScrolled events also include the mouse position in its `x` and `y` variables.
The MouseMoved event of course also includes the mouse position.

```
while(window.pollEvent(event)){
        if(event.type == ...){
                if(event.mouseButton.button == sf::Mouse::Left){
                        ...
                        std::cout << "Mouse Left click at: ( " <<
```

```
event.mouseButton.x << ", " << event.mouseButton.y << " )\n";
                } else if (event.mouseButton.button == sf::Mouse::Right){
                        ...}
        } else if (event.type == sf::Event::MouseWheelScrolled){
                std::cout << "Wheel movement in ticks: " <<
event.mouseWheelScroll.x << std::endl;
        }
        }
}
```

User Input

Directly reading from keyboard and mouse is usually the better option between that and using events to handle real-time inputs.

Keyboard

You can choose between using keycodes, which are associated with the key legend according to the current keyboard layout, or using scancodes, which are associated with the physical location on the keyboard, which will always be the same regardless of keyboard layout
Key codes are defined in the `sf::Keyboard::Key` enum.
Scancodes are defined in the `sf::Keyboard::Scancode` enum.

```
if(sf::Keyboard::isKeyPressed(sf::Keyboard::A)){
        std::cout << "Key A pressed";
}
if(sf::Keyboard::isKeyPressed(sf::Keyboard::Scan::A)){
        std::cout << "Scan A pressed";
}
```

Mouse

FOR LATER

Graphics

Drawing 2D

For drawing object from the graphics module in a window, you must use the `sf::RenderWindow` class.
It's derived from window and uses the same functions, but adds functionality for drawing 2D objects.
Instead of creating a default window you should create a RenderWindow.
In the game loop you should clear, then draw, then display everything.

Don't worry about performance with this, you can draw thousands or even millions of objects efficiently, because that's what computers nowadays are made for.

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

int main()
{
    // create the window
    sf::RenderWindow window(sf::VideoMode(800, 600), "My window");

    // run the program as long as the window is open
    while (window.isOpen())
    {
        // check all the window's events that were triggered since the last
iteration of the loop
        sf::Event event;
        while (window.pollEvent(event))
        {
            // "close requested" event: we close the window
            if (event.type == sf::Event::Closed)
                window.close();
        }

        // clear the window with black color
        window.clear(sf::Color::Black);

        // draw everything here...
        // window.draw(...);

        // end the current frame
        window.display();
    }

    return 0;
}
```

There are 4 types of entities that can be drawn, 3 of those are ready to use: shapes, sprites, and text
There are also SFML primitives, which can be used like so:

```
window.draw(vertices, vertexCount, PrimitiveType);
```

Shapes

Shapes are the primitive things you can draw:
circles, rectangles, squares, polygons, lines, and convex shapes; defined by coordinate points
Every shape has 3 properties: color, outline, and texture.

```
sf::CircleShape shape(radius);

shape.setFillColor(sf::Color(r, g, b));

shape.setOutLineThickness(thickness); // set a negative thickness to grow the
outline inwards
shape.setOutLineColor(sf::Color(r, g, b));

shape.setTexture(&sf::Texture); // create texture
shape.setTextureRect(sf::IntRect()); // set texture coordinates
```

Rectangles

```
// One parameter: size
sf::RectangleShape rectangle(sf::Vector2f(100.f, 50.f)); // 100 pixels wide, 50
pixels high
rectangle.setSize(sf::Vector2f(200.f, 200.f)) // resize rectangle
```

Circles

```
// two parameters: radius, number of sides
sf::CircleShape circle(200.f, 32); // 32 sides, means smooth-ish circle

circle.setRadius(40.f); // change radius
circle.setPointCount(8); // change number of sides / 'resolution'
```

Equal Polygons (triangle, octagon etc.)

There is no dedicated class for equal polygons, but you can use a circle shape with a certain amount of sides to create a polygon.

```
// define a triangle
sf::CircleShape triangle(80.f, 3);

// define a square
sf::CircleShape square(80.f, 4);

// define an octagon
sf::CircleShape octagon(80.f, 8);
```

Convex Shapes (polygons)

FOR LATER

Transforming Entities

Transforming entities consists of 4 parts:

- Origin : The point in the window every transform is based off
- Position : Where on the screen the entity is (relative to its origin)
- Rotation : In what rotation the entity is in degrees (relative to horizontal, around its origin)
- Scale : How big the entity is scaled (relative to its origin, DOESN'T MODIFY THINGS LIKE RADIUS OR SIZE)

Sprites & Textures

FOR LATER

Text & Fonts

FOR LATER