

Game Engine Development II

Week2

Hooman Salamat

Game Loop Programming

Objectives

- Explore an example game loop
- Review and apply the concepts of frames, frame rates and fixed time steps
- Create and display sprites to the game window

How to set an Icon for the window

- `sf::Image mlcon;`
- `mlcon.loadFromFile("Media/Textures/icon.png");`
- `mWindow.setIcon(mlcon.getSize().x, mlcon.getSize().y, mlcon.getPixelsPtr());`
- `sf::Image` also provides functions to load, read, write pixels
- `mlcon.create(20, 20, sf::Color::Yellow);`
- `sf::Color color = mlcon.getPixel(0, 0);`
- `color.a = 0;` //make the top-left pixel transparent
- `color.r = 0;` //set the r = 0 (rgb) from the color
- `mlcon.setPixel(0, 0, color);`

Font and Text

- Create a graphical text to display

```
sf::Font mFont;  
sf::Text mText;  
if (!mFont.loadFromFile("Media/Sansation.ttf"))  
return;
```

```
mText.setString("Hello SFML");  
mText.setFont(mFont);  
mText.setPosition(5.f, 5.f);  
mText.setCharacterSize(50);  
mText.setFillColor(sf::Color::Black);
```

Play the Music

- Streamed music played from an audio file
- Musics are sounds that are streamed rather than completely loaded in memory
- A music is played in its own thread in order not block the rest of the program
- Supported audio formats: ogg, wav, flac, aiff, au, raw, paf, svx, nist, voc, ircam, w64, mat4, mat5, pvf, htk, sds, avr, sd2, caf, wve, mpc2k, rf64

Music Parameters

- `#include <SFML/Audio.hpp>`
- `sf::Music mMusic;`
- `mMusic.openFromFile("Media/Textures/nice_music.ogg");`
- `//change some parameters`
- `mMusic.setPosition(0, 1, 10); //change its 3D position`
- `mMusic.setPitch(2); //increase the pitch`
- `mMusic.setVolume(50); //reduce the volume`
- `mMusic.setLoop(true); //make it loop`
- `mMusic.setAttenuation(100);`
- `mMusic.play();`

Game Loop

- The run() function you saw in the example from last week, and below, is known as the main loop or game loop

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


Game Loop (cont'd.)

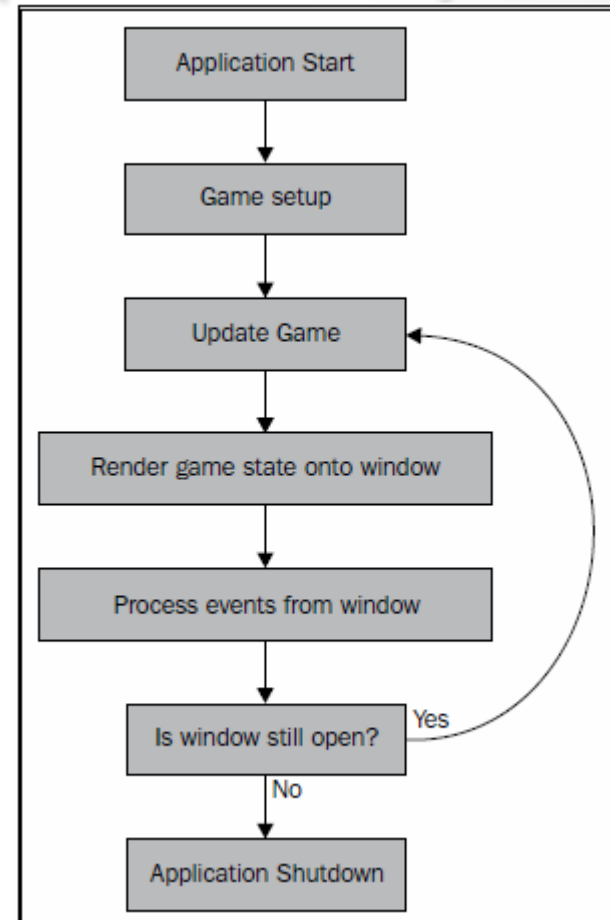
- It processes all the components in the game and continues to do so until the application is terminated
- The processing of events, updating of all game assets and then rendering them to the destination output is a standard loop for games

Game Loop (cont'd.)

- It processes all the components in the game and continues to do so until the application is terminated
- The processing of events, updating of all game assets and then rendering them to the destination output is a standard loop for games
- You've heard the term **frame** or **tick** before, and that is what we call an iteration of the loop

Game Loop (cont'd.)

- The flowchart to the right illustrates the logic and different processes of the game, including the main loop



Events

- Events can be user-generated such as mouse clicks or movement or keyboard presses
- They can also be generated by the assets in the game, such as when an enemy spots the player
- We don't have any events in our example yet, so let's create some!
- How 'bout moving that circle with the keyboard?

Events (cont'd.)

- For events in SFML, we use the `sf::Event` object
- We're going to use two events for this example:

`sf::Event::KeyPressed` and `sf::Event::KeyReleased`

processEvents()

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

handlePlayerInput()

```
void Game::handlePlayerInput(sf::Keyboard::Key key, bool isPressed)
{
    if (key == sf::Keyboard::W)
        mIsMovingUp = isPressed;
    else if (key == sf::Keyboard::S)
        mIsMovingDown = isPressed;
    else if (key == sf::Keyboard::A)
        mIsMovingLeft = isPressed;
    else if (key == sf::Keyboard::D)
        mIsMovingRight = isPressed;
}
```

New update()

```
void Game::update()
{
    sf::Vector2f movement(0.f, 0.f);
    if (mIsMovingUp)
        movement.y -= 1.f;
    if (mIsMovingDown)
        movement.y += 1.f;
    if (mIsMovingLeft)
        movement.x -= 1.f;
    if (mIsMovingRight)
        movement.x += 1.f;

    mPlayer.move(movement);
}
```


How about mouse?

```
if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {  
    sf::Vector2i mousePosition =  
    sf::Mouse::getPosition(mWindow);  
    mPlayer.setPosition((float)mousePosition.x,  
        (float)mousePosition.y);  
}
```

Vector Object

- SFML's Vector object is instantiated as:

```
sf::Vector2<float>
```

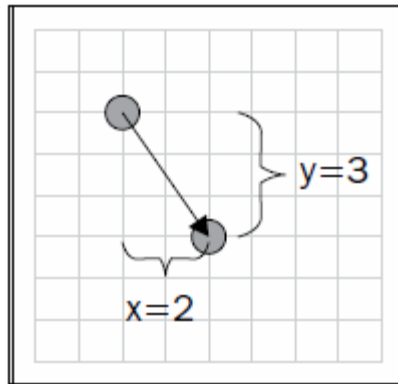
- We use the typedef for variable declarations, which is as follows:

```
sf::Vector2f myVector(0.f, 0.f);
```

- As you would expect, there are many common operations in the Vector class that we can access through a Vector object

Vector Object (cont'd.)

- In games, vectors can represent coordinates or a direction to move
- The diagram below represents a vector(2,3) and it could be a translation of 2 units to the right and 3 down:



Frame-Independence

- You might remember from Unity that we were able to move an object a certain number of units per second
 - We multiplied the speed by `Time.deltaTime`
- We can do this in SFML too, so that the player's movement isn't dependent on the framerate – or number of times the update runs per second

Frame-Independence (cont'd.)

- Well, we're still relying on the framerate, but the movement is spread out evenly over the frames
- So let's have a look at our new update function and see what's new...

New update()

```
void Game::update(sf::Time deltaTime)
{
    sf::Vector2f movement(0.f, 0.f);
    if (mIsMovingUp)
        movement.y -= PlayerSpeed;
    if (mIsMovingDown)
        movement.y += PlayerSpeed;
    if (mIsMovingLeft)
        movement.x -= PlayerSpeed;
    if (mIsMovingRight)
        movement.x += PlayerSpeed;

    mPlayer.move(movement * deltaTime.asSeconds());
}
```

Measuring Frames

- We can measure the time each frame takes in order to figure out `deltaTime`
- We use the `Sf::Clock` class
- `Sf::Clock` has only two methods: `getElapsedTime()` and `restart()`. Both returns the elapsed time since the clock was started and then it resets the clock to zero. `getElapsedTime()` can be called without calling `restart()`

```
sf::Clock clock;  
sf::Time time = clock.getElapsedTime();  
float seconds = time.asSeconds();  
sf::Int32 milliseconds = time.asMilliseconds();  
sf::Int64 microseconds = time.asMicroseconds();  
time = clock.restart();
```

Measuring Frames (cont'd.)

```
void Game::run()
{
    sf::Clock clock;
    while (mWindow.isOpen())
    {
        sf::Time deltaTime = clock.restart();
        processEvents();
        update(deltaTime);
        render();
    }
}
```


Fixed Time Step

- Time based on a system function such as a while loop will never be constant
 - We saw this way back with HTML5 and its highly-varying frame rate
 - Unity too!
- Fortunately we can create fixed time execution using a counter and a check
 - The while loop is definitely going to execute fast enough to serve as very small time increments added to our counter variable

Fixed Time Step (cont'd.)

```
void Game::run()
{
    sf::Clock clock;
    sf::Time timeSinceLastUpdate = sf::Time::Zero;
    while (mWindow.isOpen())
    {
        processEvents();
        timeSinceLastUpdate += clock.restart();
        while (timeSinceLastUpdate > TimePerFrame)
        {
            timeSinceLastUpdate -= TimePerFrame;
            processEvents();
            update(TimePerFrame);
        }
        render();
    }
}
```

Fixed Time Step (cont'd.)

- If you want to read more on this topic, you can read the article at the following address:
 - <http://gafferongames.com/game-physics/fix-your-timestep>

Displaying Sprites

```
sf::Texture texture;  
if (!texture.loadFromFile("path/to/file.png"))  
{  
    // Handle loading error  
}  
sf::Sprite sprite(texture);  
sprite.setPosition(100.f, 100.f);  
window.clear();  
window.draw(sprite);  
window.display();
```

Rendering

- Rendering is the process of drawing your assets to the screen
- Ideally, we'd only want our assets being drawn if they were updated somehow in the program
 - Would save on performance
- However, *real-time rendering* just draws to the screen as fast as possible

Rendering (cont'd.)

- Have you ever wondered why there is an FPS count in games? Like 30 or 60
- It's because the end user can't see blindingly-fast updates so the frame rate is limited to allow the processor to perform other tasks

Rendering (cont'd.)

- *Double buffering* is a technique of rendering that uses two virtual windows or screens, called buffers
 - Front and back buffers
- The front buffer is what the user sees
- The back buffer is what's going to be drawn next frame – will become the front buffer

Adding the Sprite

```
// Game.hpp
class Game
{
    public:
        Game();
        ...
    private:
        sf::Texture mTexture;
        sf::Sprite mPlayer;
        ...
};
```


Adding the Sprite (cont'd.)

```
// Game.cpp
Game::Game()
: ...
, mTexture()
, mPlayer()
{
    if (!mTexture.loadFromFile("Media/Textures/Eagle.png"))
    {
        // Handle loading error
    }
    mPlayer.setTexture(mTexture);
    mPlayer.setPosition(100.f, 100.f);
}
```