SDL2 Course Day-2

Intro to Graphics

SDL2 basics:

We're using a graphics library called **SDL2** which gives us low-level access to the system's graphics components. It is a **cross platform** software development library (same code runs on any OS). It provides a layer of abstraction for computer hardware interaction. (We don't know how the functions are implemented, we just know what they do and that's all we need to know). Because we're using a library defined by someone else, we need to follow **their rules** for our code to work properly.

To include the library (windows):

```
#include <SDL2/SDL.h>
```

To include the library (mac):

```
#include <SDL.h>
```

the main function should have the following parameters (we're never going to use them, but SDL2 mandates that they need to be there):

```
int main(int argc, char **argv)
```

to initialize the SDL library we need to call the below function:

```
SDL_Init(SDL_INIT_VIDEO);
```

SDL_INIT_VIDEO is used to initialize the video subsystem of the library, we can alternatively initialize the following: (SDL_INIT_EVERYTHING initializes everything but we need only video for now).

```
SDL_INIT_VIDEO
SDL_INIT_AUDIO
SDL_INIT_EVENTS
SDL_INIT_EVERYTHING
SDL_INIT_GAMECONTROLLER
SDL_INIT_HAPTIC
SDL_INIT_JOYSTICK
SDL_INIT_NOPARACHUTE
SDL_INIT_SENSOR
SDL_INIT_TIMER
```

Whatever we create, we need to destroy; At the end of the code we need to call the following function to clear up stuff (Our code will work without it but its good practice to call this):

```
SDL_Quit();
```

Creating a Window:

A window is where you'll see the actual game. Its what the user interacts with and is considered as the GUI (graphical user interface) of the game. It acts as a canvas where all of our graphics are drawn.

To create a window:

```
SDL_Window *Window = SDL_CreateWindow("My window", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 900, 600, 0);
```

The parameter order passed to the function are as follows:
1. Title of the window,
2. Where the Window should be placed on the X axis,
3. Where the Window should be placed on the Y axis,
4. The Width of the Window,
5. The Height of the Window,
6. Optional Flags (For now we put it as 0).

To destroy a window:

```
SDL_DestroyWindow(Window);
```

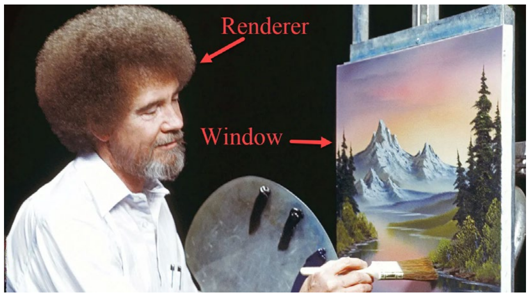
The problem with just creating a window and then destroying it in the next line Is that the window will be visible for a split second before it gets destroyed and the program ends. So, we add a delay:

```
SDL_Delay(4000);
```

The parameter passed to this is the amount of time you want the program to be delayed in **milli-seconds**.

Creating the Renderer:

"Render" is just a fancy word for "Draw". A Renderer is basically an object or a black box which we use to draw something on the window. A very nice way to understand how it works is to imagine the window as a painter's canvas and the renderer as the painter.



To create a renderer:

```
SDL_Renderer *Renderer = SDL_CreateRenderer(Window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
```

The first parameter is the window which you want the renderer to draw to, The 2nd and 3rd parameters are extra flags (just put them, if you're really curious google them).

To destroy the renderer:

```
SDL_DestroyRenderer(Renderer);
```

Basics of Drawing:
The following steps must need to be followed in order: (if they're not, the game wont work as expected, try out doing them in a different order or even not doing them to see how it affects the program).
1.) Clear background.
2.) Draw anything you want.
3.) Tell the renderer to finally draw to screen.

Before calling any renderer function (except the last function which tells the renderer to finally draw to the screen), we need to set the color of the renderer. (If you tell a painter to draw an object, you'll also need to tell him what color to draw the object in first).

```
SDL_SetRenderDrawColor(Renderer, 0, 0, 0, 255);
```

The parameters passed are the Renderer and then the RGBA values of the color.
https://www.nixsensor.com/blog/what-is-rgb-color/
https://www.w3schools.com/css/css3_colors.asp

To activate the A part of the color scheme (the last parameter which decides opacity), we need to add the following line after creating the Renderer:

```
SDL_Renderer *Renderer = SDL_CreateRenderer(Window, -1, SDL
SDL_SetRenderDrawBlendMode(Renderer, SDL_BLENDMODE_BLEND);
```

We wont be using opacity anywhere until wayyyy later so its not necessary right now, but you're free to test it out.

To clear the background:

```
SDL_RenderClear(Renderer);
```

To tell the renderer to finally draw to the screen:

```
SDL_RenderPresent(Renderer);
```

The final graphics of the screen is represented in the form of 2D array (a buffer) which is a system owned buffer. If our renderer wrote into this buffer while the system is reading from this to display, it could cause problems. To solve this problem we have our own copy of the buffer. Calling renderpresent swaps our copy of the buffer with the system's buffer. This is called double buffering, Extra reference:
https://youtu.be/7cRRxlWRl8g
https://youtu.be/f3tO_gyyLmk

```
// clears background with the color Black
SDL_SetRenderDrawColor(Renderer, 0, 0, 0, 255);
SDL_RenderClear(Renderer);

// we draw stuff here

// swaps buffer
SDL_RenderPresent(Renderer);
```

Drawing Stuff on the screen:

The only thing SDL2 allows us to draw on the window is **Rectangles**. Rectangles are our only way to get anything onto the screen and everything you see in the final game is all rectangles. Its rectangles everywhere, there is no other default shape.

Creating Rectangles:

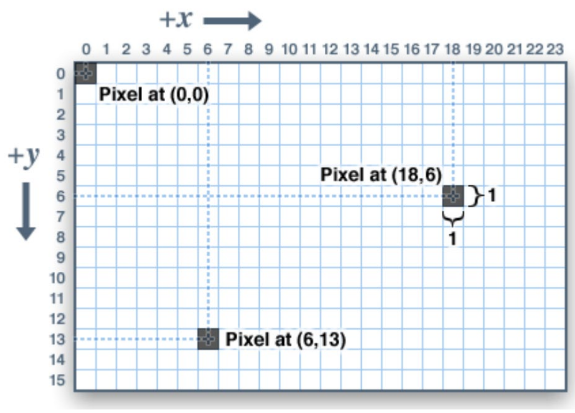
To create a rectangle, we use SDL2's built in structure:

```
SDL_Rect rect1 = {50, 50, 30, 30};
```

SDL_Rect has the following members:

```
rect1.x  // x co-ordinate
rect1.y  // y co-ordinate
rect1.w  // width of rect
rect1.h  // height of rect
```

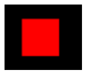
Note that the co-ordinate system in computer graphics is different from what we're used to in math. It starts on the top left rather than bottom left in maths.

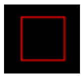


To draw a rectangle:
we first set the color of what we want the color of the rectangle to be:

```
SDL_SetRenderDrawColor(Renderer, 255, 0, 0, 255);
```
(red)

Then we can do either one of the below: (note that we pass the **address** of the rectangle to be drawn)

```
SDL_RenderFillRect(Renderer, &rect1);
```
draws = >  (draws full rectangle)

```
SDL_RenderDrawRect(Renderer, &rect1);
```
draws = >  (draws only outline)

Game Loop:

A normal program follows the following structure:

```
1.) initialize stuff
2.) take input from user
3.) process input / make calculations
4.) output calculations
5.) clear stuff
```

However a game needs to be continuously running, so we put steps 2 – 4 in a loop:

```
1.) initialize stuff

loop over:
    2.) take input from user
    3.) process input / make calculations
    4.) output calculations

5.) clear stuff
```

In our case our output is in the form of graphics (drawing on the screen).

So we put all of our drawing code in a while loop and remove the delay we put earlier:

```
#include <SDL2/SDL.h>

int main(int argc, char **argv) {

    SDL_Init(SDL_INIT_VIDEO);

    SDL_Window *Window = SDL_CreateWindow("My window", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 900, 600, 0);
    SDL_Renderer *Renderer = SDL_CreateRenderer(Window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);

    SDL_Rect rect1 = {50, 50, 30, 30};

    bool quit = false;

    while (!quit) {
        SDL_SetRenderDrawColor(Renderer, 0, 0, 0, 255);
        SDL_RenderClear(Renderer);

        SDL_SetRenderDrawColor(Renderer, 255, 0, 0, 255);
        SDL_RenderFillRect(Renderer, &rect1);

        SDL_RenderPresent(Renderer);
    }

    SDL_DestroyWindow(Window);
    SDL_DestroyRenderer(Renderer);
    SDL_Quit();

    return 0;
}
```

But now our quit variable is not being updated inside our while loop and so we have an infinite loop. Ideally once we press the close button on the window we should be able to exit our loop.

This leads us to events.

Events:
An event can be anything from the user pressing something on his keyboard to moving his mouse or clicking anywhere on the screen.

To store an event we first need to use an event datatype:

```
SDL_Event event;
```

This just creates an event variable, however, if you refer back to the structure of our game, we need to continuously take input from the user in the start of the loop. To do this we have: (Note that we pass in the **address** of the event so that the function can set the value of our variable that we pass in).

```
SDL_PollEvent(&event);
```

Now after this function, our event variable now stores what the user has done in this iteration of the loop. However since SDL_Event is a structure defined by SDL2 we need to know how to access specific events like pressing the close button or a key press:

Close button:

The SDL_Event structure has a member called "type". If the value of this is equal to a constant defined by SDL2 called SDL_QUIT, this indicates that the user has pressed the close button, so we can write the following code:

```
if (event.type == SDL_QUIT) {
    quit = true;
}
```

Keypress:

A key press action in SDL2 is defined by two separate actions,
1.) When you press the key down.
2.) When you release the key after pressing it.

The first is defined by the constant SDL_KEYDOWN and the second by SDL_KEYUP.

To check if a particular key is pressed:

```
if (event.type == SDL_KEYDOWN)
```

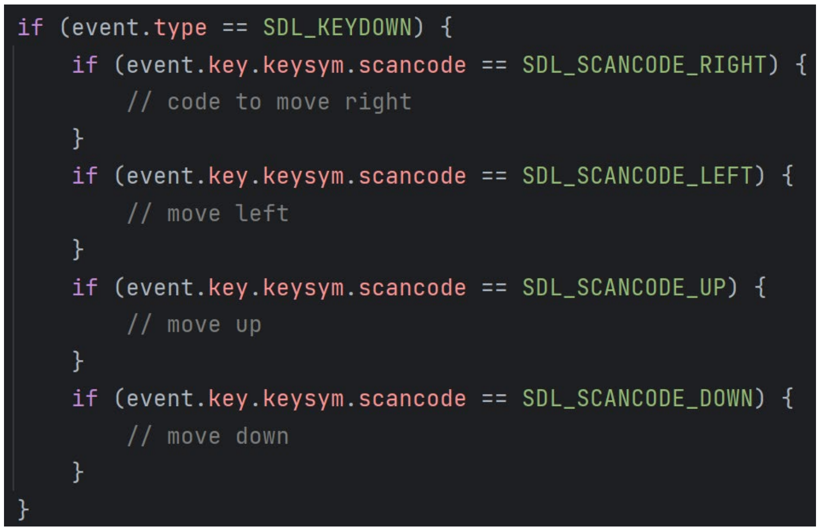
After checking the initial type of the event, if the event.type equals SDL_KEYDOWN, the next step is to find what key is pressed. This information is stored in:

```
event.key.keysym.scancode
```

We can check this value for a particular key: (this checks for the right arrow on the keyboard)

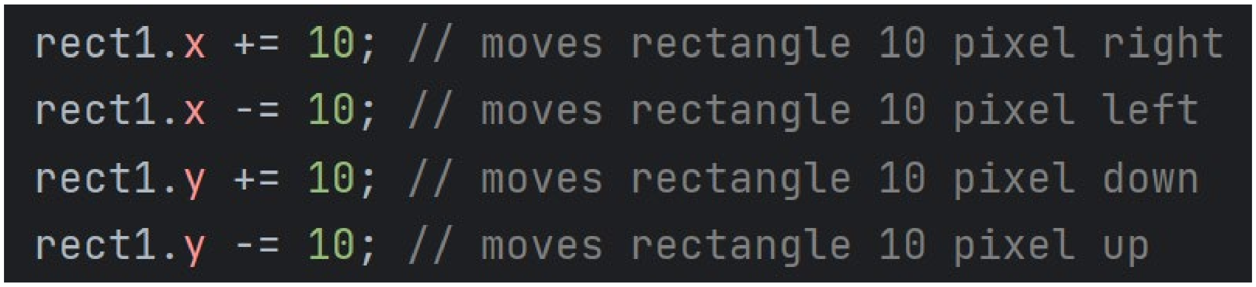
```
if (event.key.keysym.scancode == SDL_SCANCODE_RIGHT) {
    // code to move right
}
```

Similarly we can check for other keys too:



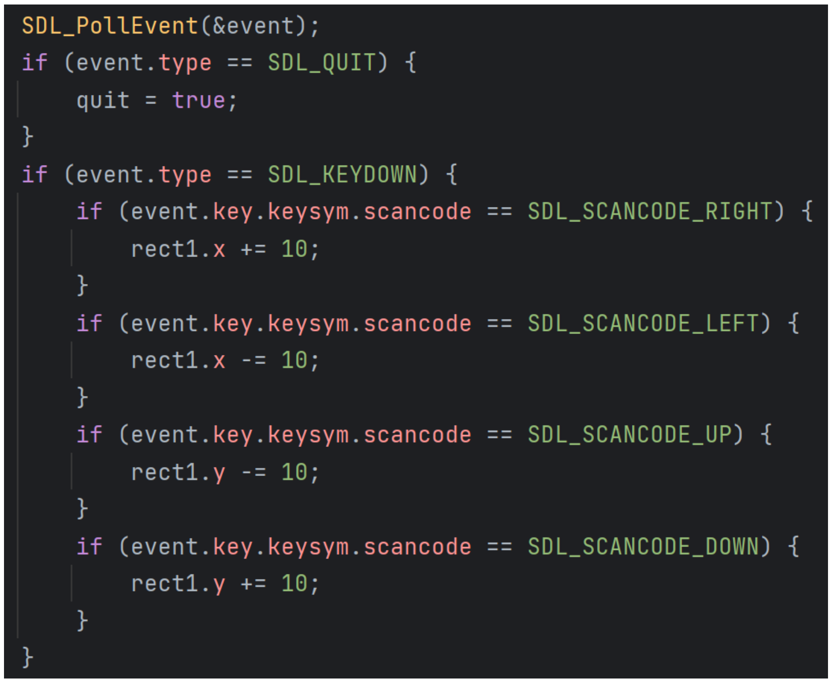
```
if (event.type == SDL_KEYDOWN) {
    if (event.key.keysym.scancode == SDL_SCANCODE_RIGHT) {
        // code to move right
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_LEFT) {
        // move left
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_UP) {
        // move up
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_DOWN) {
        // move down
    }
}
```

The code to move right, left, up or down is as simple as just changing the co-ordinates of our rectangle



```
rect1.x += 10; // moves rectangle 10 pixel right
rect1.x -= 10; // moves rectangle 10 pixel left
rect1.y += 10; // moves rectangle 10 pixel down
rect1.y -= 10; // moves rectangle 10 pixel up
```

If you're confused by the math in this, refer to the co-ordinate system of a computer in the picture in page 4 of this pdf. Finally we have (this code takes input and handles the processing of it):



```
SDL_PollEvent(&event);
if (event.type == SDL_QUIT) {
    quit = true;
}
if (event.type == SDL_KEYDOWN) {
    if (event.key.keysym.scancode == SDL_SCANCODE_RIGHT) {
        rect1.x += 10;
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_LEFT) {
        rect1.x -= 10;
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_UP) {
        rect1.y -= 10;
    }
    if (event.key.keysym.scancode == SDL_SCANCODE_DOWN) {
        rect1.y += 10;
    }
}
```

(Refer to github for the final code)