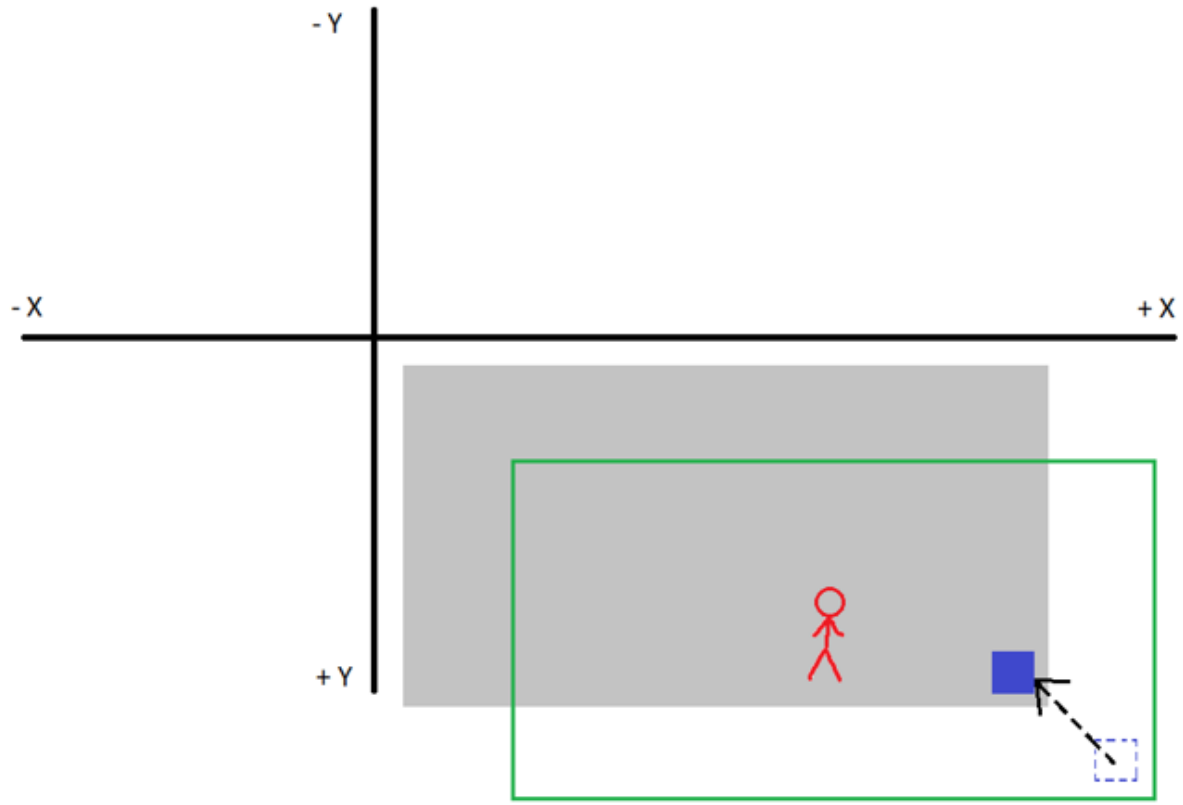
Implementing a Scrolling Camera in SDL2 / C++



By Darclander

2022-07-03

1. Introduction

I want to begin this paper by saying that all the content and its source code is open for anyone who would want to reuse it, help me maintain it, improve it or whatever comes to mind. The only requirement is that you refer to this paper or me IF you believe others will benefit from it.

It is also important to note that this implementation is not perfect but the idea is to give you a basic overview of *my idea* of what a scrolling camera can look like in SDL2. If you find something in the code which can be improved, please make a pull request here: https://github.com/darclander/sdl2-camera

With that being said, let us begin. This paper will go through the different parts of how things are rendered, why we are doing certain things and how we are able to create a scrolling camera. For those who are not familiar with SDL I would recommend watching John Hammond's videos on SDL which helped me understand the fundamentals: https://www.youtube.com/watch?v=iggmjJ_C_C4&list=PL1H1sBF1VAKXMz8kETLHRo1LwnvB08Q2J

2. The Main Program

To begin with we need to have a main program which will create a window, make us able to move that window around and stop if we press the “X”. The code provided in the figure below (which is also available on https://github.com/darclander/sdl2-camera) is our first step of the program.

```
#define WINDOW_WIDTH 1280
#define WINDOW_HEIGHT 720

int main(int argc, char *argv[]) {
    if (SDL_Init(SDL_INIT_VIDEO) > 0) {
        std::cout << "SDL, Error: " << SDL_GetError() << std::endl;
    }

    SDL_Event event;

    /* Instead of an fpscap we limit movement to time passed. */
    double deltaTime;
    Uint64 currentTick = SDL_GetPerformanceCounter();
    Uint64 lastTick = 0;

    Window window = Window("SDL2 Camera", WINDOW_WIDTH, WINDOW_HEIGHT);
    bool running = true;

    while(running) {

        /* Update deltaTime - see definition above. */
        lastTick = currentTick;
        currentTick = SDL_GetPerformanceCounter();
        deltaTime = (double)((currentTick - lastTick)*1000 / (double)SDL_GetPerformanceFrequency() );

        window.clear();
        window.display();

        /* To move window, etcetera... */
        while (SDL_PollEvent(&event)) {
            switch(event.type)
            {
            case SDL_QUIT:
                running = false;
                break;
            }
        }
    }
}
```

Figure 1 - *The first iteration of the main class.*

We create a basic while-loop which will make sure that if we press the X the window will close, that we are able to move certain things etcetera. The result of running this code will be seen in figure 3.

The second part of the main program is the window class which will assist us in doing everything that the window is “responsible” for. In many cases people will disagree on what classes should do and if classes are necessary. Therefore we want you to do as you please but we will be using the window class in our scenario.

The constructor which can be seen in figure 2 together with the functions *clear()* and *display()* are the most important ones for now. This allows us to clear the renderer and present the renderer. By definition a SDL_Renderer is: *"SDL_Renderer is a struct that handles all rendering. It is tied to a SDL_Window so it can only render within that SDL_Window. It also keeps track of the settings related to the rendering. There are several important functions tied to the SDL_Renderer"* [https://stackoverflow.com/questions/21007329/what-is-an-sdl-renderer].

```
Window::Window(const char* title, int w, int h) {

    window = SDL_CreateWindow(title, SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, w, h, SDL_WINDOW_SHOWN);

    if(window == NULL) {
        std::cout << "Could not init window, Error: " << SDL_GetError() << std::endl;
    }

    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_PRESENTVSYNC);
    worldPos.x = 0;
    worldPos.y = 0;

}
void Window::clear() {
    SDL_RenderClear(renderer);
}

void Window::display() {
    SDL_RenderPresent(renderer);
}
```

Figure 2 - *The source code for the window class.*

Note that there are two variables called *worldPos.x / y,* these will be used later for calculating the "real" position of the background. After compiling the code and running the executable in the debug folder (together with required DLLs) we achieve this result:

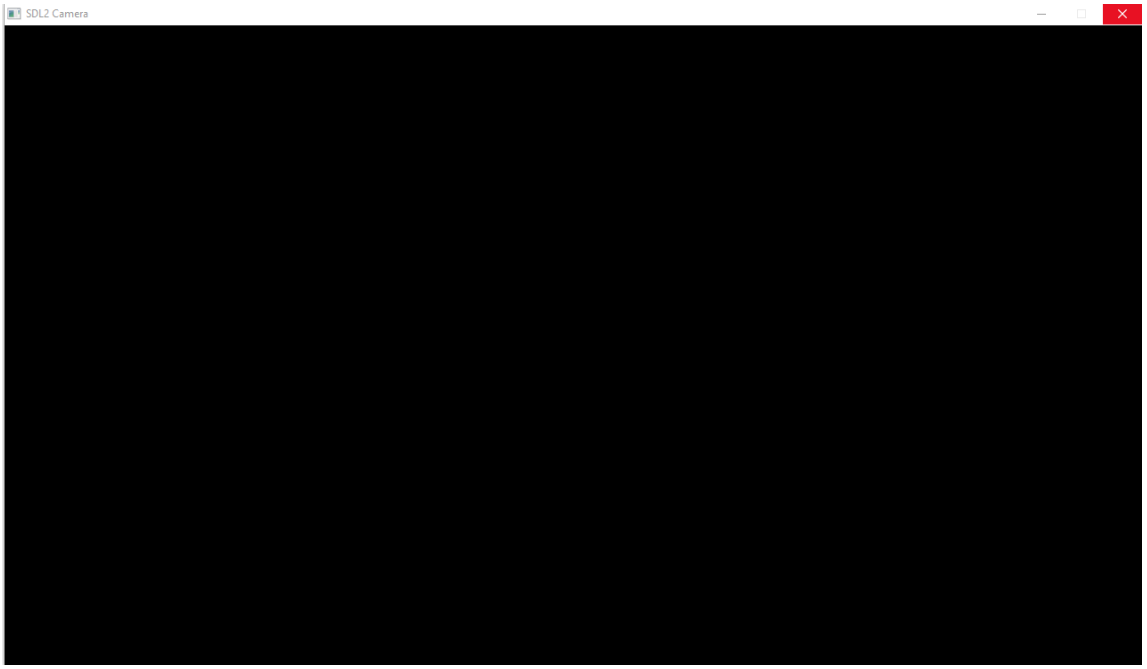


Figure 3 - *Result of running figure 1 and figure 2.*

3. Creating a Background

Right now our program does not look very interesting though, just a window with nothing inside of it. Let us change that. In our example we will make use of textures and move those around. To render a texture we will implement two new functions.

```
SDL_Texture *Window::loadTexture(const char* filePath) {
    SDL_Texture* texture = NULL;
    texture = IMG_LoadTexture(renderer, filePath);

    if (texture == NULL)
        std::cout << "Failed to load texture. Error: " << SDL_GetError() << std::endl;

    return texture;
}

void Window::render(int x, int y, SDL_Texture *texture) {
    SDL_Rect src;
    src.x = 0;
    src.y = 0;
    src.w;
    src.h;

    SDL_QueryTexture(texture, NULL, NULL, &src.w, &src.h);

    SDL_Rect dst;
    dst.x = worldPos.x - x;
    dst.y = worldPos.y - y;
    dst.w = src.w;
    dst.h = src.h;

    SDL_RenderCopy(renderer, texture, &src, &dst);
}
```

Figure 4 - *The window class implementation.*

We will use these functions in our main while-loop. After we have created the window we use the helper function *loadTexture()* in order to load a texture from a path on our computer. We then pass the texture we just loaded every time into the render function and we also specify the coordinates for it.

The code for the current main function can be seen in the image below. Note that *window.render()* should be between *window.clear()* and *window.display()*.

```
bool running = true;

Window window = Window("SDL2 Camera", WINDOW_WIDTH, WINDOW_HEIGHT);
SDL_Texture *backgroundTxt = window.loadTexture("../gfx/background.png");

while(running) {

    /* Update deltaTime - see definition above. */
    lastTick = currentTick;
    currentTick = SDL_GetPerformanceCounter();
    deltaTime = (double)((currentTick - lastTick)*1000 / (double)SDL_GetPerformanceFrequency() );

    window.clear();

    /* Function to render the background texture at coordinates X, Y. */
    window.render(0, 0, backgroundTxt);

    window.display();

    /* To move window, etcetera... */
    while (SDL_PollEvent(&event)) {
        switch(event.type)
        {
        case SDL_QUIT:
            running = false;
            break;
        }
    }
}
```

Figure 5 - *An iteration of the main file (incomplete).*

By compiling and running this code we have managed to make our program a bit more interesting. The result we achieve is this:

Figure 6 - *Result of the program after chapter(s) 2-3.*

4. Adding the Player

Similar to the other classes we create a class which will be responsible for the player's movement and rendering. We take help from a class called *Vector2* in order to perform operations easier. What happens in the player class is that we initialize a player at a given position (x, y) where we pass the renderer from the window and a file-path to the image we want to render the player as.

```
#include "player.hpp"

Player::Player(SDL_Renderer *r, float x, float y, const char* filePath) {
    pos = Vector2(x,y);
    renderer = r;

    SDL_Texture* texture = NULL;
    texture = IMG_LoadTexture(renderer, filePath);

    if (texture == NULL) {
        std::cout << "Failed to load texture. Error: " << SDL_GetError() << std::endl;
    }

    this->texture = texture;

}

Player::~Player() {
    std::cout << "Destroying player..." << std::endl;
}

void Player::render() {
    SDL_Rect src;
    src.x = 0;
    src.y = 0;


    SDL_QueryTexture(texture, NULL, NULL, &src.w, &src.h);

    SDL_Rect dst;
    dst.x = pos.x;
    dst.y = pos.y;
    dst.w = src.w;
    dst.h = src.h;

    SDL_RenderCopy(renderer, texture, &src, &dst);
}
```

Figure 7 - *The playerclass which is similar to previous classes.*

By implementing figure 7 we are able to load a player onto the screen but the issue is that without movement there is not much *playing* going on and therefore we want to add an additional function called *update().* Now there are two ways you can call update but in this paper we call it from the main function. The example used in this paper can be seen in figure 8.

```
39  void Player::update(double deltaTime) {
40
41      const Uint8 *key_state = SDL_GetKeyboardState(NULL);
42
43      if(key_state[SDL_SCANCODE_S]) {
44          pos = pos.operator+(Vector2(0, 1*0.1*deltaTime));
45      }
46
47      if(key_state[SDL_SCANCODE_W]) {
48          pos = pos.operator-(Vector2(0, 1*0.1*deltaTime));
49      }
50
51      if(key_state[SDL_SCANCODE_D]) {
52          pos = pos.operator+(Vector2(1*0.1*deltaTime, 0));
53      }
54
55      if(key_state[SDL_SCANCODE_A]) {
56          pos = pos.operator-(Vector2(1*0.1*deltaTime, 0));
57      }
58
59  }
```

Figure 8 - *The addition to the playerclass which allows us to move the player.*

When this has been implemented we have achieved a result similar to figure 9. We now have the basics of our program and we are able to move the player around. As you will notice though as you move around things are very static.

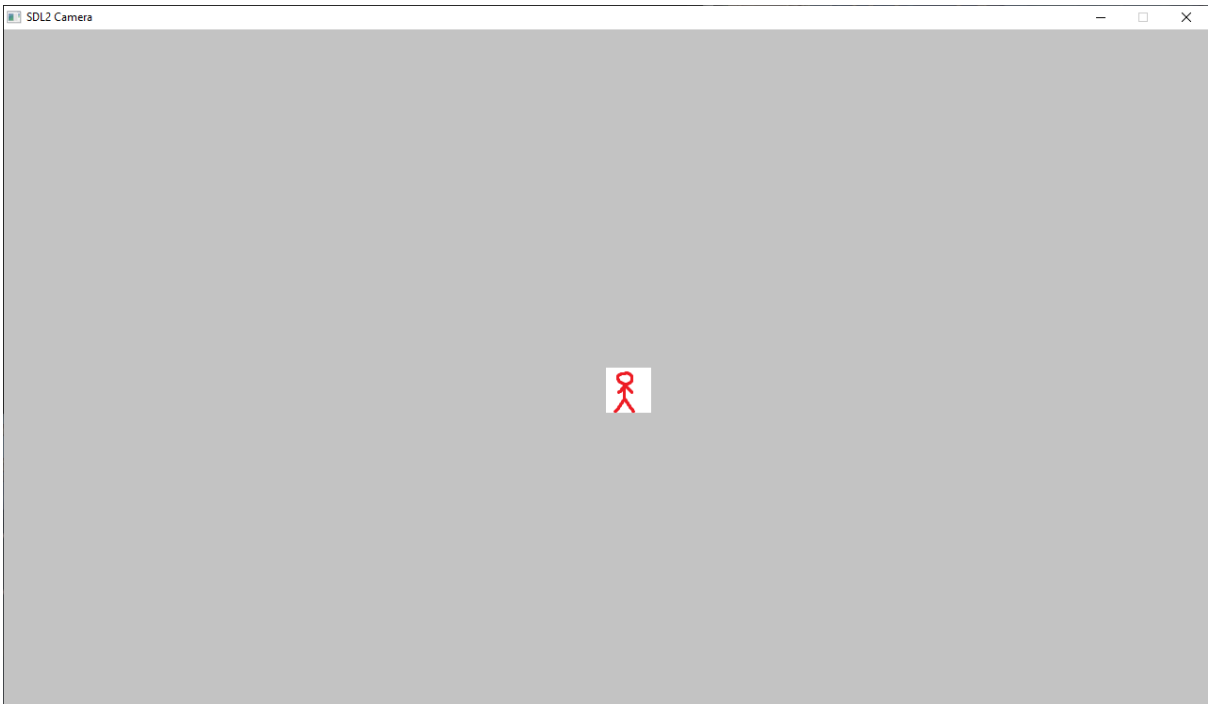


Figure 9 - *Result of following chapter(s) 4.*

5. Inserting a Reference Point

Most games might have something such as walls, chests, trees or similar which the player can interact with. If we only have 1920 x 1080 pixels to work with that wouldn't be such a fun game. Therefore we will insert a tile in the program just to make sure that the player is indeed moving in the world.

```cpp
#include "tile.hpp"

Tile::Tile(SDL_Renderer *r, float x, float y, const char* filePath) {
    worldPos = Vector2(x,y);
    renderer = r;
    SDL_Texture* texture = NULL;
    texture = IMG_LoadTexture(renderer, filePath);

    if (texture == NULL) {
        std::cout << "Failed to load texture. Error: " << SDL_GetError() << std::endl;
    }

    this->texture = texture;
}

void Tile::render() {

    float tileScreenX = screenPos.x;
    float tileScreenY = screenPos.y;

    /* View culling */
    if(tileScreenX > 0 && tileScreenY > 0 && tileScreenX < 1280 && tileScreenY < 720) {
        SDL_Rect src;
        src.x = 0;
        src.y = 0;


        SDL_QueryTexture(texture, NULL, NULL, &src.w, &src.h);

        SDL_Rect dst;
        dst.x = screenPos.x;
        dst.y = screenPos.y;
        dst.w = 48;
        dst.h = 48;

        SDL_RenderCopy(renderer, texture, &src, &dst);
    }

}

void Tile::update(SDL_Rect camera) {
    /* Update position relative to camera. */
    screenPos.x = worldPos.x - camera.x;
    screenPos.y = worldPos.y - camera.y;
}

Vector2 Tile::getPos() {
    return screenPos;
}
```

Figure 10 - *The tile class implementation.*

This class will function as a wall or a reference point in our world which is static (similar to how the background is static). We will then achieve the result in figure 11.

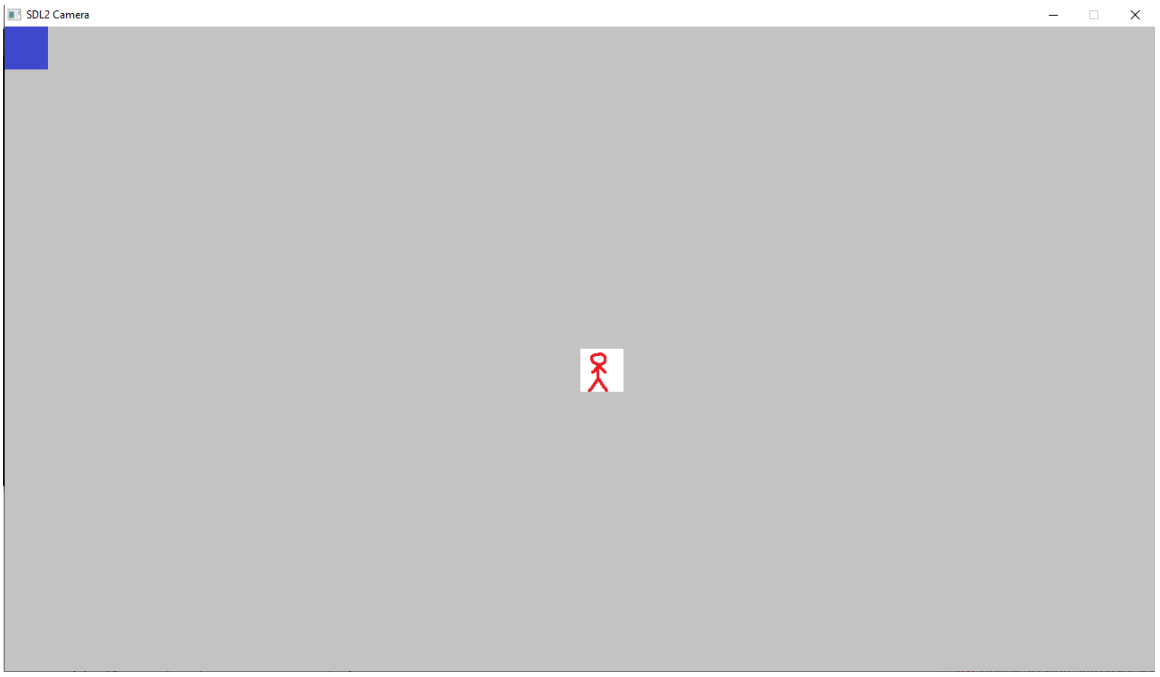


Figure 11 - *Final result of chapter(s) 2-5 where we now have everything we need.*

6. Making the Camera

What most people will probably notice now is that no matter how much we move the player we will always be within the 1920 x 1080 or the screen dimensions we set for the window. If we create another entity at for example (2000, 1100) it would not be seen on screen. Therefore in order to “expand” our world we want to make a camera or when we move around update the camera so we can see beyond the scope of our window size.

In order to do this we want to create a camera which updates according to our position (you can also use other factors which will update your camera but for simplicity the player location is used here). Therefore we will use the following line:

```
SDL_Rect camera = {0, 0, WINDOW_WIDTH, WINDOW_HEIGHT};
```

Figure 12 - *The camera implementation.*

This will function as our reference point for when we draw objects on the screen. We will also initialize every object with a worldpos and a screenpos. What this means is that an object will have a worldposition which is its *true* position and also a screenpos which is the position it should have on the screen.

To calculate an object's screenpos we subtract the camera’s X and Y position from the respectively worldpos X and Y position.

```
/* Update position relative to camera. */
screenPos.x = worldPos.x - camera.x;
screenPos.y = worldPos.y - camera.y;
```

Figure 13 - *Updating the screen-position of an object by subtracting the camera-position from the world-position.*

This needs to be done in every class and we will pass the camera position as an argument from the main function since we will also update the camera in main. The final loop of main will look something similar to:

```
while(running) {

    /* Update deltaTime - see definition above. */
    lastTick = currentTick;
    currentTick = SDL_GetPerformanceCounter();
    deltaTime = (double)((currentTick - lastTick)*1000 / (double)SDL_GetPerformanceFrequency() );

    window.clear();

    /* Update camera position. */
    camera.x = player.getWorldPos().x - (WINDOW_WIDTH / 2); /* Centralize around X axis. */
    camera.y = player.getWorldPos().y - (WINDOW_HEIGHT / 2); /* Centralize around Y axis. */

    /* Function to render the background texture at coordinates X, Y. */
    /* Update position relative to camera. */
    window.render(camera.x + (WINDOW_WIDTH/2), camera.y + (WINDOW_HEIGHT/2), backgroundTxt);

    /* Update player and render. Camera is sent in to calculate screenPosition. */
    player.update(deltaTime, camera);
    player.render();

    tile.update(camera);
    tile.render();

    window.display();

    /* To move window, etcetera... */
    while (SDL_PollEvent(&event)) {
        switch(event.type)
        {
        case SDL_QUIT:
            running = false;
            break;
        }
    }
}
```

Figure 14 - *Final implementation of main from chapter(s) 2-6*

To summarize this step we make a camera with positions based on the player position which is in the middle of the screen, hence the line:

$$player.getWorldPos().x \; - \; (WINDOW\_WIDTH \; / \; 2);$$

Afterwards each entity is rendered on the screen based on their relative position to the camera position. This means that for example if a tile has a position on the negative X and Y axis they are outside of the screen and will therefore not be rendered on the screen. If their worldpos instead lies within the camera-position the formula will yield a location which will be within the screen space and will be rendered.

6.1 View Culling

What can be really good about this is that we can implement something called *view-culling* **[3]**. This means that entities which are not meant to be seen by the player will not be rendered, because what is the point of rendering a tile at (-9999, -9999)? For example with the tile in our solution. If we do not want to render it when the player cannot see it we can implement this line of code before we render the tile:

```
float tileScreenX = screenPos.x;
float tileScreenY = screenPos.y;

/* View culling */
if(tileScreenX > 0 && tileScreenY > 0 && tileScreenX < 1280 && tileScreenY < 720) {
```

Figure 15 - *View culling implementation.*

Then we only render the tile if it is inside the area we want it to be seen within. For example It will not render at the start of the program because the tile position we check is (0, 0) and $0 > 0$ is not true which means it will not be rendered. If we move up a bit to the left it will be rendered though because now the topmost left corner of the camera is negative (x, y) and therefore $0 > -1$ for instance.

If there are any uncertainties I recommend looking through the git repository **[1]** which will contain all the code necessary to complete this project yourself.

7. Conclusion and Summary

Now you finally have a scrolling camera of your own!

8. References

[1] - https://github.com/darclander/sdl2-camera

[2] - https://www.youtube.com/watch?v=iggmjJ_C_C4&list=PL1H1sBF1VAKXMz8kETLHRo1LwnvB08Q2J

[3] - https://www.lighthouse3d.com/tutorials/view-frustum-culling/