CPSC6050 Final Project
Implementation of Spritesheet Animation in OpenGL Graphics API

Instructor: Dr. Daljit Singh Dhillon

Student Name: Siddhant Gupta

Project Goal:

Implement page flip animation using 2D sprite sheets. To bring some life to the black screen using animated rendered sprites.

Video Reel / Result:

https://drive.google.com/file/d/18zqW-VzvVc1wIeM-YEJK2H-Tzp4aKupV/view?usp=sharing

Controls:

Press and hold "A" or "D" to move left or right respectively.

Project Description:

Idea is to render a single image/actor in different poses so as to give the illusion of animation/movement. The implementation uses a technique called slicing of spritesheet in the fragment shader. Spritesheet is a single png image containing multiple frames of the same actor in different poses (think like page flip animation) arranged in rows and column orderly manner with uniform padding.
The project is written in a framework style of programming where a lot of helper classes have been constructed/designed to handle a lot of boiler plate OpenGL code so as to keep the implementation file (main.cpp) as simple as possible and reduce redundancy in code.

Some of the helper classes that the framework provides are :

SGTexture : Handles all the code required to load JPG and PNG files using stb_image library. It maintains ready to use texture data as unsigned int, which can be passed to the shader program readily.

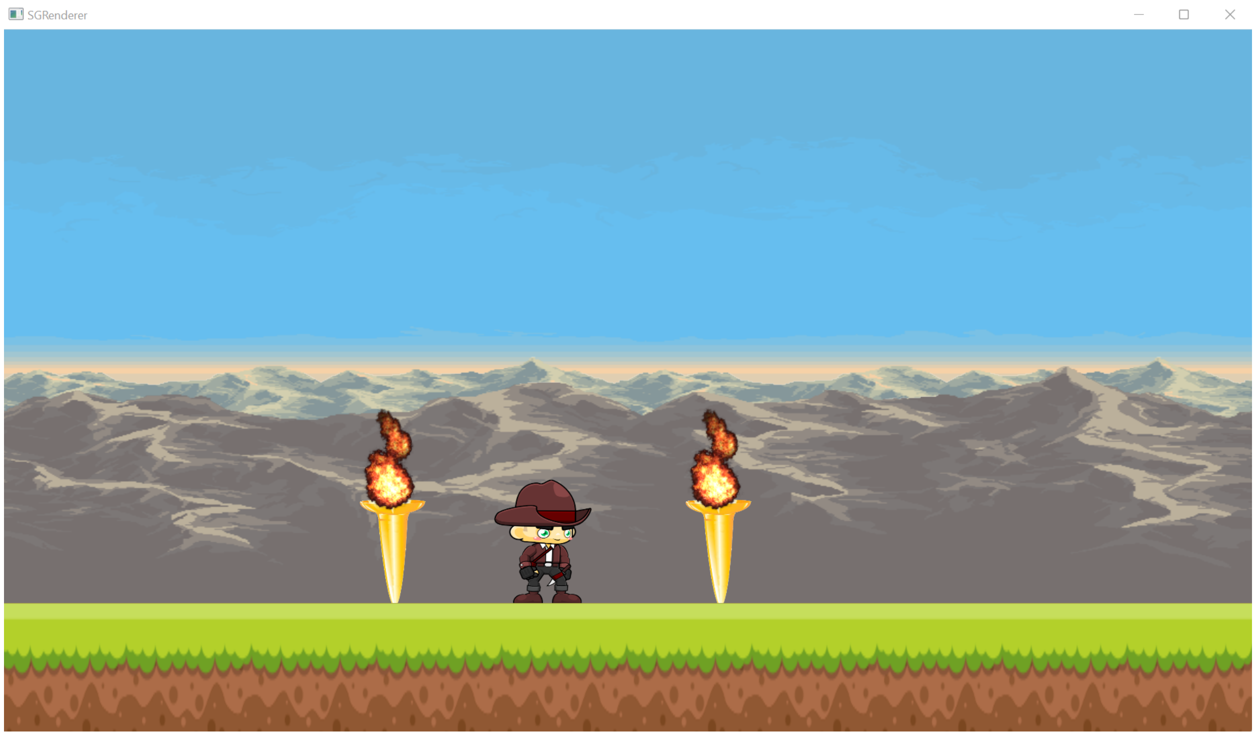
SGShader : Handles all the code required to read/parse both vertex and fragment files and convert them into OpenGL linkable shader programs. Attaches and Links the GLSL code to the shader program. Handles creation of the shader program. Textures generated using SGTexture class can be attached/linked to the shader code using this class, hence it maintains a list of all the textures being used by the shader code and their respective GL_TEXTURE(Index). Handles texture binding before draw calls.

SGSprite : Handles all the code required to generate a 2D quad with position and texture coordinates as vertex buffer data. Generates the vertex buffer object and a vertex array object with correct vertex attribute pointers linking the vertex buffer data to the GLSL vertex shader.

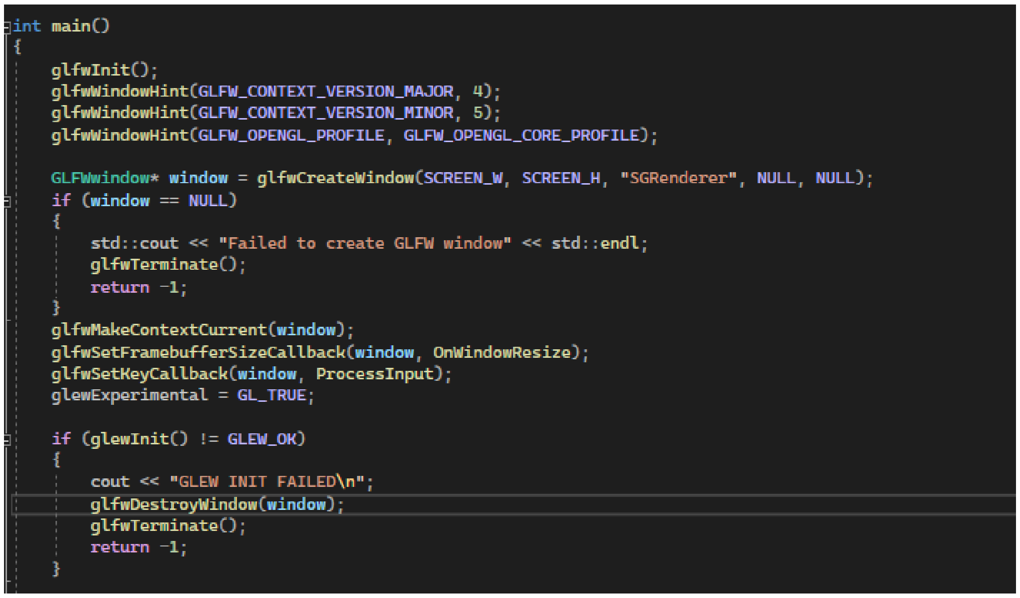
This class also computes the model matrix data and links it to the shader code. Handles drawing of each sprite object by binding the corresponding vertex array object.

SGUtility : Contains pre-processor directives, const values and includes that are helpful and required by other classes in the project.

Workflow:

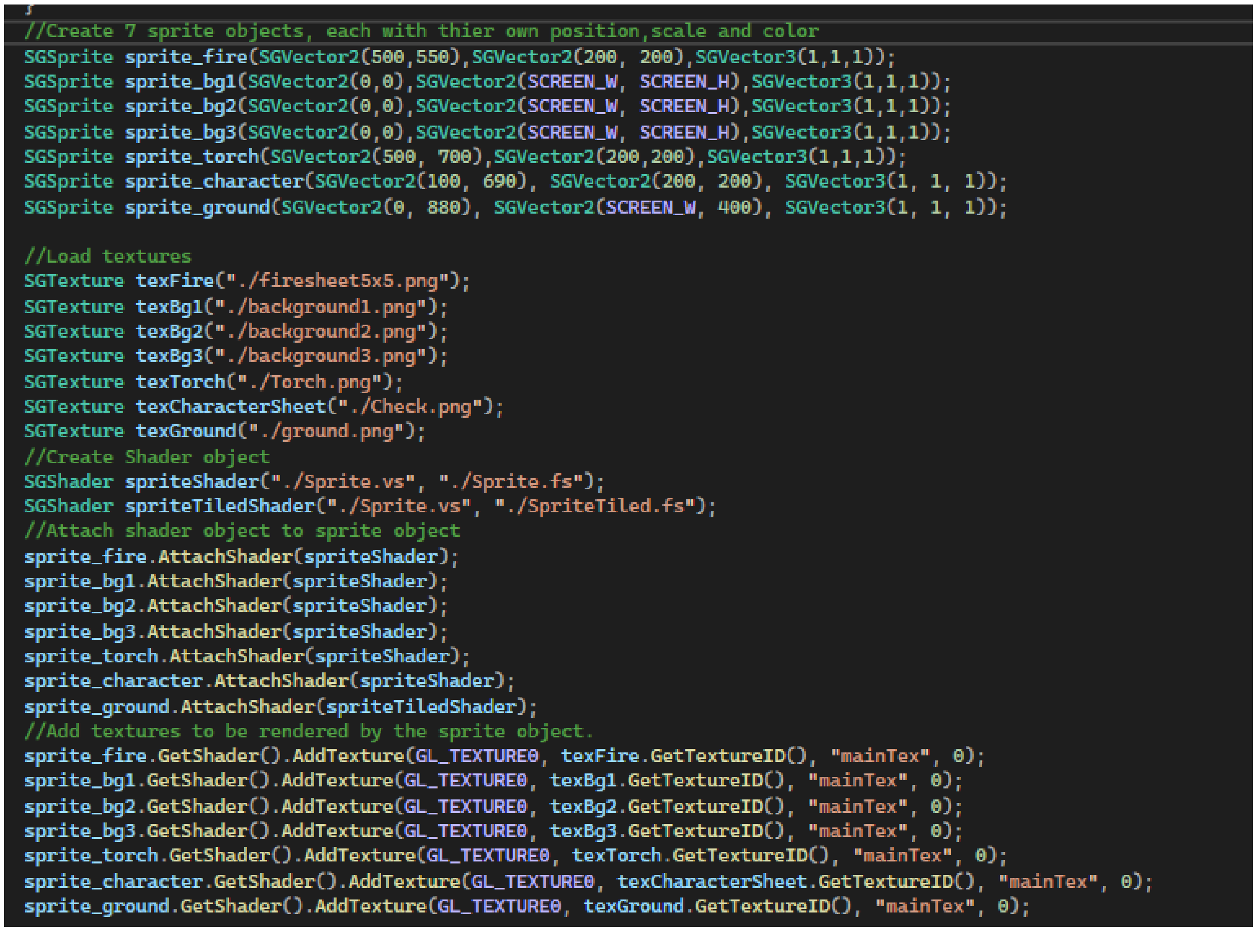


The scene renders an animated character with Idle and Run animation implementation. Two torch objects with animated fire sprites. The character can be controlled using keys A and D to move left and right respectively. For aesthetics the scene also renders ground sprite on which the player walks and three parallaxed background sprites.
To render the above scene the project uses main.cpp as implementation file. GLFW API is used to handle all the windows API functionality like creating a window and grabbing the handle. Two openGL callback functions are referenced glfwSetFrameBufferSizeCallback() is called each time the window is resized and glfwSetKeyCallback() handles every key input.



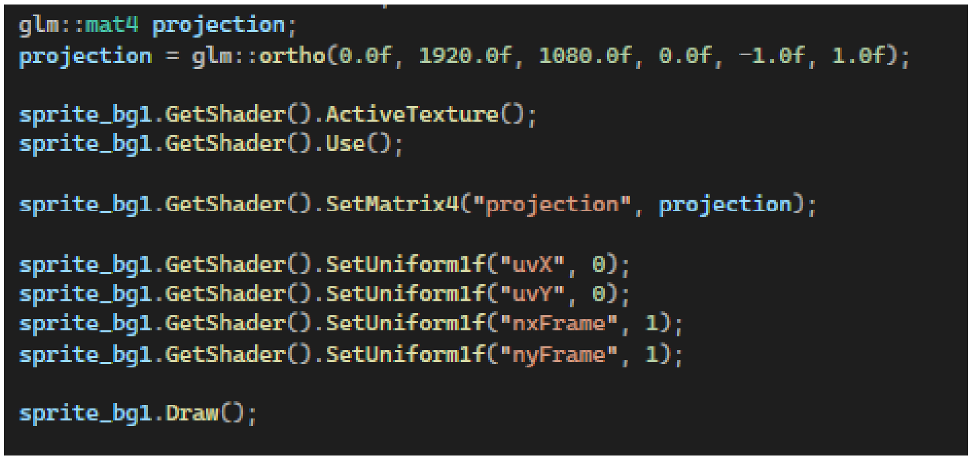
```
int main()
{
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* window = glfwCreateWindow(SCREEN_W, SCREEN_H, "SGRenderer", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, OnWindowResize);
    glfwSetKeyCallback(window, ProcessInput);
    glewExperimental = GL_TRUE;

    if (glewInit() != GLEW_OK)
    {
        cout << "GLEW INIT FAILED\n";
        glfwDestroyWindow(window);
        glfwTerminate();
        return -1;
    }
```

We then create seven 2D sprite objects using SGSprite helper class, seven texture objects using SGTexture class and two shader objects using SGShader class. The sprite object contains a shader member variable that holds the corresponding shader to be used to render the sprite. We attach a corresponding shader object using SGSprite::AttachShader() function and then add texture to be rendered by the sprite. All these classes really simplify the process of generating VBO,VAO, shader program objects and texture objects and reduce a lot of redundancy in the code.



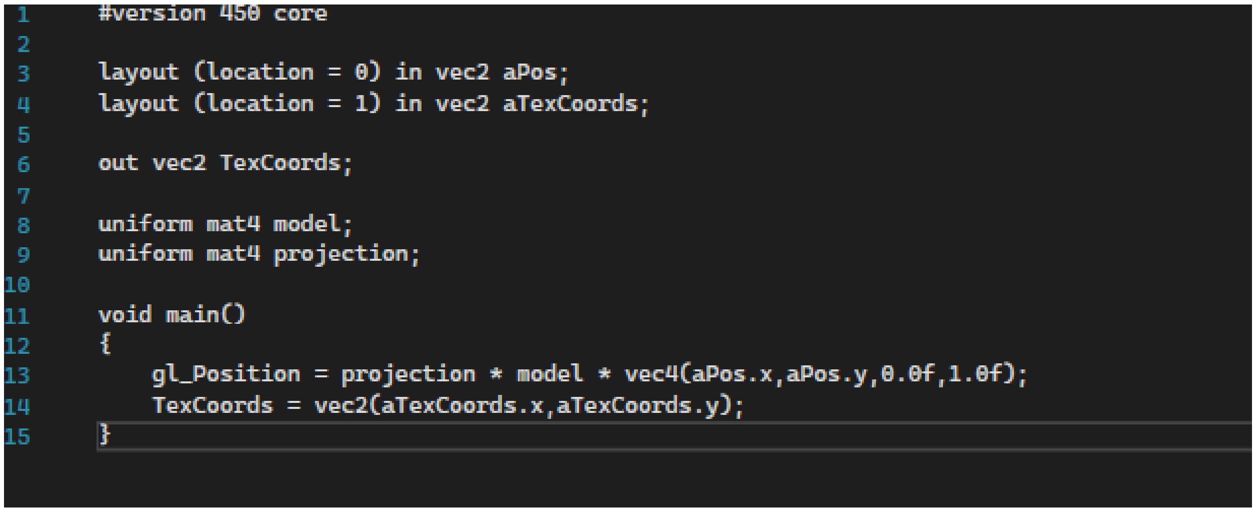
```
//Create 7 sprite objects, each with thier own position,scale and color
SGSprite sprite_fire(SGVector2(500,550),SGVector2(200, 200),SGVector3(1,1,1));
SGSprite sprite_bg1(SGVector2(0,0),SGVector2(SCREEN_W, SCREEN_H),SGVector3(1,1,1));
SGSprite sprite_bg2(SGVector2(0,0),SGVector2(SCREEN_W, SCREEN_H),SGVector3(1,1,1));
SGSprite sprite_bg3(SGVector2(0,0),SGVector2(SCREEN_W, SCREEN_H),SGVector3(1,1,1));
SGSprite sprite_torch(SGVector2(500, 700),SGVector2(200,200),SGVector3(1,1,1));
SGSprite sprite_character(SGVector2(100, 690), SGVector2(200, 200), SGVector3(1, 1, 1));
SGSprite sprite_ground(SGVector2(0, 880), SGVector2(SCREEN_W, 400), SGVector3(1, 1, 1));

//Load textures
SGTexture texFire("./firesheet5x5.png");
SGTexture texBg1("./background1.png");
SGTexture texBg2("./background2.png");
SGTexture texBg3("./background3.png");
SGTexture texTorch("./Torch.png");
SGTexture texCharacterSheet("./Check.png");
SGTexture texGround("./ground.png");
//Create Shader object
SGShader spriteShader("./Sprite.vs", "./Sprite.fs");
SGShader spriteTiledShader("./Sprite.vs", "./SpriteTiled.fs");
//Attach shader object to sprite object
sprite_fire.AttachShader(spriteShader);
sprite_bg1.AttachShader(spriteShader);
sprite_bg2.AttachShader(spriteShader);
sprite_bg3.AttachShader(spriteShader);
sprite_torch.AttachShader(spriteShader);
sprite_character.AttachShader(spriteShader);
sprite_ground.AttachShader(spriteTiledShader);
//Add textures to be rendered by the sprite object.
sprite_fire.GetShader().AddTexture(GL_TEXTURE0, texFire.GetTextureID(), "mainTex", 0);
sprite_bg1.GetShader().AddTexture(GL_TEXTURE0, texBg1.GetTextureID(), "mainTex", 0);
sprite_bg2.GetShader().AddTexture(GL_TEXTURE0, texBg2.GetTextureID(), "mainTex", 0);
sprite_bg3.GetShader().AddTexture(GL_TEXTURE0, texBg3.GetTextureID(), "mainTex", 0);
sprite_torch.GetShader().AddTexture(GL_TEXTURE0, texTorch.GetTextureID(), "mainTex", 0);
sprite_character.GetShader().AddTexture(GL_TEXTURE0, texCharacterSheet.GetTextureID(), "mainTex", 0);
sprite_ground.GetShader().AddTexture(GL_TEXTURE0, texGround.GetTextureID(), "mainTex", 0);
```

We don't need any perspective applied to the coordinates, since we are rendering 2D sprites. Hence we are creating an orthographic projection matrix. We then set our texture target and bind the texture to be rendered by the next draw call using SGShader’s corresponding functions. After setting the projection uniform matrix in GLSL code and other uniforms that we will discuss later, we invoke SGSprite::Draw() function which handles the binding of vertex array object and also sets the correct model matrix and then draws the sprite on the framebuffer.



```
glm::mat4 projection;
projection = glm::ortho(0.0f, 1920.0f, 1080.0f, 0.0f, -1.0f, 1.0f);

sprite_bg1.GetShader().ActiveTexture();
sprite_bg1.GetShader().Use();

sprite_bg1.GetShader().SetMatrix4("projection", projection);

sprite_bg1.GetShader().SetUniform1f("uvX", 0);
sprite_bg1.GetShader().SetUniform1f("uvY", 0);
sprite_bg1.GetShader().SetUniform1f("nxFrame", 1);
sprite_bg1.GetShader().SetUniform1f("nyFrame", 1);

sprite_bg1.Draw();
```

The vertex shader is pretty straightforward; it takes in two vector2 values aPos and aTexCoords from the the vertex buffer data which provide the local vertex position and local texture coordinates of the 2D quad respectively. The shader outputs a vector2 TexCoords which is same as the input vector aTexCoords and gl_Position vector4 which is the transformed vertex from local space to normalized device coordinates. The shader also uses two matrix4 values called model and projection which transform the vertex from local to model space and then to screen space respectively.



```
#version 450 core

layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 projection;

void main()
{
    gl_Position = projection * model * vec4(aPos.x,aPos.y,0.0f,1.0f);
    TexCoords = vec2(aTexCoords.x,aTexCoords.y);
}
```

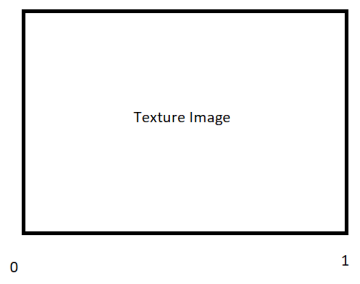
The fragment shader is where the real magic happens. It takes an input vector2 from the vertex shader which are the texture coordinates and outputs a vector4 FragColor which is the final color of the fragment. Uniform sampler2D mainTex contains the albedo texture to be sampled using the input texture coordinates.
The main 4 variables that slice the sprite sheet and render each frame in “X” frame per second are :
uvX : frame increment along the X axis
uvY : frame increment along the Y axis
nxFrame : total number of frames along the X axis
nyFrame : total number of frames along the Y axis

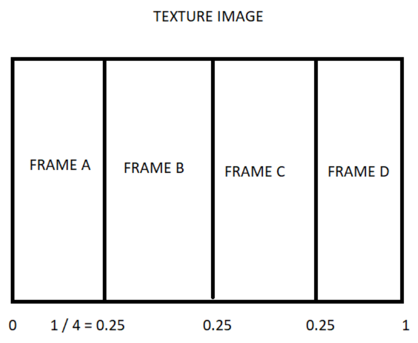
Algorithm :

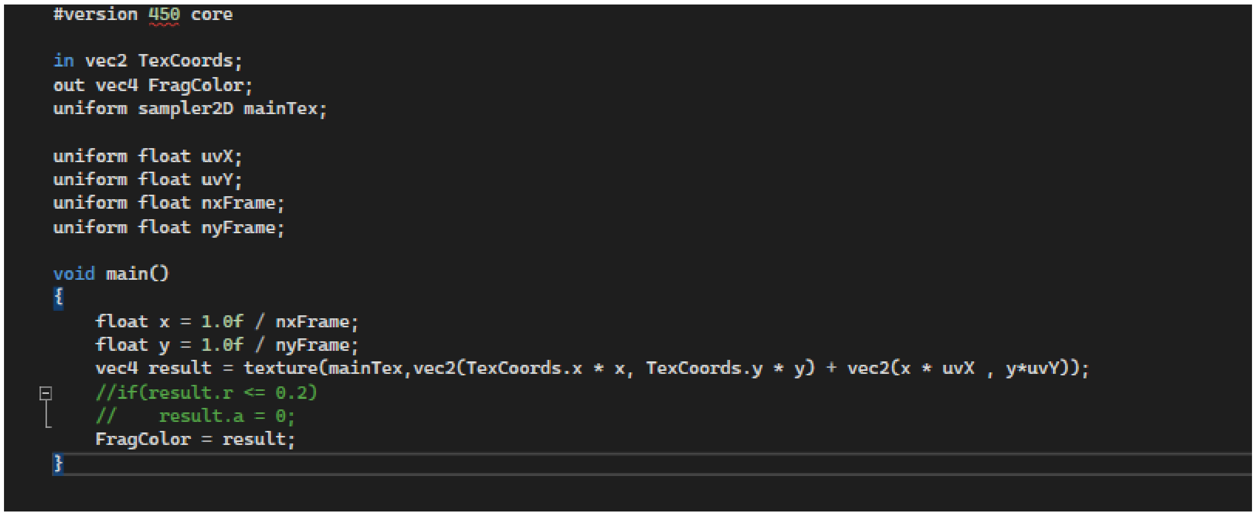
Let us discuss the algorithm used.
X = 1 / nxFrame and Y = 1/ nyFrame, calculate the uv coordinates at which the next frame starts. This basically slices the sheet into rows and columns.

Example if a sheet has 4 frames along the x axis and we know that the uv values range from 0-1 . So to calculate at what “u” value intervals will each frame start at, we will divide
1 / 4 = 0.25

Which gives us four 0.25 intervals i.e each frame in the sheet spans from 0-0.25 , 0.25-0.50 . 0.50-0.75 and so on till we reach 1.0 . Thereby slicing the sheet into 4 frames each of length 0.25 in "uv" coordinates.





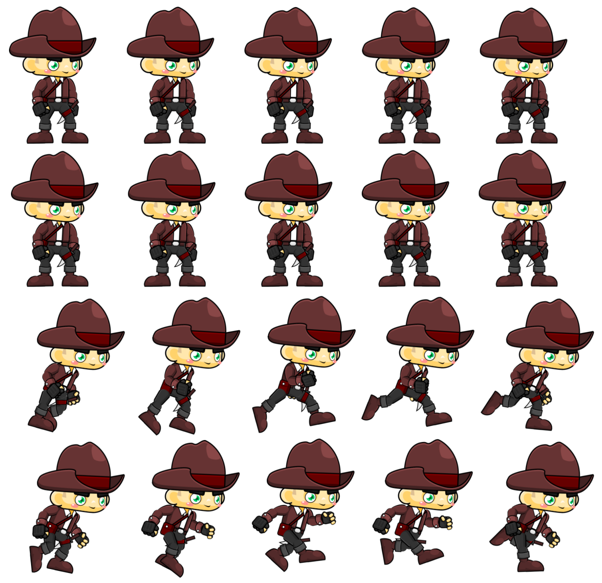


```
#version 450 core

in vec2 TexCoords;
out vec4 FragColor;
uniform sampler2D mainTex;

uniform float uvX;
uniform float uvY;
uniform float nxFrame;
uniform float nyFrame;

void main()
{
    float x = 1.0f / nxFrame;
    float y = 1.0f / nyFrame;
    vec4 result = texture(mainTex,vec2(TexCoords.x * x, TexCoords.y * y) + vec2(x * uvX , y*uvY));
    //if(result.r <= 0.2)
    //    result.a = 0;
    FragColor = result;
}
```

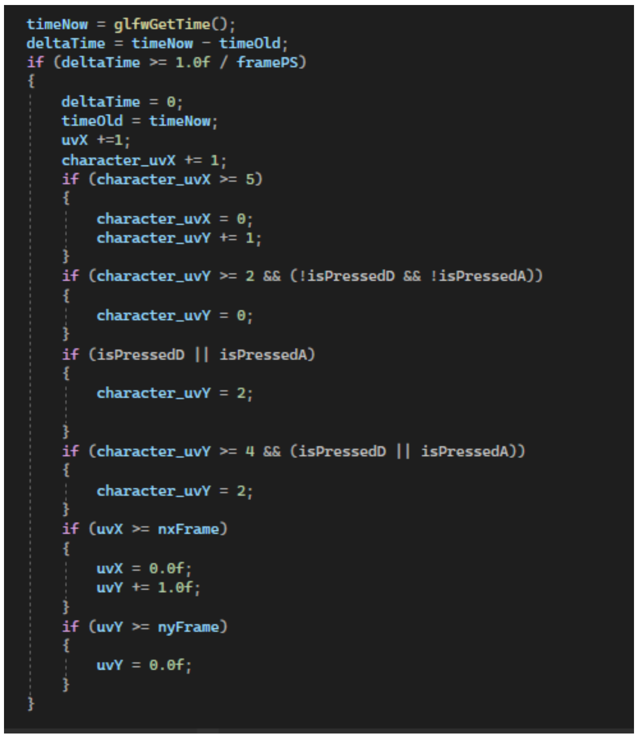
So we sample our "mainTex" using the above modified texture coordinates (TexCoords) and then we add to it increments of uvX and uvY (ranging from 0 to number of rows and columns), both incrementing by 1 each time we have to switch the frame. Using this technique the fragment shader only renders a part of the sheet one by one in "X" frames per second to give the illusion of animation/movement.

Below is an example sprite sheet rendered by the project. For the below sheet the values of :
nxFrame = 5
nyFrame = 4
uvX = 0 to 5
uvY = 0 to 2 for idle animation

uvY = 2 to 4 for walk animation



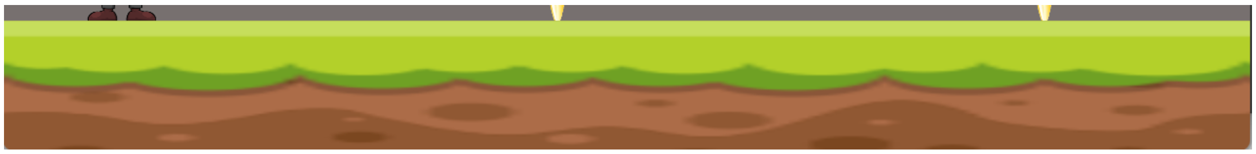
uvX and uvY values are modified in the while() loop using calculated deltaTime variable to ensure smooth transition and uniform timing. We calculate the time it takes to render a frame (deltaTime), add it up and compare it to 1 / framePS. Where framePS is the speed in frames per second of the sprite animation. Default value has been set to 25, so each time deltaTime is greater than 1 / 25, the fragment shader will move to the next frame in the sliced sheet. Hence rendering/playing the animation 25 frames per second.



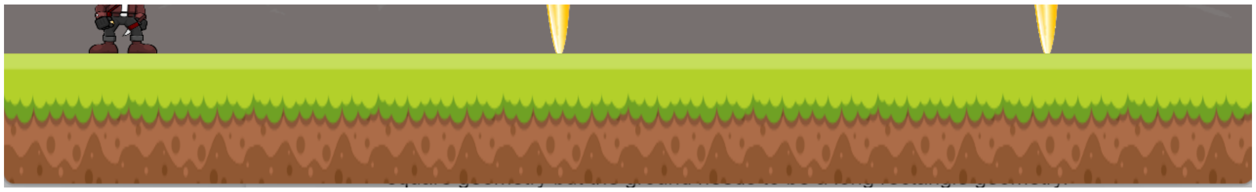
```
timeNow = glfwGetTime();
deltaTime = timeNow - timeOld;
if (deltaTime >= 1.0f / framePS)
{
    deltaTime = 0;
    timeOld = timeNow;
    uvX +=1;
    character_uvX += 1;
    if (character_uvX >= 5)
    {
        character_uvX = 0;
        character_uvY += 1;
    }
    if (character_uvY >= 2 && (!isPressedD && !isPressedA))
    {
        character_uvY = 0;
    }
    if (isPressedD || isPressedA)
    {
        character_uvY = 2;

    }
    if (character_uvY >= 4 && (isPressedD || isPressedA))
    {
        character_uvY = 2;
    }
    if (uvX >= nxFrame)
    {
        uvX = 0.0f;
        uvY += 1.0f;
    }
    if (uvY >= nyFrame)
    {
        uvY = 0.0f;
    }
}
```

Extra feature:
The project also implements another fragment shader called SpriteTiled.fs which implements a texture uv tilling function. This shader is used to render the texture of the ground sprite as the quad is 2D square geometry but the ground needs to be a long rectangle geometry. Without tilling the texture stretches to compensate for lack of resolution/pixels along the X axis. With tilling set to value “X”, the texture repeats X number of times from 0 to 1 texture coordinates of the quad.

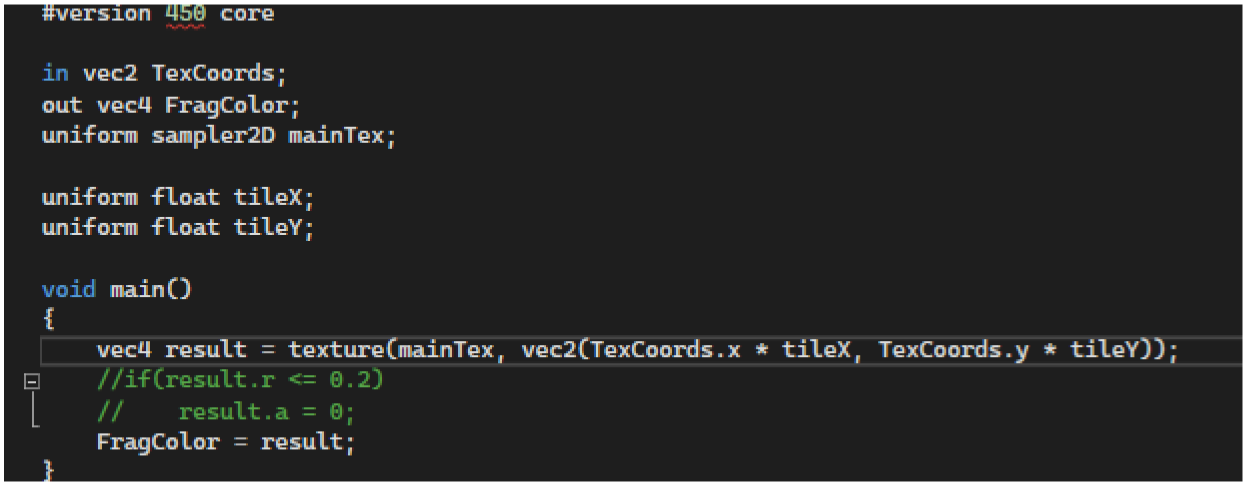
Without Tiling



With Tiling set to 10 along the X/U coordinate of the texture space.



Tiling Shader : SpriteTiled.fs
Uniform float tileX and tileY are used to repeat the texture on the quad between 0 - 1.



```
#version 450 core

in vec2 TexCoords;
out vec4 FragColor;
uniform sampler2D mainTex;

uniform float tileX;
uniform float tileY;

void main()
{
    vec4 result = texture(mainTex, vec2(TexCoords.x * tileX, TexCoords.y * tileY));
    //if(result.r <= 0.2)
    //    result.a = 0;
    FragColor = result;
}
```

Tools and API used :
The project is accomplished using API such as GLEW, GLFW, OpenGL, C/C++ standard libraries and tools such as Visual Studio Community 2022 (IDE for Windows) and Visual Studio Code (for Linux).

Technical Learning :
- Vertex Buffer Objects
- Vertex Array Objects

- Vertex Attribute Array Pointers
- Vertex Shader
- Fragment Shader
- Graphics Pipeline
- Linear Transformations
- Projection and Model transformations
- Texture Sampling
- OpenGL pipeline to create and bind texture objects before draw calls
- UV mapping
- Creating GL_VERTEX_SHADER , GL_FRAGMENT SHADER
- Compile, link and attach created shaders to the shader program object
- Setting shader uniforms
- OpenGL keyboard input
- Framework style approach encouraged applying Object Oriented Principles in C++.

Conclusion :

Fragment shaders can be used to achieve unique effects and implement a lot of creative techniques when it comes to rendering textures and various other types of image effects.