Real-Time Rendering Techniques



Full Screen Anti-Aliasing

[30077548@students.southwales.ac.uk](mailto:30077548@students.southwales.ac.uk?subject=Rule%20Based%20Systems%20Report)

Sebastian Ha

30077548

Table of Contents

[Table of Acronyms 4](#_Toc220103545)

[Introduction 5](#_Toc220103546)

[Scene Setup and Rendering Pipeline 6](#_Toc220103547)

[Scene Description 6](#_Toc220103548)

[Rendering pipeline 6](#_Toc220103549)

[Shaders 6](#_Toc220103550)

[Texture usage 6](#_Toc220103551)

[Performance Tracking 6](#_Toc220103552)

[Base scene without Anti-Aliasing 7](#_Toc220103553)

[Visual notes 7](#_Toc220103554)

[Screenshots 7](#_Toc220103555)

[Performance 8](#_Toc220103556)

[Multi-Sample Anti-Aliasing (MSAA) 9](#_Toc220103557)

[How does MSAA work? 9](#_Toc220103558)

[How MSAA was enabled in the scene 9](#_Toc220103559)

[Sample counts 9](#_Toc220103560)

[Screenshots 10](#_Toc220103561)

[Performance and visual notes 11](#_Toc220103562)

[Super-Sample Anti-Aliasing (SSAA) 12](#_Toc220103563)

[Implementation 12](#_Toc220103564)

[Sampling resolutions 12](#_Toc220103565)

[Screenshots 13](#_Toc220103566)

[Performance cost 14](#_Toc220103567)

[Comparison and evaluation 15](#_Toc220103568)

[Conclusion 15](#_Toc220103569)

[Appendix 16](#_Toc220103570)

[Asset licenses 16](#_Toc220103571)

[Rubik’s Cube 16](#_Toc220103572)

[Moon 16](#_Toc220103573)

[Grass 16](#_Toc220103574)

[Code tutorial Sources 17](#_Toc220103575)

[GitHub Repository 17](#_Toc220103576)

[Code 18](#_Toc220103577)

[Scene Renderer Class 18](#_Toc220103578)

[Scene FBO Class 26](#_Toc220103579)

[Basic Shader 30](#_Toc220103580)

[Source File 33](#_Toc220103581)

# Table of Acronyms

|  |  |
| --- | --- |
| FSAA | Full-Screen Anti-Aliasing |
| MSAA | Multi-Sample Anti-Aliasing |
| SSAA | Super-Sample Anti-Aliasing |
| AA | Anti-Aliasing |
| VBOs | Vertex Buffer Objects |
| VAOs | Vertex Array Objects |
| FPS | Frames Per Second |
| FBO | Frame Buffer Object |

# Introduction

Anti-Aliasing is a technique used in computer graphics that reduces aliasing effects by smoothing jagged edges or stair-like steps on curved lines and diagonals on computer screens. This matters in real-time rendering as these effects become more noticeable especially during movement within the rendered scene, which can cause the visual quality of the scene to drop. Anti-aliasing helps rendered objects appear more realistic due to smoother edges. Two full-screen anti-aliasing techniques (FSAA) will be compared against a base 3D real-time rendered scene. The techniques compared are Multi-Sample Anti-Aliasing (MSAA) and Super-Sample Anti-Aliasing (SSAA).

MSAA is a type of FSAA that takes multiple depth samples within a single pixel without needing to recalculate lighting or textures. On the other hand, SSAA renders the whole scene at a higher resolution and combines multiple samples per pixel. This increases the performance cost and the visual quality. The goal of this report is to compare the quality and the performance between both FSAA techniques against a base 3D scene that has no anti-aliasing (AA) techniques applied. The comparison will determine which AA technique is better suited for modern real-time rendering applications.

# Scene Setup and Rendering Pipeline

## Scene Description

The scene is constructed of a simple 3D setup consisting of a textured ground grass plane, a sphere representing a moon and a Rubik’s cube. The background of the scene is a flat blue colour instead of a skylight or background, making sure nothing affects the AA technique comparisons. The scene was kept simple intentionally so that comparisons could be shown clearly using fewer contrasting models. Overcomplication could reduce how clearly the results are shown from the different aliasing effects.

## Rendering pipeline

The rendering pipeline consists of the scene implementing OpenGL version 3.3 Core. GLFW is used for configuration, input callback functions and window creation while GLAD is used to load all OpenGL function pointers. Geometry is stored within Vertex Buffer Objects (VBOs), objects are then configured using Vertex Array Objects (VAOs). Depth testing is enabled ensuring visibility and occlusion are correct within the scene, otherwise the perception of depth would not be portrayed correctly if parts of an object were not rendered in order. GLFW disables vertical synchronisation using swap intervals to show the frame rate of the scene. The camera class provides an object that displays the view and projection matrices, allowing the user to see the scene from a viewport that can also be controlled.

## Shaders

The shaders used within this scene were a basic shader and a Phong lighting shader, each implemented as vertex and fragment shaders. The basic shader is used on the Rubik’s cube and grass plane models to render the textured objects where detailed surface lighting is not necessary. In contrast, the moon model is rendered using a Phong shader for per-fragment lighting. This allows smooth shading across the surface of the sphere which produces more realistic results. These shaders were chosen to show a clear contrast between simple and complex shading methods while ensuring the scene remained appropriate for comparing AA techniques.

## Texture usage

The models within the scene were texture mapped using PNG image files loaded at run time and sampled in the fragment shader. Material files were not used, as implementing image textures allowed the scene setup to be more controlled and simpler so focus could be on the comparison of different anti-aliasing techniques.

## Performance Tracking

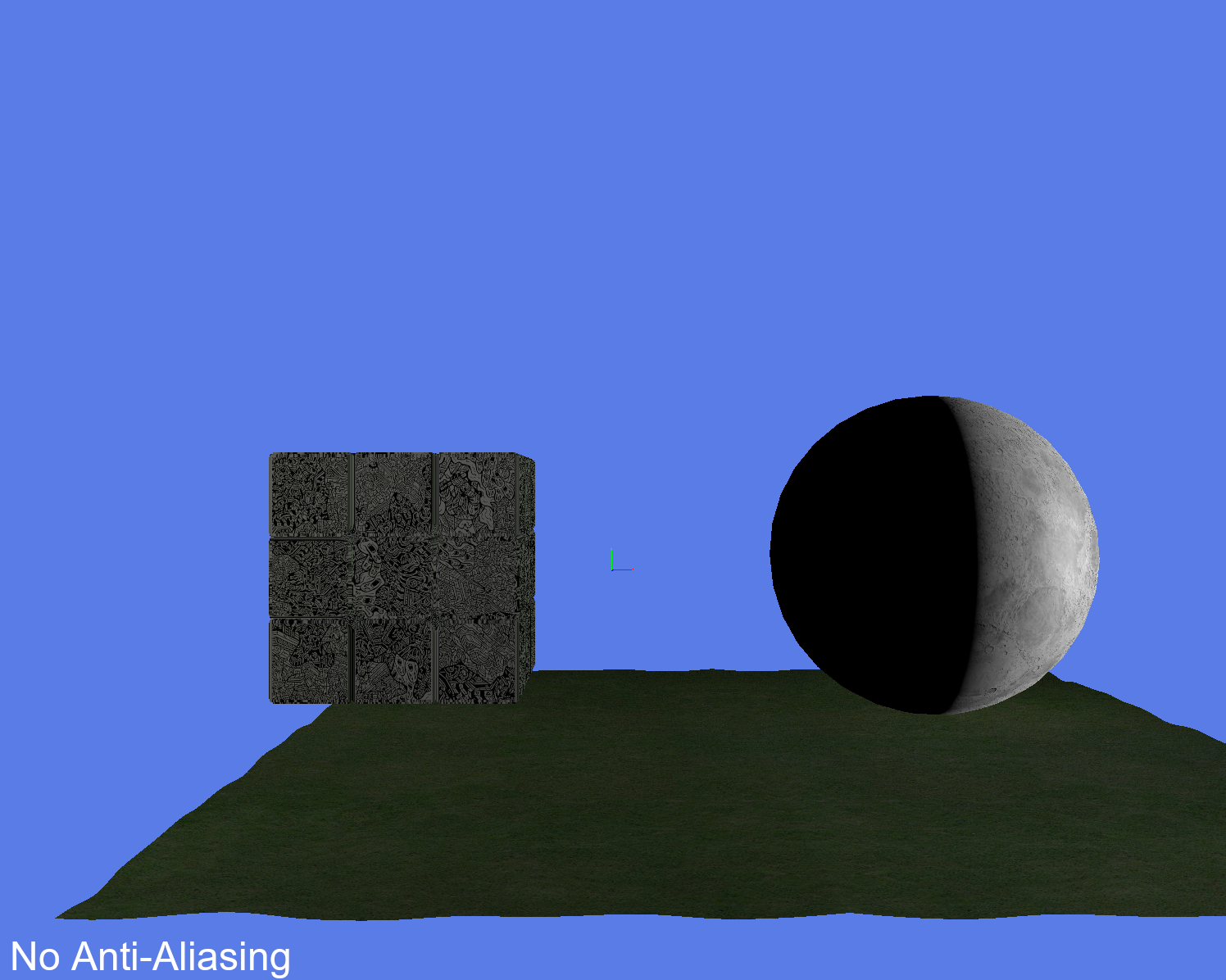
To track performance within the scene when different AA effects are applied to the scene, a tracker was implemented that tracked the frames per second (FPS) and how many milliseconds each frame takes. These values are updated once a second and displayed on the scene’s window title, this allows performance comparisons in real time.

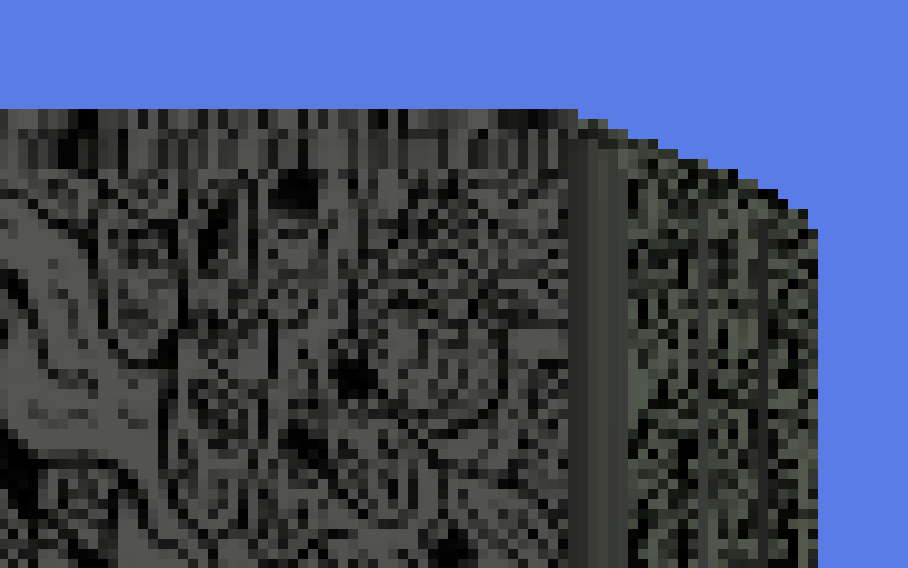
# Base scene without Anti-Aliasing

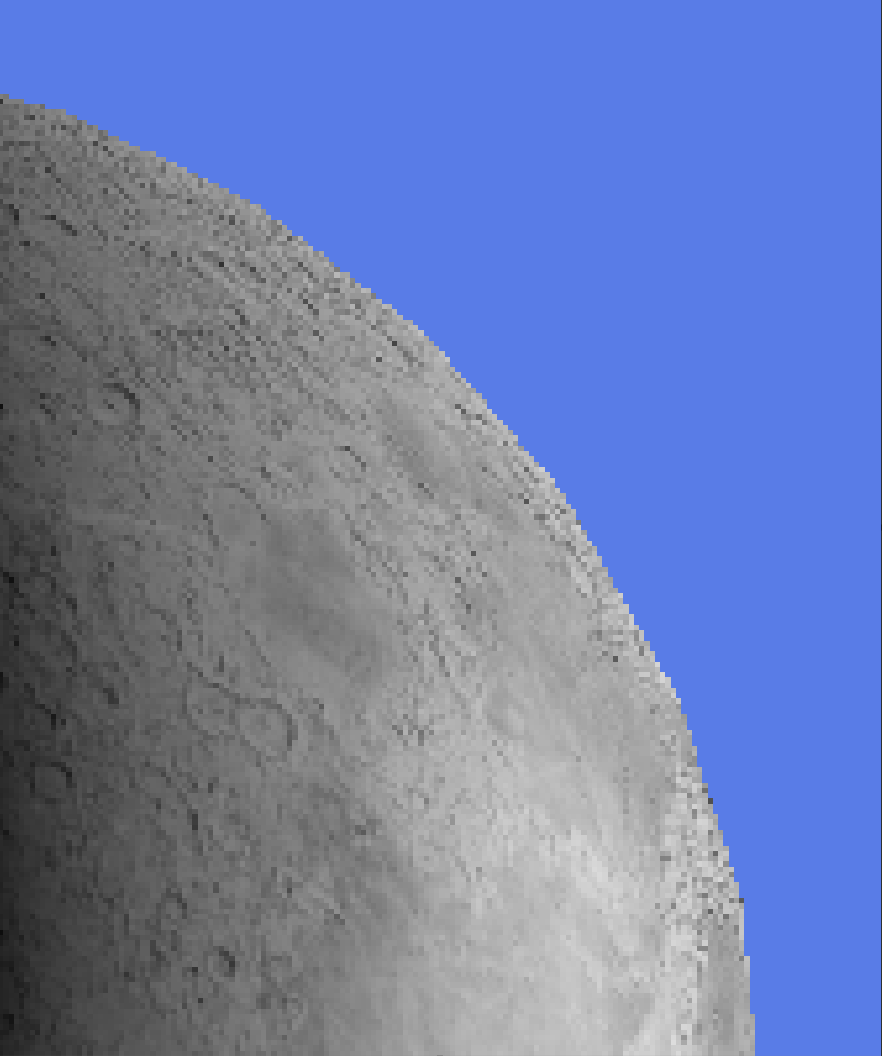
## Visual notes

Upon rendering the scene, jagged edges along the models are prominent, with the grass model being the most affected due to the uneven surface. Models that are curved, such as the moon show noticeable stair-stepping effects along the outlines. When the moon model is rotating around its own axis, shimmering effects are observed as the jagged edges change relative to the camera view making the edge appear unstable.

## Screenshots









## Performance

The scene’s FPS performed within a range of five thousand to seven thousand five hundred frames. This shows the scene sustains a high-performance level even with the three models rendered. When moving around the scene, the frame rate drops to around five thousand FPS, but when the camera is still the frames can range between six thousand upwards to seven and a half thousand.

# Multi-Sample Anti-Aliasing (MSAA)

## How does MSAA work?

MSAA works by having multiple depth samples taken at different sub-pixel locations within a pixel. These samples share the same texture and lighting calculations while allowing different depth values per sample. This reduces the computational overhead significantly compared to SSAA. MSAA is widely used in modern game engines due to its ability to balance between quality and performance.

## How MSAA was enabled in the scene

MSAA was enabled during the initialisation and configuration of GLFW. The number of samples was specified before the window was created.



Once the window was created, multi-sampling was handled using OpenGL. This allowed the scene to enable and disable multi-sampling when needed. No changes were required to the shaders used within the scene as MSAA is handled internally by the hardware.

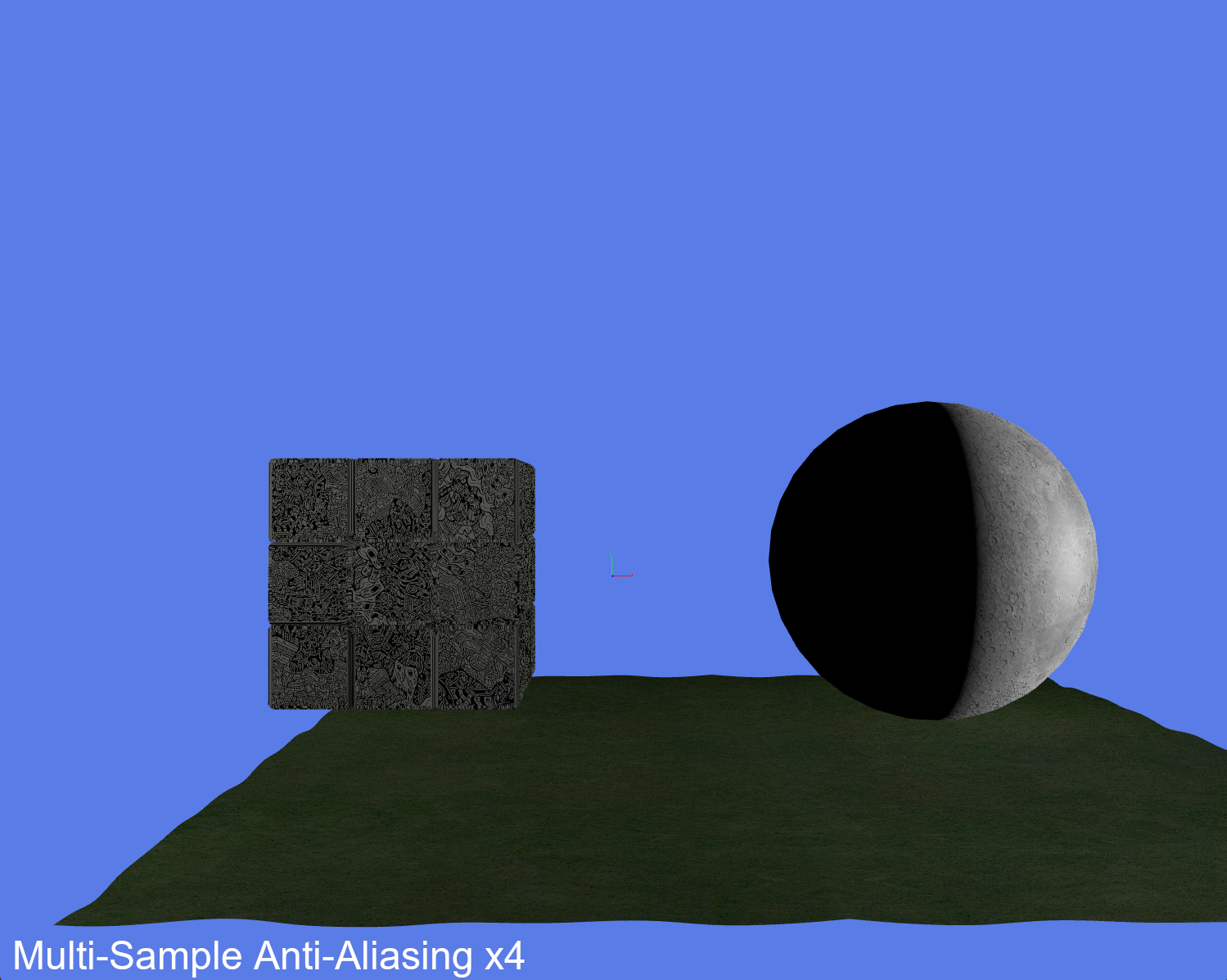


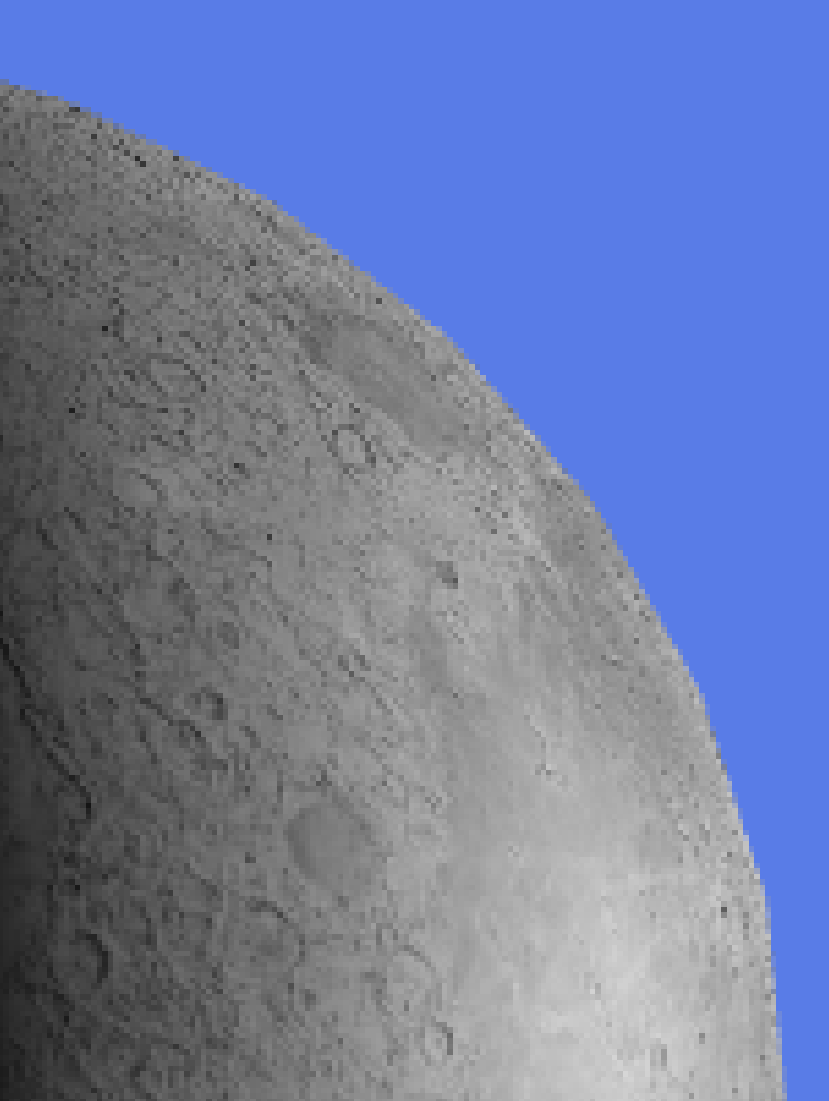
## Sample counts

During development, multiple sample counts were tested, including 2x, 4x, and 8x sampling. As the sample count increased, edge smoothing became more prominent, especially on curved model edges. However, higher sample counts increased the performance cost, with minimal differences visually after 4x multi-sampling.

From the observations made, 4x MSAA was chosen for the final screenshots. This was so performance remained high while seeing a clear improvement in visual quality compared to the base scene that had no AA.

## Screenshots







## Performance and visual notes

Enabling MSAA resulted in an average of four thousand one hundred to four thousand three hundred FPS. Although this shows a reduction of around one thousand FPS, there was still a considerably high-performance rate within the scene while displaying noticeably smoother object edges. MSAA helped to reduce aliasing artefacts along non-straight edges and improving visual stability during camera movement within the scene.

# Super-Sample Anti-Aliasing (SSAA)

## Implementation

SSAA requires off-screen rendering where a framebuffer object (FBO) is used to render the scene before it is displayed to the viewport. Rendering to the FBO allows the scene to be displayed at a higher resolution. The render target resolution attached to the FBO is higher than the window resolution. The higher resolution produces multiple samples per output pixel, resulting in improvements to the visual quality of the scene. However, this increase in quality comes at a significant performance cost, as SSAA has a substantial amount of computational overhead.

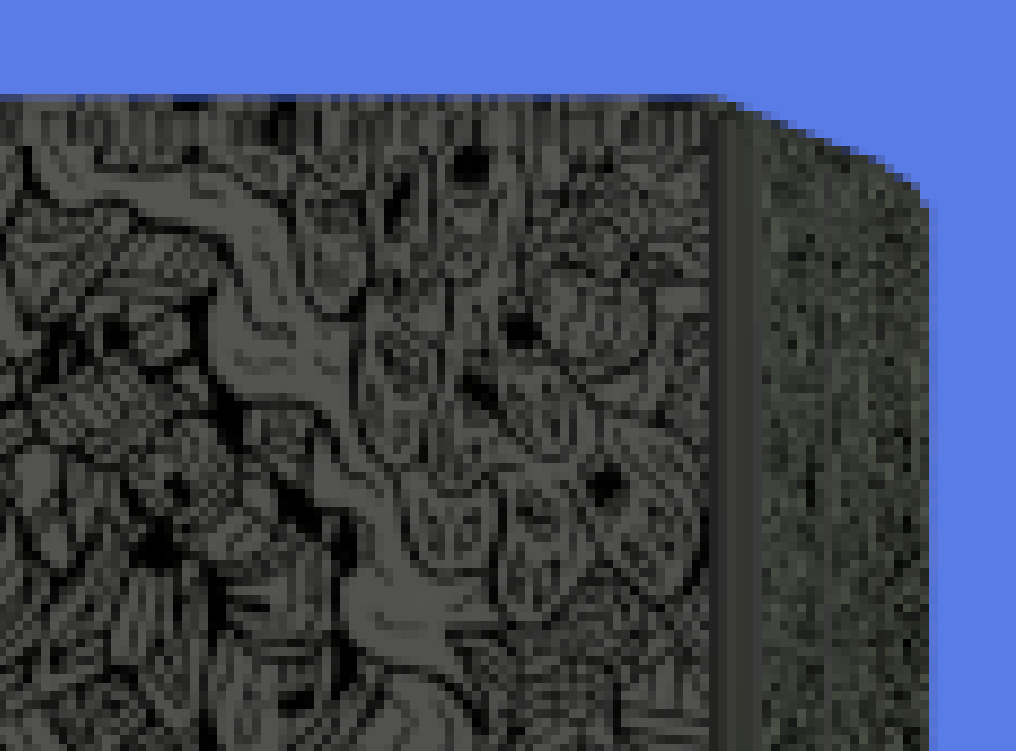
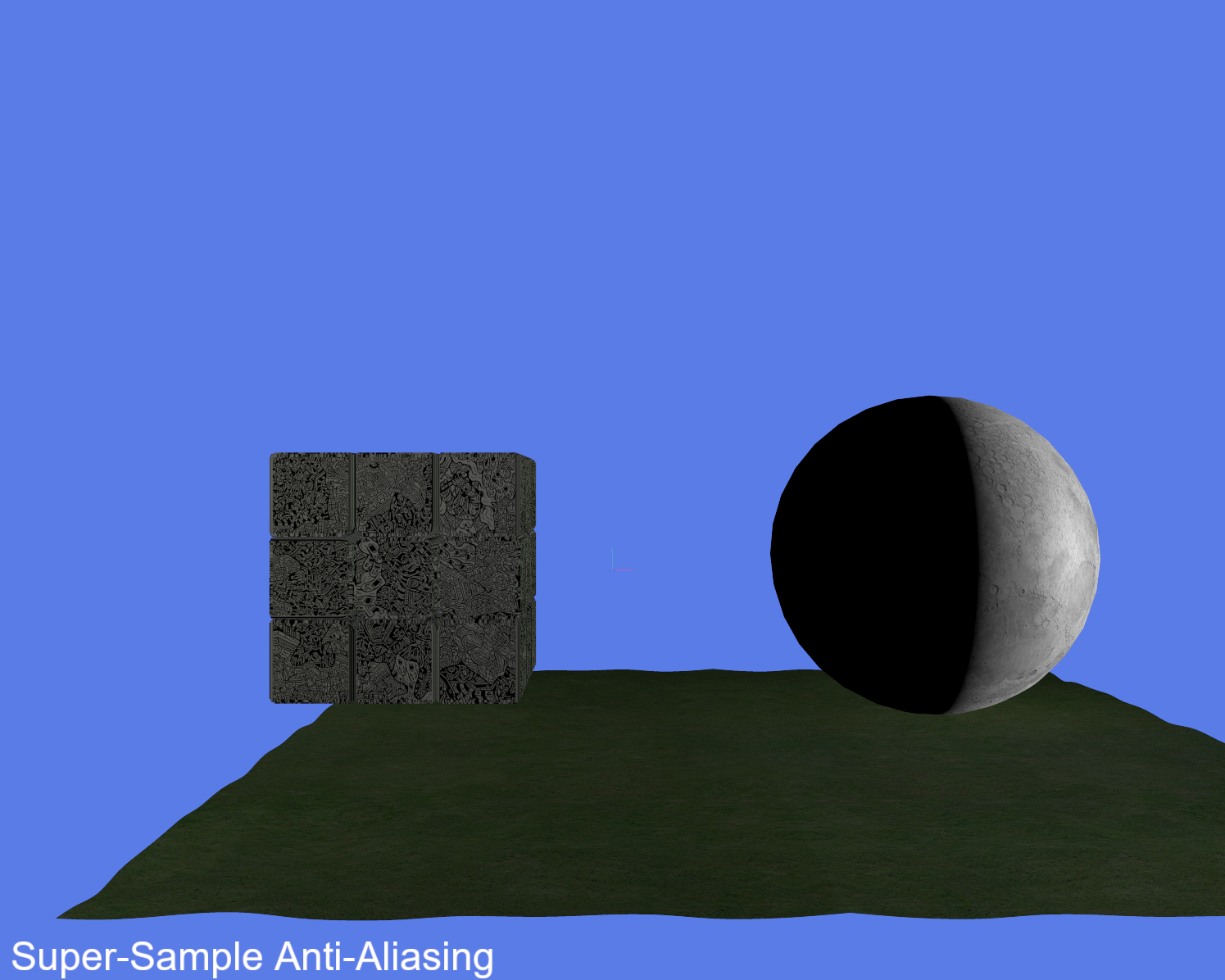
Rendering the scene with SSAA consists of two passes. The scene is rendered to the off-screen FBO at a higher resolution in the first pass, which is then used by the second to render the scene to the viewport. Without the initial pass to the FBO, the scene is unable to be rendered using SSAA properly.

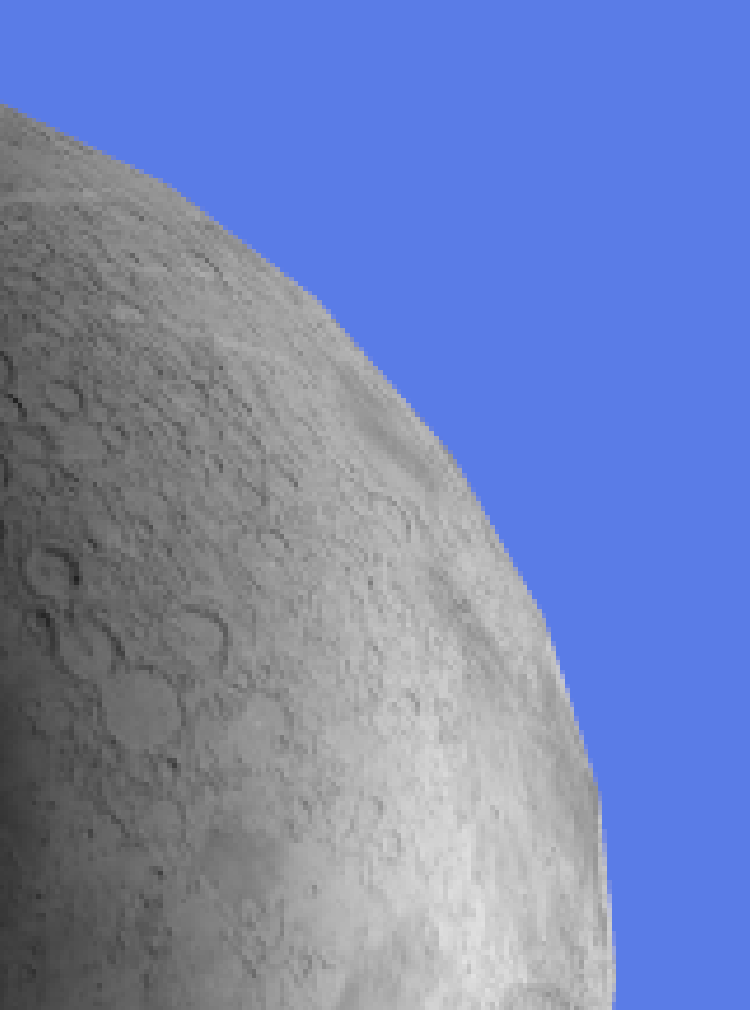
The high-resolution texture attached to the FBO was rendered using a textured quad that was aligned to the screen and a basic texture shader. No modifications were made to the shader, which down sampled the higher resolution image during the second pass to the resolution of the viewport window. By ensuring the quad aligned to the screen, the scene rendered correctly and camera movement was still possible as if looking at the scene rendered without AA or MSAA.

## Sampling resolutions

When testing, 2x, 4x and 8x resolutions were explored during development. Although, 8x resolution was only briefly evaluated as the performance cost was significantly higher with no additional visual improvements at all. Due to this, SSAA 4x was chosen for comparisons to other AA techniques.

## Screenshots





## Performance cost

When enabled, the scene performs within a range of five hundred to one thousand two hundred FPS. This significant reduction in performance was due to the scene being rendered at a higher resolution, which increased the number of fragments processed each frame. A correlation was observed where the higher the sample resolution became, the lower the performance of the scene was. With the scene being simple on purpose, this highlights how SSAA would be even more expensive computationally when applied to more complex and larger scenes regularly seen within video games.

# Comparison and evaluation

No AA held the highest performance metric, with MSAA placing second and SSAA falling far behind last. MSAA performed the best with visual quality as it made a noticeable difference to reducing aliasing artefacts within the scene. SSAA did reduce aliasing and improve overall smoothness. However, the improvement visually was limited in comparison to the significant cost to the scene’s performance, with some minor aliasing artefacts still being noticeable along curved edges.

With MSAA’s ability to increase visual quality so well while retaining a significant amount of performance within a scene, it’s unsurprising that it is favoured within modern engines over SSAA. Lower computational costs with the ability to be easily implemented as well as being handled internally by hardware make MSAA quite the popular choice.

# Conclusion

This report investigated different AA techniques on the impacts to visual and performance quality in a real-time 3D scene using OpenGL. Although rendering without AA resulted in the highest performance, the visual quality was poor with prominent aliasing artefacts. SSAA improved the scene by increasing the overall image smoothness, but at a significant cost to the performance, becoming unsuitable for a real-time rendered scene. On the other hand, MSAA offered the best balance between both visual and performance quality. MSAA is popular within modern game engines, both OpenGL and DirectX provide built in support. The reductions in aliasing artefacts, paired with the performance quality being maintained placed MSAA as the most appropriate AA technique for real time rendering.

# Appendix

## Asset licenses

### Rubik’s Cube



<https://www.turbosquid.com/3d-models/rubik-cube-with-dbrand-robot-skin-1587120>

License: <https://blog.turbosquid.com/turbosquid-3d-model-license/>

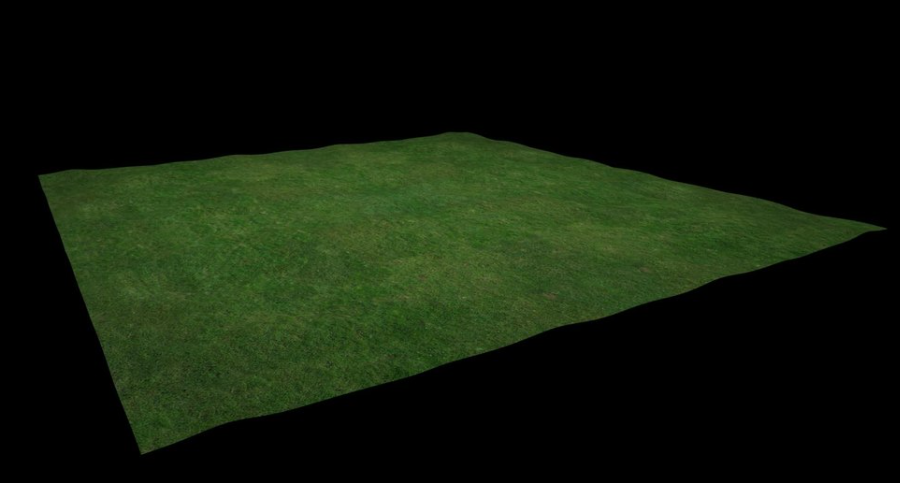
### Moon



<https://www.turbosquid.com/3d-models/moon-3d-model/535681>

License: <https://blog.turbosquid.com/turbosquid-3d-model-license/>

### Grass



<https://www.turbosquid.com/3d-models/3d-grassland-duco-3d-1822648>

License: <https://blog.turbosquid.com/turbosquid-3d-model-license/>

## Code tutorial Sources

FPS counter video: <https://www.youtube.com/watch?v=BA6aR_5C_BM>

Enabling MSAA tutorial video: <https://www.youtube.com/watch?v=oHVh8htoGKw>

### GitHub Repository

<https://github.com/sebha01/Real-Time-Rendering-Assignment-1.git>

## Code

### Scene Renderer Class

#pragma once

#include "Includes.h"

#include <array>

class SceneRenderer

{

private:

//// Shaders ////

GLuint phongShader;

GLuint basicShader;

// Models //

Sphere\* moonModel = nullptr;

Model\* grass = nullptr;

Model\* rubik = nullptr;

// Textures //

GLuint grassTexture;

GLuint rubikTexture;

GLuint moonTexture;

//Light Data///

// Lights

GLfloat light\_ambient[4] = { 1.0, 1.0, 1.0, 1.0 }; // Dim light

GLfloat light\_diffusers[20] = {

1.0, 0.0, 0.0, 1.0, // Red

0.0, 1.0, 0.0, 1.0, // Green

1.0, 1.0, 0.0, 1.0, //Yellow

0.0, 0.0, 1.0, 1.0, //Blue

1.0, 1.0, 1.0, 1.0 //White light above tower

}; // White main light

GLfloat light\_positions[20] = {

27.0, 5.0, 18.0, 1.0, //Red Light

-27.0, 5.0, 18.0, 1.0, //Green

-27.0, 5.0, -18.0, 1.0, //Yellow

27.0, 5.0, -18.0, 1.0, // Blue

0.0, 25.0, 0.0, 1.0 // White light above tower

}; // Point light (w=1.0)

GLfloat attenuation[3] = {1.0, 0.10, 0.08};

// Materials

GLfloat mat\_amb\_diff[4] = {1.0, 1.0, 1.0, 1.0}; // Texture map will provide ambient and diffuse.

GLfloat mat\_specularCol[4] = {1.0, 1.0, 1.0, 1.0}; // White highlight

GLfloat mat\_specularExp = 32.0; // Shiny surface

// Uniform variables //

GLuint textureUniformLoc;

GLint modelMatrixLocation;

GLint viewProjectionMatrixLocation;

GLint invTransposeMatrixLocation;

GLint lightDirectionLocation;

GLint lightDiffuseLocation;

GLint lightSpecularLocation;

GLint lightSpecExpLocation;

GLint cameraPosLocation;

//Uniform Locations - Basic Shader////////////////////////////////////////////

// Get unifom locations in shader

GLuint uLightAmbient;

GLuint uLightDiffusers;

GLuint uLightAttenuation;

GLuint uLightPositions;

GLuint uEyePos;

// Get material unifom locations in shader

GLuint uMatAmbient;

GLuint uMatDiffuse;

GLuint uMatSpecularCol;

GLuint uMatSpecularExp;

// Other variables //

float moonTheta = 0.0f;

float sunTheta = 0.0f;

PrincipleAxes\* principleAxes = nullptr;

//Variables to be passed in from source.cpp

unsigned int w = 1500;

unsigned int h = 1200;

// Camera settings

// width, heigh, near plane, far plane

Camera\_settings camera\_settings{ w, h, 0.1, 1500.0 };

//Timer

Timer timer;

// Instantiate the camera object with basic data

Camera\* camera;

double lastX = camera\_settings.screenWidth / 2.0f;

double lastY = camera\_settings.screenHeight / 2.0f;

public:

SceneRenderer();

void Init();

void Render();

void changeSunThetaValue(double value);

//Getters

Camera\_settings getCameraSettings();

Timer& getTimer();

Camera\* getCamera();

void setLastX(double x);

void setLastY(double y);

double getLastX();

double getLastY();

};

#include "SceneRenderer.h"

SceneRenderer::SceneRenderer()

{

//Render the camera ready for the source file to generate the window

camera = new Camera(camera\_settings, glm::vec3(0.0f, 4.0f, 60.0f));

}

void SceneRenderer::Init()

{

//Set the princple axes

principleAxes = new PrincipleAxes();

//Define which file the models need to use

moonModel = new Sphere(32, 16, 1.0f, glm::vec4(1.0f, 0.0f, 0.0f, 1.0f), CG\_RIGHTHANDED);

grass = new Model("Resources\\Models\\Grass\\grass.obj");

rubik = new Model("Resources\\Models\\Cube\\Rubik Cube.obj");

//Textures

grassTexture = TextureLoader::loadTexture(string("Resources\\Models\\Grass\\grassTexture.jpg"));

grass->attachTexture(grassTexture);

rubikTexture = TextureLoader::loadTexture(string("Resources\\Models\\Cube Textures\\Robot-Skin.jpg"));

rubik->attachTexture(rubikTexture);

moonTexture = TextureLoader::loadTexture(string("Resources\\Models\\Moon\_Textures\\Moon\_Diffuse.jpg"));

//PHONG SHADER

GLSL\_ERROR glsl\_err = ShaderLoader::createShaderProgram(

string("Resources\\Shaders\\Phong-texture.vs"),

string("Resources\\Shaders\\Phong-texture.fs"),

&phongShader);

//BASIC SHADER

glsl\_err = ShaderLoader::createShaderProgram(

string("Resources\\Shaders\\Basic\_shader.vert"),

string("Resources\\Shaders\\Basic\_shader.frag"),

&basicShader);

//Setup uniform locations for shader

textureUniformLoc = glGetUniformLocation(phongShader, "texture");

modelMatrixLocation = glGetUniformLocation(phongShader, "modelMatrix");

viewProjectionMatrixLocation = glGetUniformLocation(phongShader, "viewProjectionMatrix");

invTransposeMatrixLocation = glGetUniformLocation(phongShader, "invTransposeModelMatrix");

lightDirectionLocation = glGetUniformLocation(phongShader, "lightDirection");

lightDiffuseLocation = glGetUniformLocation(phongShader, "lightDiffuseColour");

lightSpecularLocation = glGetUniformLocation(phongShader, "lightSpecularColour");

lightSpecExpLocation = glGetUniformLocation(phongShader, "lightSpecularExponent");

cameraPosLocation = glGetUniformLocation(phongShader, "cameraPos");

//Uniform Locations - Basic Shader////////////////////////////////////////////

// Get unifom locations in shader

uLightAmbient = glGetUniformLocation(basicShader, "lightAmbient");

uLightDiffusers = glGetUniformLocation(basicShader, "lightDiffusers");

uLightAttenuation = glGetUniformLocation(basicShader, "lightAttenuation");

uLightPositions = glGetUniformLocation(basicShader, "lightPositions");

uEyePos = glGetUniformLocation(basicShader, "eyePos");

// Get material unifom locations in shader

uMatAmbient = glGetUniformLocation(basicShader, "matAmbient");

uMatDiffuse = glGetUniformLocation(basicShader, "matDiffuse");

uMatSpecularCol = glGetUniformLocation(basicShader, "matSpecularColour");

uMatSpecularExp = glGetUniformLocation(basicShader, "matSpecularExponent");

}

void SceneRenderer::Render()

{

// Clear the screen

glClearColor(0.1f, 0.2f, 0.8f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

glm::mat4 grassModel = glm::mat4(1.0);

glm::mat4 rubikModel = glm::mat4(1.0);

glm::mat4 moon(1.0f);

glm::mat4 view = camera->getViewMatrix();

glm::mat4 projection = camera->getProjectionMatrix();

glm::mat4 viewProjection = projection \* view;

glm::vec3 eyePos = camera->getCameraPosition();

principleAxes->render(viewProjection);

/////////////////////

// GRASS MODEL

/////////////////////

grassModel = glm::translate(grassModel, glm::vec3(-10.0f, -10.0f, 0.0f));

grassModel = glm::rotate(grassModel, glm::radians(180.0f), glm::vec3(1.0f, 0.0f, 0.0f));

grassModel = glm::scale(grassModel, glm::vec3(0.1f));

glUseProgram(basicShader);

glUniform4fv(uLightDiffusers, 5, (GLfloat\*)&light\_diffusers);

glUniform4fv(uLightAmbient, 1, (GLfloat\*)&light\_ambient);

glUniform4fv(uLightPositions, 5, (GLfloat\*)&light\_positions);

glUniform3fv(uLightAttenuation, 1, (GLfloat\*)&attenuation);

glUniform3fv(uEyePos, 1, (GLfloat\*)&eyePos);

//Pass material data

glUniform4fv(uMatAmbient, 1, (GLfloat\*)&mat\_amb\_diff);

glUniform4fv(uMatDiffuse, 1, (GLfloat\*)&mat\_amb\_diff);

glUniform4fv(uMatSpecularCol, 1, (GLfloat\*)&mat\_specularCol);

glUniform1f(uMatSpecularExp, mat\_specularExp);

glUniformMatrix4fv(glGetUniformLocation(basicShader, "view"), 1, GL\_FALSE, glm::value\_ptr(view));

glUniformMatrix4fv(glGetUniformLocation(basicShader, "projection"), 1, GL\_FALSE, glm::value\_ptr(projection));

//Drawing the objects

glUniformMatrix4fv(glGetUniformLocation(basicShader, "model"), 1, GL\_FALSE, glm::value\_ptr(grassModel));

grass->draw(basicShader); //Draw the plane

//////////////////

// RUBIK CUBE

//////////////////

rubikModel = glm::translate(rubikModel, glm::vec3(-10.0f, 0.0f, 0.0f));

//grassModel = glm::rotate(grassModel, glm::radians(180.0f), glm::vec3(1.0f, 0.0f, 0.0f));

rubikModel = glm::scale(rubikModel, glm::vec3(20.0f));

glUniformMatrix4fv(glGetUniformLocation(basicShader, "model"), 1, GL\_FALSE, glm::value\_ptr(rubikModel));

rubik->draw(basicShader);

glUseProgram(0);

/////////////

// MOON MODEL

/////////////

moon = glm::translate(moon, glm::vec3(20.0f, 0.0f, -15.0f));

moon = glm::scale(moon, glm::vec3(10.0f));

moonTheta += 15.0f \* float(timer.getDeltaTimeSeconds());

if (moonModel)

{

// Modelling transform

moon = glm::rotate(moon, glm::radians(23.44f), glm::vec3(0.0, 0.0, 1.0));//Moon tilt

moon = glm::rotate(moon, glm::radians(moonTheta), glm::vec3(0.0, 1.0, 0.0));//Earth rotation

// Calculate inverse transpose of the modelling transform for correct transformation of normal vectors

glm::mat4 invT = glm::transpose(glm::inverse(moon));;

glUseProgram(phongShader);

//// Get the location of the camera in world coords and set the corresponding uniform in the shader

glm::vec3 cameraPos = camera->getCameraPosition();

glUniform3fv(cameraPosLocation, 1, (GLfloat\*)&cameraPos);

// Set the model, view and projection matrix uniforms (from the camera data obtained above)

glUniformMatrix4fv(modelMatrixLocation, 1, GL\_FALSE, glm::value\_ptr(moon));

glUniformMatrix4fv(invTransposeMatrixLocation, 1, GL\_FALSE, glm::value\_ptr(invT));

glUniformMatrix4fv(viewProjectionMatrixLocation, 1, GL\_FALSE, glm::value\_ptr(viewProjection));

//// Set the light direction uniform vector in world coordinates based on the Sun's position

glUniform4f(lightDirectionLocation, cosf(glm::radians(sunTheta)), 0.0f, sinf(glm::radians(sunTheta)), 0.0f);

glUniform4f(lightDiffuseLocation, 1.0f, 1.0f, 1.0f, 1.0f); // white diffuse light

glUniform4f(lightSpecularLocation, 0.5f, 0.5f, 0.5f, 1.0f); // white specular light

glUniform1f(lightSpecExpLocation, 10.0f); // specular exponent / falloff

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, moonTexture);

moonModel->render();

glUseProgram(0);

}

}

void SceneRenderer::changeSunThetaValue(double value)

{

sunTheta += value;

}

Camera\_settings SceneRenderer::getCameraSettings()

{

return camera\_settings;

}

Timer& SceneRenderer::getTimer()

{

return timer;

}

Camera\* SceneRenderer::getCamera()

{

return camera;

}

void SceneRenderer::setLastX(double x)

{

this->lastX = x;

}

void SceneRenderer::setLastY(double y)

{

this->lastY = y;

}

double SceneRenderer::getLastX()

{

return lastX;

}

double SceneRenderer::getLastY()

{

return lastY;

}

### Scene FBO Class

#pragma once

#include "Includes.h"

class SceneFBO

{

private:

//FBO variables

//Actual FBO

GLuint demoFBO;

//Colour texture to render into

GLuint fboColourTexture;

//Depth texture to render into

GLuint fboDepthTexture;

//Flag to indicate that the FBO is valid

bool fboOkay;

public:

SceneFBO(int windowWidth, int windowHeight, int scale);

void BeginRender(int windowWidth, int windowHeight, int scale);

void EndRender(int windowWidth, int windowHeight);

GLuint getSceneTexture();

};

#include "SceneFBO.h"

SceneFBO::SceneFBO(int windowWidth, int windowHeight, int scale)

{

glGenFramebuffers(1, &demoFBO);

glBindFramebuffer(GL\_FRAMEBUFFER, demoFBO);

// Setup colour buffer texture.

// Note: The texture is stored as linear RGB values (GL\_RGBA8). There is no need to

//pass a pointer to image data - we're going to fill in the image when we render the

//scene at render time!

glGenTextures(1, &fboColourTexture);

glBindTexture(GL\_TEXTURE\_2D, fboColourTexture);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA8, windowWidth \* scale, windowHeight \* scale, 0, GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

// Setup depth texture

glGenTextures(1, &fboDepthTexture);

glBindTexture(GL\_TEXTURE\_2D, fboDepthTexture);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_DEPTH\_COMPONENT24, windowWidth \* scale, windowHeight \* scale, 0, GL\_DEPTH\_COMPONENT, GL\_UNSIGNED\_INT, NULL);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

//

// Attach textures to the FBO

//

// Attach the colour texture object to the framebuffer object's colour attachment point #0

glFramebufferTexture2D(

GL\_FRAMEBUFFER,

GL\_COLOR\_ATTACHMENT0,

GL\_TEXTURE\_2D,

fboColourTexture,

0);

// Attach the depth texture object to the framebuffer object's depth attachment point

glFramebufferTexture2D(

GL\_FRAMEBUFFER,

GL\_DEPTH\_ATTACHMENT,

GL\_TEXTURE\_2D,

fboDepthTexture,

0);

//

// Before proceeding make sure FBO can be used for rendering

//

GLenum demoFBOStatus = glCheckFramebufferStatus(GL\_FRAMEBUFFER);

if (demoFBOStatus != GL\_FRAMEBUFFER\_COMPLETE) {

fboOkay = false;

cout << "Could not successfully create framebuffer object to render texture!" << endl;

}

else {

fboOkay = true;

cout << "FBO successfully created" << endl;

}

// Unbind FBO for now! (Plug main framebuffer back in as rendering destination)

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

}

void SceneFBO::BeginRender(int windowWidth, int windowHeight, int scale)

{

if (!fboOkay)

return; // Don't render anything if the FBO was not created successfully

// Bind framebuffer object so all rendering redirected to attached images (i.e. our texture)

glBindFramebuffer(GL\_FRAMEBUFFER, demoFBO);

// All rendering from this point goes to the bound textures (setup at initialisation time) and NOT the actual screen!!!!!

// Clear the screen (i.e. the texture)

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// Set viewport to specified texture size (see above)

glViewport(0, 0, windowWidth \* scale, windowHeight \* scale);

//Render code to go next

}

void SceneFBO::EndRender(int windowWidth, int windowHeight)

{

// Set OpenGL to render to the MAIN framebuffer (ie. the screen itself!!)

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

glViewport(0, 0, windowWidth, windowHeight);

}

GLuint SceneFBO::getSceneTexture() {

return fboColourTexture;

}

### Basic Shader

#### Fragment Shader

#version 330 core

in vec2 TexCoord;

in vec3 Normal;

in vec3 Vertex;

//Texture sampler

uniform sampler2D texture\_diffuse1;

//Camera location

uniform vec3 eyePos;

//Light information

uniform vec4 lightPositions[5];

uniform vec4 lightAmbient;

uniform vec4 lightDiffusers[5];

uniform vec3 lightAttenuation; // x=constant, y=linear, z=quadratic (x<0 means light is not active)

//Material iformation

uniform vec4 matAmbient;

uniform vec4 matDiffuse;

uniform vec4 matSpecularColour;

uniform float matSpecularExponent;

out vec4 FragColour;

vec4 Lights[5];

//Function prototype

vec4 calculateLight(vec4 lightPosition, vec4 lightDiffuse);

void main()

{

//Calling the light functions

for (int i = 0; i < 5; i++)

{

Lights[i] = calculateLight(lightPositions[i], lightDiffusers[i]);

}

FragColour = Lights[0] + Lights[1] + Lights[2] + Lights[3] + Lights[4];

}

vec4 calculateLight(vec4 lightPosition, vec4 lightDiffuse)

{

//Attenuation/drop-off

float d = length(lightPosition.xyz - Vertex);

float att = 1.0 / (lightAttenuation.x + lightAttenuation.y \* d + lightAttenuation.z \* (d \* d));

//Ambient light value

vec4 texColour = texture(texture\_diffuse1, TexCoord);

vec4 ambient = lightAmbient \* matAmbient \* texColour \* att;

//Diffuse light value

vec3 N = normalize(Normal);

vec3 L = normalize(lightPosition.xyz - Vertex);

float lambertTerm = clamp(dot(N, L), 0.0, 1.0);

vec4 diffuse = lightDiffuse \* matDiffuse \* lambertTerm \* texColour \* att;

//Specular light value

vec3 E = normalize(eyePos - Vertex);

vec3 R = reflect(-L, N); // reflected light vector about normal N

float specularIntensity = pow(max(dot(E, R), 0.0), matSpecularExponent);

vec4 specular = matSpecularColour \* specularIntensity \* texColour \* att;

return ambient + diffuse + specular;

}

#### Vertex Shader

#version 330 core

layout (location = 0) in vec3 vertexPos;

layout (location = 1) in vec3 normal;

layout (location = 2) in vec2 texCoord;

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

out vec2 TexCoord;

out vec3 Normal;

out vec3 Vertex;

void main()

{

TexCoord = texCoord;

Normal = mat3(transpose(inverse(model))) \* normal; // normal vector in eye coordinates

Vertex = vec3(model \* vec4(vertexPos, 1.0)); // vertex in eye coordinates (map to vec3 since gl\_Vertex is a vec4)

gl\_Position = projection \* view \* model \* vec4(vertexPos, 1.0f);

}

### Source File

#include "Includes.h"

#include "SceneFBO.h"

#include "SceneRenderer.h"

// Function prototypes

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height);

void mouse\_callback(GLFWwindow\* window, double xpos, double ypos);

void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset);

void processInput(GLFWwindow \*window);

//Boolean to capture first mouse input to prevent snapback bug

bool firstMouseInput = false;

//Initialise componenets needed to render scene for non AA, MSAA and SSAA scene

//SceneFBO sceneFBO;

SceneRenderer\* sceneRenderer;

TexturedQuad \*superSamplingScene = nullptr;

int currentAntiAliasingFilter = 0;

static const char\* filterStrings[] = {

"No Anti-Aliasing",

"Multi-Sample Anti-Aliasing x4",

"Super-Sample Anti-Aliasing"

};

bool SSAA\_enabled = false;

int SSAA\_scale = 4;

int main()

{

//Initialise the scene Renderer object

sceneRenderer = new SceneRenderer();

// glfw: initialize and configure

glfwInit();

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MAJOR, 3);

glfwWindowHint(GLFW\_CONTEXT\_VERSION\_MINOR, 3);

glfwWindowHint(GLFW\_OPENGL\_FORWARD\_COMPAT, GL\_TRUE);

glfwWindowHint(GLFW\_OPENGL\_PROFILE, GLFW\_OPENGL\_CORE\_PROFILE);

//Got from https://www.youtube.com/watch?v=oHVh8htoGKw

glfwWindowHint(GLFW\_SAMPLES, 4);

// glfw window creation

GLFWwindow\* window = glfwCreateWindow(sceneRenderer->getCameraSettings().screenWidth, sceneRenderer->getCameraSettings().screenHeight, "30077548 Anti-Aliasing assignment", NULL, NULL);

if (window == NULL)

{

std::cout << "Failed to create GLFW window" << std::endl;

glfwTerminate();

return -1;

}

// Set the callback functions

glfwMakeContextCurrent(window);

glfwSetFramebufferSizeCallback(window, framebuffer\_size\_callback);

glfwSetCursorPosCallback(window, mouse\_callback);

glfwSetScrollCallback(window, scroll\_callback);

// tell GLFW to capture our mouse

glfwSetInputMode(window, GLFW\_CURSOR, GLFW\_CURSOR\_DISABLED);

//Set firstMouseInput to true so camera does not snap to mouse position

firstMouseInput = true;

// glad: load all OpenGL function pointers

if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))

{

std::cout << "Failed to initialize GLAD" << std::endl;

return -1;

}

sceneRenderer->Init();

SceneFBO sceneFBO(sceneRenderer->getCameraSettings().screenWidth, sceneRenderer->getCameraSettings().screenHeight, SSAA\_scale);

superSamplingScene = new TexturedQuad(sceneFBO.getSceneTexture(), true);

//Rendering settings

glfwSwapInterval(0); // glfw enable swap interval to match screen v-sync 1 for vsync 0 for off

glEnable(GL\_DEPTH\_TEST);

glEnable(GL\_FRAMEBUFFER\_SRGB);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

int width, height;

glfwGetFramebufferSize(window, &width, &height);

TextRenderer textRenderer(width, height);

currentAntiAliasingFilter = 0;

///////////////////////////////////////////////////////////////////////////

//FPS Counter variables

//Video used to create FPS counter -> https://www.youtube.com/watch?v=BA6aR\_5C\_BM

double prevTime = 0.0;

double crntTime = 0.0;

double timeDiff = 0.0;

unsigned int counter = 0;

///////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////

// render loop

while (!glfwWindowShouldClose(window))

{

//FPS counter

crntTime = glfwGetTime();

timeDiff = crntTime - prevTime;

counter++;

if (timeDiff >= 1.0)

{

//initialised title variable

std::stringstream title;

//Calculate the fps and ms per frame

double FPS = (1.0 / timeDiff) \* counter;

double ms = (timeDiff / counter) \* 1000;

//set the title to show the fps and ms

title << std::fixed << std::setprecision(2) << "30077548 Anti-Aliasing assignment\tFPS:"

<< FPS << "\tMs: " << ms;

//Set the new window title

glfwSetWindowTitle(window, title.str().c\_str());

//set previous time to current time and reset counter

prevTime = crntTime;

counter = 0;

}

//Change aliasing effects, learnt how to do MSAA from https://www.youtube.com/watch?v=oHVh8htoGKw

//Used a switch statement to change between effects.

switch (currentAntiAliasingFilter)

{

case 0:

//No Anti-Aliasing

glDisable(GL\_MULTISAMPLE);

SSAA\_enabled = false;

break;

case 1:

//Multi Sample Anti-Aliasing

glEnable(GL\_MULTISAMPLE);

SSAA\_enabled = false;

break;

case 2:

//Super Sample Anti-Aliasing

glDisable(GL\_MULTISAMPLE);

SSAA\_enabled = true;

//Frame buffer automatically swapped within and reset back to main swap chain

//no need to set frame buffer back to 0

break;

default:

//Default set to no anti-aliasing

glDisable(GL\_MULTISAMPLE);

SSAA\_enabled = false;

break;

}

// input

processInput(window);

sceneRenderer->getTimer().tick();

if (SSAA\_enabled)

{

sceneFBO.BeginRender(sceneRenderer->getCameraSettings().screenWidth, sceneRenderer->getCameraSettings().screenHeight, SSAA\_scale);

sceneRenderer->Render();

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

//Enable blending to render the text otherwise renders as a block form

// disable after text rendered so that it doesn't affect the other models being rendered

glEnable(GL\_BLEND);

textRenderer.renderText(filterStrings[currentAntiAliasingFilter], 15.0f, 15.0f, 1.0f, glm::vec3(1.0, 1.0f, 1.0f));

glDisable(GL\_BLEND);

sceneFBO.EndRender(sceneRenderer->getCameraSettings().screenWidth, sceneRenderer->getCameraSettings().screenHeight);

glDisable(GL\_DEPTH\_TEST);

superSamplingScene->render(glm::mat4(1.0));

glEnable(GL\_DEPTH\_TEST);

}

else

{

sceneRenderer->Render();

//Enable blending to render the text otherwise renders as a block form

// disable after text rendered so that it doesn't affect the other models being rendered

glEnable(GL\_BLEND);

textRenderer.renderText(filterStrings[currentAntiAliasingFilter], 15.0f, 15.0f, 1.0f, glm::vec3(1.0, 1.0f, 1.0f));

glDisable(GL\_BLEND);

}

// glfw: swap buffers and poll events

glfwSwapBuffers(window);

glfwPollEvents();

}

// glfw: terminate, clearing all previously allocated GLFW resources.

glfwTerminate();

return 0;

}

// process all input: query GLFW whether relevant keys are pressed/released this frame and react accordingly

void processInput(GLFWwindow\* window)

{

sceneRenderer->getTimer().updateDeltaTime();

if (glfwGetKey(window, GLFW\_KEY\_ESCAPE) == GLFW\_PRESS)

glfwSetWindowShouldClose(window, true);

if (glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS)

sceneRenderer->getCamera()->processKeyboard(FORWARD, sceneRenderer->getTimer().getDeltaTimeSeconds());

if (glfwGetKey(window, GLFW\_KEY\_S) == GLFW\_PRESS)

sceneRenderer->getCamera()->processKeyboard(BACKWARD, sceneRenderer->getTimer().getDeltaTimeSeconds());

if (glfwGetKey(window, GLFW\_KEY\_A) == GLFW\_PRESS)

sceneRenderer->getCamera()->processKeyboard(LEFT, sceneRenderer->getTimer().getDeltaTimeSeconds());

if (glfwGetKey(window, GLFW\_KEY\_D) == GLFW\_PRESS)

sceneRenderer->getCamera()->processKeyboard(RIGHT, sceneRenderer->getTimer().getDeltaTimeSeconds());

if (glfwGetKey(window, GLFW\_KEY\_LEFT\_SHIFT) == GLFW\_PRESS)

sceneRenderer->getCamera()->setRunSpeed(3.0);

else if (glfwGetKey(window, GLFW\_KEY\_LEFT\_SHIFT) == GLFW\_RELEASE)

sceneRenderer->getCamera()->setRunSpeed(1.0);

//Rotate light direction on the moon

if (glfwGetKey(window, GLFW\_KEY\_LEFT) == GLFW\_PRESS)

sceneRenderer->changeSunThetaValue(-0.1f);

if (glfwGetKey(window, GLFW\_KEY\_RIGHT) == GLFW\_PRESS)

sceneRenderer->changeSunThetaValue(0.1f);

//Switch the texture filter modes

if (glfwGetKey(window, GLFW\_KEY\_1) == GLFW\_PRESS)

currentAntiAliasingFilter = 0;

if (glfwGetKey(window, GLFW\_KEY\_2) == GLFW\_PRESS)

currentAntiAliasingFilter = 1;

if (glfwGetKey(window, GLFW\_KEY\_3) == GLFW\_PRESS)

currentAntiAliasingFilter = 2;

}

// glfw: whenever the window size changed (by OS or user resize) this callback function executes

void framebuffer\_size\_callback(GLFWwindow\* window, int width, int height)

{

// make sure the viewport matches the new window dimensions; note that width and

glViewport(0, 0, width, height);

sceneRenderer->getCamera()->updateScreenSize(width, height);

}

// glfw: whenever the mouse moves, this callback is called

void mouse\_callback(GLFWwindow\* window, double xpos, double ypos)

{

//Ensure that camera does not snap due to the initial large offset of where lastX and

// lastY have not yet matched the mouse position the first time the mousecallback

// function is called

if (firstMouseInput)

{

sceneRenderer->setLastX(xpos);

sceneRenderer->setLastY(ypos);

firstMouseInput = false;

}

double xoffset = xpos - sceneRenderer->getLastX();

double yoffset = sceneRenderer->getLastY() - ypos; // reversed since y-coordinates go from bottom to top

sceneRenderer->setLastX(xpos);

sceneRenderer->setLastY(ypos);

sceneRenderer->getCamera()->processMouseMovement(xoffset, yoffset);

}

// glfw: whenever the mouse scroll wheel scrolls, this callback is called

void scroll\_callback(GLFWwindow\* window, double xoffset, double yoffset)

{

sceneRenderer->getCamera()->processMouseScroll(yoffset);

}