Name: Ralenski Doucet
Contents: Computer Graphics Project Documentation

Project Umls:

- **Application class:**

Public:
Application()
Virtual ~Application()
Void clearScreen()
Void run(const char * title, unsigned int width , unsigned int height, bool fullscreen)
Protected:
Virtual void startup()
Virtual void shutdown()
Virtual void update()
Virtual void draw()

- Camera class:

Public:
Camera()
~Camera()
Void update(float Deltatime)
Glm::vec4 set Perspective(float field of view, float aspect ratio, float near, float far)
Void set Look At(glm::vec3 from, glm::vec3 to, glm::vec3 up)
Void set Position(glm::vec3 position)
Glm::mat4 get World Transform()
Glm:: mat4 get View()
Glm::mat4 get Projection()
Glm::mat4 get Projection View()
Void change Projection(int is Active
Private:
Glm::mat4 world Transform
Glm:: mat4 view Transform
Glm::mat4 projection Transform
Glm::mat4 projection View Transform
Void update Projection View Transform()

- **Fly Camera class:**

Public:
Fly Camera()
~Fly Camera()
Void update(float delta Time)
Void set Speed(float value)
Private:
Float speed
Glm::vec3 up

- **Transform class:**

Public:
Transform()
~Transform()
Glm::mat4 Translate(glm::mat model, glm::vec3 move Amount)
Glm::mat4 Rotate(float radians, flm::glm::vec3 axis)
Glm::mat4 Scale(float size)
Private:
Glm::mat4 m_model
Glm::mat4 m_world Position
Glm::mat4 m_local Position
Glm::mat4 m_world Rotation
Glm::mat4 m_local Rotation
Glm::mat4 m_world Scale
Glm::mat4 m_local Scale

- **Mesh Renderer Class:**

Public:
Mesh Renderer()
Virtual ~Mesh Renderer()
vertex{glm::vec4 position, glm::vec4 color, glm::vec4 normal, glm::vec2 text Coord,}
void initialize(std::vector<unsigned int>& m_indices, std::vector<Vertex>& m_vertices);
void render();
std::vector<unsigned int> m_indices;
std::vector<Vertex> m_vertices;
unsigned int vao;
unsigned int vbo;

```
unsigned int ibo;
void create_Buffers();
```

- **Rendering Geometry App:**

Public:

```
void startup() override;
void shutdown() override;
void update(float dt) override;
void draw() override;
std::vector<glm::vec4>points;
std::vector <glm::vec4> genHalfCircle(int np, double radius);
std::vector<glm::vec4> genSphere(std::vector<glm::vec4>points, unsigned int numofM);
std::vector<unsigned int>genSphereIndices(int np, int num of M);
std::vector<Vertex> genPlane(int size);
std::vector<Vertex>genCube(std::vector<Vertex> vertices);
std::vector<glm::vec4> rotate Half Circle(std::vector<glm::vec4>points, unsigned int nm);
std::vector<unsigned int> get Cube Indices();
Shader *mShader;
MeshRenderer* m Mesh;
Camera *camera;
Transform *m Transform= new Transform();
glm::mat4 model, view, projection;
```

- **Shader Class:**

Public:

```
Shader();
~Shader();
enum SHADER_TYPE { VERTEX = 0, FRAGMENT = 1};
enum Light_Type {phong = 0};
void choose Lighting(Shader::Light_Type);
std::string fs SourceString;
std::string vs SourceString;
std::string data;
void bind();
void unbind();
```

```
bool load(const char* filename, Shader::SHADER_TYPE);
bool attach();
void Load();
unsigned int getUniform(const char* mvp);
unsigned int m_vertexShader;
unsigned int m_fragmentShader;
const char* vs Source;
const char* fsSource;
const char* phongL;
unsigned int m_program;
struct Shader Data {Shader* shader; char* source; unsigned type; bool Correct File;};
```

- **Intro Application class:**

Public:

```
Intro Application();
~Intro Application();
glm::mat4 view = glm::lookAt(glm::vec3(10, 10, 10), glm::vec3(0), glm::vec3(0, 10, 0));
glm::mat4 projection = glm::perspective(glm::pi<float>() * 0.30f, 20 / 3.f, 0.15f, 1000.f);
glm::mat4 model = glm::mat4(1);
Fly Camera *mCamera = new Fly Camera();
```

Protected:

```
void startup() override;
void shutdown() override;
void update(float dt) override;
void draw() override;
```

3. Code Architecture

Application class:
Prototype: Application()
Description: A constructor for the camera class
Precondition: none
Post condition: none
Protection Level: public

Prototype: ~Application()
Description: A de-constructor that deletes new instances of the application class after they are used during runtime
Precondition: none
Post condition: none
Protection Level: public

Prototype: clear Screen()
Description: A function that will clear the screen whenever called
Precondition: none
Post condition: none
Protection Level: public

Prototype: run()
Description: When called takes in an argument for title width, and height
Precondition: none
Post condition: none
Protection Level: public

Prototype: startup()
Description: A function that declares what other functions will happen at the start of the program
Precondition: none
Post condition: none
Protection Level: public

Prototype: ShutDown()
Description: A function that stops the application from running
Precondition: none
Post condition: none
Protection Level: public

Prototype: update()
Description: A function that changes over while the program is running
Precondition: none
Post condition: none

Protection Level: public

Prototype: draw()
Description: A function that draws to the window while the program is running
Precondition: none
Post condition: none
Protection Level: public

- **Camera class:**

Prototype: Camera()
Description: a Constructor for the camera class
Precondition:none
Post condition: none
Protection Level: public

Prototype: ~ Camera()
Description: A de-constructor for the camera class that deletes any new instances of camera created at runtime
Precondition: none
Post condition: none
Protection Level: public

Prototype: update()
Description: A function that updates the state of the program during runtime
Precondition: none
Post condition: none
Protection Level: public

Prototype: set Perspective()
Description: A function that allows the perspective to be set
Precondition: none
Post condition: none
Protection Level: public

Prototype: set Look At()

Description: A function that allows you to set what the camera is looking at
Precondition: none
Post condition: none
Protection Level: public

Prototype: set Position()
Description: A function that allows you to set the position of the camera
Precondition: none
Post condition: none
Protection Level: public

Prototype: get World Transform()
Description: A function that gets the value of the world transform and then returns it
Precondition: none
Post condition: none
Protection Level: public

Prototype: get View()
Description: a function that gets the value of view and then returns it
Precondition: none
Post condition: none
Protection Level: public

Prototype: get projection()
Description: A function that gets the value of projection and then returns it
Precondition: none
Post condition: none
Protection Level: public

Prototype: get Projection View()
Description: A function that gets the value of the projection view and returns it
Precondition: none
Post condition: none

Protection Level: public

Prototype: change Projection()
Description: A function that changes the projection the types it can be switched with is perspective and orthographic
Precondition: none
Post condition: none
Protection Level: public

Prototype: update projection View transform()
Description: A function that change the view transform to correspond with the projection type.
Precondition: none
Post condition: none
Protection Level: private

- **Fly Camera class:**

Prototype: Fly Camera()
Description: A constructor for the camera class
Precondition: none
Post condition: none
Protection Level: public

Prototype: ~Fly Camera()
Description: A de-constructor to delete any new instances of Fly Camera after they are used in runtime
Precondition: none
Post condition: none
Protection Level: public

Prototype: Update()
Description: A function that updates the delta time while the program is running
Precondition: none
Post condition: none
Protection Level: public

Prototype: set Speed()
Description: A function that sets the speed of the fly camera
Precondition: none
Post condition: none
Protection Level: public

- **Transform Class:**

Prototype: Transform()
Description: A constructor for the Transform class
Precondition: none
Post condition: none
Protection Level: public

Prototype: ~Transform()
Description: A de-constructor for the transform that deletes any new instances of transform after they are used in runtime
Precondition: none
Post condition: none
Protection Level: public

Prototype: Translate()
Description: A function that moves the transform by changing the position of the transform
Precondition: none
Post condition: none
Protection Level: public

Prototype: Rotate()
Description: A function that rotates the transform based on what axises value is changed
Precondition: none
Post condition: none
Protection Level: public

Prototype: Scale()

Description: A function that scale the transform by a give amount
Precondition: none
Post condition: none
Protection Level: public

- **Mesh Render class:**

Prototype: Mesh Renderer()
Description: A constructor for the mesh renderer class
Precondition: none
Post condition: none
Protection Level: public

Prototype: ~mesh Renderer()
Description: A de-constructor that deletes any new instances of the mesh renders after they are used in runtime
Precondition: none
Post condition: none
Protection Level: public

Prototype: initialize()
Description: A function that will create certain variables at the start of runtime.
Precondition: none
Post condition: none
Protection Level: public

Prototype: render()
Description: A function that binds the vertex array and draws the elements of the shape to be drawn
Precondition: none
Post condition: none
Protection Level: public

Prototype: Create Buffers()
Description: A function that generates the vertex array and buffers ,binds the buffers, and also binds the vertex array

Precondition: none
Post condition: none
Protection Level: public

- **Shader class:**

Prototype: Shader()
Description: A constructor for the shader class.
Precondition: none
Post condition: none
Protection Level: public

Prototype: ~Shader()
Description: A de-constructor that deletes any instances of shader created during runtime.
Precondition: none
Post condition: none
Protection Level: public

Prototype: Choose LIghting()
Description:
Precondition:
Post condition:
Protection Level:

Prototype: Bind()
Description: A function that binds gl use program to m_program
Precondition: none
Post condition: none
Protection Level: public

Prototype: unbind()
Description: A function that sets the program to be used to null or 0
Precondition: none
Post condition: none
Protection Level: public

Prototype: load()
Description: A function that allows the program to load the information for each shader type to be load for use elsewhere in the project
Precondition: none
Post condition: none
Protection Level: public

Prototype: attach()
Description: A function that creates the shaders, sources the shaders, complies the shaders, attaches the shaders to the program,and links the program
Precondition: none
Post condition: none
Protection Level: public

Prototype: load()
Description: A function that assigns vsSource and fsSource are equal to their counterparts as defined in the file name shader.cpp
Precondition: none
Post condition: none
Protection Level: public

Prototype: get Uniform()
Description: A function that returns the uniform location of m_program
Precondition: none
Post condition: none
Protection Level: public

- **GUI Application Class:**

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Prototype:
Description:
Precondition:
Post condition:
Protection Level:

Source Code

Application.cpp

```
#include "Application.h"
#include <gl_core_4_4.h>
#include <GLM\glm.hpp>
#include <GLM\ext.hpp>
#include <GLM\fwd.hpp>
#include <GLFW\glfw3.h>
#include <Gizmos.h>
#include "stdio.h"
#include "imgui.h"
#include "imgui_impl_glfw_gl3.h"
Application::Application() :m_window(nullptr), m_gameover(false),
m_clearColor{ 5,5,5,5 }, m_runningTime(0){}
Application::~Application()
{
}
```

```
void Application::run(const char * title, unsigned int width, unsigned int
height, bool fullscreen)
{
    glfwInit();
    float prevTime = glfwGetTime();

    m_window = glfwCreateWindow(720, 720, "ChalkZone", NULL,
NULL);
    glfwMakeContextCurrent(m_window);
    ogl_LoadFunctions();
    auto minor = ogl_GetMinorVersion();
    auto major = ogl_GetMajorVersion();
    double PreviousTime = glfwGetTime();

    printf("GL: %i.%i\n", major, minor);

    glClearColor(0.15f, 0.15f, 0.15f, 1);
    glEnable(GL_DEPTH_TEST);
    ImGui_ImplGlfwGL3_Init(m_window,true);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    startup();
    while (glfwWindowShouldClose(m_window) == false &&
glfwGetKey(m_window, GLFW_KEY_ESCAPE) != GLFW_PRESS)
    {
        float currentTime = glfwGetTime();
        float deltaTime = currentTime - prevTime;
        prevTime = currentTime;
        update(deltaTime);
        double CurrentTime = glfwGetTime();
        draw();
        glfwSwapBuffers(m_window);

    }

    shutdown();
}
```

```
void Application::clearScreen()
{
}
```

- **Camera.cpp:**

```
#include "Camera.h"
#include <GLM\glm.hpp>
#include <GLM\fwd.hpp>
#include <GLM\ext.hpp>
#include <iostream>
Camera::Camera() : projectionTransform(glm::mat4(1))
{
        worldTransform = glm::mat4(1);
}

Camera::~Camera()
{
}
void Camera::update(float Deltatime)
{
}
glm::mat4 Camera::setPerspective(float fieldofview, float aspectRatio,
float near, float far)
{
        projectionTransform[0].x = 1 / aspectRatio * tan(fieldofview / 2);
        projectionTransform[1].y = 1 / tan(fieldofview / 2);
        projectionTransform[2].z = 1 / -((far + near) / (far - near));
        projectionTransform[2].w = -1;
        projectionTransform[3].z = ((2 * far *near)/(far - near));
        return projectionTransform;
}
void Camera::setLookAt(glm::vec3 from, glm::vec3 to, glm::vec3 up)
{
        viewTransform = glm::lookAt(from, to, up);
}


void Camera::setPosition(glm::vec3 position)
```

```
{
	worldTransform[2].x += position[0];
	worldTransform[2].y += position[1];
	worldTransform[2].z += position[2];
	std::cout << worldTransform[2].x << " , " << worldTransform[2].y << " , " << worldTransform[2].z << std::endl;

}
glm::mat4 Camera::getWorldTransform()
{
	return worldTransform;
}
glm::mat4 Camera::getView()
{
	return viewTransform;
}

glm::mat4 Camera::getProjection()
{
	return projectionTransform;
}
glm::mat4 Camera::getProjectionView()
{
	return projectionViewTransform;
}

void Camera::ChangeProjection(int isActive)
{
}



void Camera::updateProjectionViewTransform()
{
}
```

- **FlyCamera.cpp:**

```
#include "FlyCamera.h"
#include <GLFW\glfw3.h>
FlyCamera::FlyCamera()

{

    speed = 1;

}



FlyCamera::~FlyCamera()

{

}


void FlyCamera::update(float deltaTime)
{
    auto window = glfwGetCurrentContext();
    if (glfwGetKey(window, GLFW_KEY_W))

    {
        glm::vec3 prespective = glm::vec3(0, -speed * deltaTime, 0);

    }

    if (glfwGetKey(window, GLFW_KEY_A))

    {
        glm::vec3 prespective = glm::vec3(speed*deltaTime, 0, 0);

    }
```

```
        if (glfwGetKey(window, GLFW_KEY_S))
        {
                glm::vec3 prespective = glm::vec3(0, speed * deltaTime, 0);

        }

        if (glfwGetKey(window, GLFW_KEY_D))

        {
                glm::vec3 prespective= glm::vec3(-speed * deltaTime, 0, 0);

        }
}


void FlyCamera::setSpeed(float value)

{

        speed = value;

}
```

- **Transform.cpp:**

```
#include "Transform.h"

Transform::Transform()
//constructor for any instance of a transform object.
//assigns m_model to be a 4x4 matrix
{
        m_model = glm::mat4(1);
        m_worldPosition = m_model[2];
        m_localPosition = m_model[3];
        for (int col = 0; col < 3; col++)
        {
                m_worldRotation[col].x = m_model[col].x;
                m_worldRotation[col].y = m_model[col].y;
                m_worldRotation[col].z = m_model[col].z;
```

```
            m_localRotation[col].x = m_model[col].x;
            m_localRotation[col].y = m_model[col].y;
            m_localRotation[col].z = m_model[col].z;
    }
    m_worldScale = glm::vec3(m_model[0].x, m_model[1].y,
m_model[2].z);
    m_localScale = glm::vec3(m_model[0].x, m_model[1].y,
m_model[2].z);
}


Transform::~Transform()
{
    // this will be used to deallocate memeroy that the instance of
transform allocates.
    //as of now no memory has been allocated.
}

glm::mat4 Transform::Scale(float size)
//scales the matrix by number value given.
//the bottom right number of the matrix.
{
    glm::mat4 m_scale = glm::mat4(1);
    m_scale[1].x = size;
    m_scale[2].y = size;
    m_scale[3].z = size;
    m_model *= m_scale;
    return m_model;
}

//Create two new auto assigned  varaibles cosine and sin.
//cosine is equal to the number given.
//sine equals opposite of the number given.
//Radians is the measurement of the radius.
glm::mat4 Transform::Rotate(float radians, glm::vec3 axis)
```

```
{
        auto cosine = cos(radians);
        auto sine = sin(radians);
        //if x aixs then rotate
        if (axis == glm::vec3(1, 0, 0))
        {
                //x doesn't change
                //this rotates the x aixs.
                //this rotates the transform on the x aixs.
                m_model[0].y = cosine;
                m_model[1].z = sine;
                m_model[2].y = -sin(radians);
                m_model[3].z = cosine;
        }
        if (axis == glm::vec3(0, 1, 0))
        {

                //if y aixs then rotate
                /*this rotates the transform on the y aixs.*/
                m_model[0].x = cosine;
                m_model[0].z = -sin(radians);
                m_model[2].x = sine;
                m_model[2].z = cosine;
        }
        //if z aixs then rotate
        //the transform will rotate on it's z aixs
        if (axis == glm::vec3(0, 0, 1))
        {
                m_model[0].x = cosine;
                m_model[1].y = -sin(radians);
                m_model[2].x = sine;
                m_model[3].y = cosine;
        }
        return m_model;
}
glm::mat4 Transform::Translate(glm::mat4 model,glm::vec3
moveAmount)
```

```
{
	//^
	//^assign the value of model to equal m_model's value.
	//^model[0] is equal to m_model[0].
	//^model[1] is equal to m_model[1].
	//^model[2] is equal to m_model[2].

	//model[0], model[1], model[2] += 5;
	//^
	//^model[0] which is the transforms x position add 5 to the x
position.
	//^model[1] which is the transforms y position add 5 to the y
position.
	//^model[2] which is the transforms z position add 5 to the z
position.
	return model;
	//^
	//^return model's new values.
}
```

- **MeshRenderer.cpp**

```
#include "MeshRenderer.h"
#include "gl_core_4_4.h"
#include "Shader.h"

MeshRenderer::MeshRenderer()  {}

MeshRenderer::~MeshRenderer()
{
	glDeleteVertexArrays(1, &vao);
	glDeleteBuffers(1, &vbo);
	glDeleteBuffers(1, &ibo);
}
void MeshRenderer::initialize(std::vector<unsigned int>& indices,
std::vector<Vertex>& vertices)
{
	m_indices = indices;
	m_vertices = vertices;
	create_Buffers();
}

void MeshRenderer::render()
{
	glBindVertexArray(vao);
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
	glPrimitiveRestartIndex(0xFFFF);

	glEnable(GL_PRIMITIVE_RESTART);
	glDrawElements(GL_TRIANGLE_STRIP, m_indices.size(),
GL_UNSIGNED_INT, 0);
	glDisable(GL_PRIMITIVE_RESTART);

	glBindVertexArray(0);
}

void MeshRenderer::create_Buffers()
{
	glGenVertexArrays(1, &vao);
	glGenBuffers(1, &vbo);
	glGenBuffers(1, &ibo);

	glBindVertexArray(vao);
	glBindBuffer(GL_ARRAY_BUFFER, vbo);
	glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(Vertex),
m_vertices.data(), GL_STATIC_DRAW);

	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, m_indices.size() *
sizeof(unsigned int), m_indices.data(), GL_STATIC_DRAW);

	glEnableVertexAttribArray(0);
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)0);
```

```
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex),
(void*)sizeof(glm::vec4));

    glBindVertexArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
}
```

- **RenderingGeometry.cpp**

```
#define GLM_FORCE_SWIZZLE
#include "RenderingGeometryApp.h"


void RenderingGeometryApp::startup()
{
    int nm = 30;
    int np = 30;

    mMesh = new MeshRenderer();

    points=     genHalfCircle(np, 5);

    points = genSphere(points, nm);
    camera = new Camera();

    std::vector<unsigned int> indices = genSphereIndices(np, nm);

    std::vector<MeshRenderer::Vertex> vertexs;
    for (glm::vec4 point : points)
    {
        MeshRenderer::Vertex vertex = { point, glm::vec4(150, 25, 50, 0)
};
        vertexs.push_back(vertex);
    }
    mMesh->initialize(indices, vertexs);
```

```
        mShader = new Shader();

        mShader->load("vertex.vert", Shader::SHADER_TYPE::VERTEX);
        mShader->load("fragment.frag",
Shader::SHADER_TYPE::FRAGMENT);

        mShader->attach();

}

//it happens inside of the rendering geometryapp  startup function every
time after attach is Hit
void RenderingGeometryApp::shutdown()
{
}

void RenderingGeometryApp::update(float dt)
{
        model = glm::mat4(1);
        glm::vec3 eye = glm::vec3(10, -10, -10);
        view = glm::lookAt(eye, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));

        projection=camera->setPerspective(glm::pi<float>(), 800 / (float)600,
.1f, 1000.f);
        view = camera->getView();



}

void RenderingGeometryApp::draw()
{
        glUseProgram(mShader->m_program);
        mShader->bind();
        glm::mat4 mvp = projection * view * model;
        mMesh->render();
        glUniformMatrix4fv(mShader->getUniform("ProjectionViewWorld"),
1, GL_FALSE, &mvp[0][0]);
        mShader->unbind();
        glUseProgram(0);

}

std::vector<glm::vec4> RenderingGeometryApp::genHalfCircle(int np,
double radius)
{
        //1st two arguments int np(Number of Points); double radius;
        //2nd declare number of points;
        //3rd declare local varaible that will represent an vertex's position.
        std::vector<glm::vec4>CircleVerts;

        for (float i=0;i<np;i++)
        {
                //calculate (angle or theta) in for loop.
                //angle is equals the answer of (3.14/number of points)
                float angle = glm::pi<float>() / ((float) np - 1);
                float theta = i * angle;

                //push back each vertice in the vertex _points->
                //that shows each generated portion of the half circle
                CircleVerts.push_back(glm::vec4(glm::cos(theta)*radius,
glm::sin(theta)*radius, 0, 1));

        }
        return CircleVerts;

}

std::vector<glm::vec4>
RenderingGeometryApp::genSphere(std::vector<glm::vec4>points,
unsigned int numofM)
{
    std::vector<glm::vec4> SpherePoints;
    for (int i = 0; i < numofM + 1; i++)
    {
        float sphereSlice = (glm::pi<float>() * 2) / (float)numofM;
        float theta = i * sphereSlice;
        for (int j = 0; j < points.size(); j++)
        {
            float X = points[j].x;
            float Y = points[j].y * cos(theta) + points[j].z * -sin(theta);
            float Z = points[j].z * cos(theta) + points[j].y * sin(theta);
            glm::vec4 point = glm::vec4(X, Y, Z, 1);
            SpherePoints.push_back(point);
        }
    }
    return SpherePoints;

}

std::vector<unsigned int> RenderingGeometryApp::genSphereIndices(int
np, int numofM)
{
    std::vector<unsigned int> Sphereindices;
    unsigned int start;
    unsigned int bottom_left;
    unsigned int bottom_right;
    for (int r = 0; r < numofM; r++)
    {
        start = r * np;
        for (int p = 0; p < np; p++)
        {
            bottom_left = start + p;
            bottom_right = bottom_left + np;
            Sphereindices.push_back(bottom_left);
            Sphereindices.push_back(bottom_right);
        }
        Sphereindices.push_back(0xFFFF);
    }
    return Sphereindices;

}

std::vector<Vertex> RenderingGeometryApp::genPlane(int size)
{
    Vertex A = Vertex(glm::vec4(-size, size, 0, 1), glm::vec4(1, 0, 0, 1));
    Vertex B = Vertex(glm::vec4(size, size, 0, 1), glm::vec4(1, 0, 0, 1));
    Vertex C = Vertex(glm::vec4(size, -size, 0, 1), glm::vec4(1, 0, 0, 1));
    Vertex D = Vertex(glm::vec4(-size, -size, 0, 1), glm::vec4(1, 0, 0, 1));
    std::vector<Vertex> PlaneVertices = { A,B,C,D };
    return PlaneVertices;
}

std::vector<Vertex>
RenderingGeometryApp::genCube(std::vector<Vertex> vertices)
{
    std::vector<Vertex> CubePoints;

    CubePoints.push_back(Vertex(glm::vec4(0, 1, 1, 1),glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(1, 1, 1, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(1, 0, 1, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(0, 0, 1, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(0, 0, 0, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(1, 0, 0, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(1, 1, 0, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(0, 1, 0, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(0, 1, 1, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(1, 1, 1, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(1, 1, 0, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(1, 0, 0, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(0, 1, 0, 1), glm::vec4(1)));
    CubePoints.push_back(Vertex(glm::vec4(0, 0, 0, 1), glm::vec4(1)));
    return CubePoints;
}
```

```
std::vector<glm::vec4>
RenderingGeometryApp::rotateHalfCircle(std::vector<glm::vec4> points,
unsigned int nm)
{
    std::vector<glm::vec4> allPoints;
    for (int i = 0; i <= nm; i++)
    {
        float slice = 2.0f * glm::pi<float>() / (float)nm;
        float theta = i * slice;
        for (int j = 0; j < points.size(); j++)
        {
            float newX = points[j].x;
            float newY = points[j].y * cos(theta) + points[j].z *
-sin(theta);
            float newZ = points[j].z * cos(theta) + points[j].y *
sin(theta);
            allPoints.push_back(glm::vec4(newX, newY, newZ, 1));
            //allPoints[i] = glm::round(allPoints[i]);
        }
    }
    return allPoints;
}

std::vector<unsigned int> RenderingGeometryApp::getCubeIndices()
{
    std::vector<unsigned int> indices =
    { 0, 1, 2, 2, 3, 0,//front
        3, 2, 4, 4, 5, 2,//Bot
        4, 5, 6, 6, 7, 4,//Back
        6, 7, 8, 8, 9, 6,//Top
        2, 1, 10, 10, 11, 2,//Right
        0, 3, 12, 12, 13, 0//Left
    };
    return indices;
}
```

- **Shader.cpp**

```
#define GLM_FORCE_SWIZZLE
#define _CRT_SECURE_NO_WARNINGS 1
#include "Shader.h"
#include <GLCORE/gl_core_4_4.h>
#include <fstream>
Shader::Shader()
{
        m_program = glCreateProgram();
}

Shader::~Shader()
{
}
void Shader::bind()
{
        glUseProgram(m_program);

}

void Shader::unbind()
{
        glUseProgram(0);
}

bool Shader::load(const char *Filename, Shader::SHADER_TYPE shadertype)
{
        errno_t err;
        FILE *file;
        err = fopen_s(&file, Filename, "r");
        char mstring[500];
        while(std::fgets(mstring, sizeof mstring, file))
        {
                if (shadertype == Shader::SHADER_TYPE::VERTEX)
                {
```

```
            vsSourceString.append(mstring);
        }
        else if (shadertype == Shader::SHADER_TYPE::FRAGMENT)
        {
            fsSourceString.append(mstring);
        }

    }
    vsSource = vsSourceString.c_str();
    fsSource = fsSourceString.c_str();
    return true;
}

bool Shader::attach()
{
    m_vertexShader = glCreateShader(GL_VERTEX_SHADER);
    m_fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(m_vertexShader, 1, (const char**)&vsSource, 0);
    glCompileShader(m_vertexShader);
    glShaderSource(m_fragmentShader, 1, (const char**)&fsSource, 0);
    glCompileShader(m_fragmentShader);
    glAttachShader(m_program, m_vertexShader);
    glAttachShader(m_program, m_fragmentShader);
    glLinkProgram(m_program);

    int success = GL_FALSE;
    // check that it compiled and linked correctly
    glGetProgramiv(m_program, GL_LINK_STATUS, &success);
    if (success == GL_FALSE) {
        int infoLogLength = 0;
        glGetProgramiv(m_program, GL_INFO_LOG_LENGTH,
&infoLogLength);
        char* infoLog = new char[infoLogLength + 1];
        glGetProgramInfoLog(m_program, infoLogLength, 0, infoLog);
        printf("Error: Failed to link shader program!\n");
        printf("%s\n", infoLog);
        delete[] infoLog;
```

```
	}

	return true;

}

void Shader::Load()
{
	vsSource = "#version 410\n \
	    layout(location = 0) in vec4 Position; \
	    layout(location = 1) in vec4 Color; \
	    out vec4 vColor; \
	    uniform mat4 ProjectionViewWorld; \
	    void main() { vColor = Color; \
	    gl_Position = ProjectionViewWorld * Position; }";

	fsSource = "#version 410\n \
	    in vec4 vColor; \
	    out vec4 FragColor; \
	    void main() { FragColor = vColor; }";
}

unsigned int Shader::getUniform(const char *mvp)
{
	return glGetUniformLocation(m_program, mvp);;
}
```

- **IntroApplication.cpp**

```
#include "IntroApplication.h"
#include "gl_core_4_4.h"
#include "GLFW/glfw3.h"
#include "Camera.h"
#include "FlyCamera.h"
#include "Gizmos.h"
```

```
#include <GLM/glm.hpp>
#include <GLM/ext.hpp>


IntroApplication::IntroApplication()
{
}

IntroApplication::~IntroApplication()
{
}


void IntroApplication::startup()
{
        Gizmos::create();
        mCamera = new FlyCamera();
        mCamera->setLookAt(glm::vec3(10, 10, 10), glm::vec3(0), glm::vec3(0,
1, 0));
        mCamera->setPerspective(glm::pi<float>()*0.25f, 16 / 9.0f, 0.1f,
1000.0f);
        mCamera->setSpeed(10);

}


void IntroApplication::update(float dt)
{
        mCamera->update(dt);
}

void IntroApplication::shutdown()
{
        Gizmos::destroy();
        glfwDestroyWindow(m_window);
        glfwTerminate();
}
```

```
void IntroApplication::draw()
{
        Gizmos::clear();
        Gizmos::addTransform(glm::mat4(1));
        glm::vec4 white(1);
        glm::vec4 black(0, 0, 0, 1);
        for (int i = 0; i < 21; ++i) {
                Gizmos::addLine(glm::vec3(-10 + i, 0, 10),
                        glm::vec3(-10 + i, 0, -10),
                        i == 10 ? white : black);
                Gizmos::addLine(glm::vec3(10, 0, -10 + i),
                        glm::vec3(-10, 0, -10 + i),
                        i == 10 ? white : black);
        }
        Gizmos::addSphere(glm::vec3(0, 0, 0), 5, 15, 10, glm::vec4(1, 1, 1, 1),
&model);
        Gizmos::draw(mCamera->getProjection() * mCamera->getView() *
mCamera->getWorldTransform());

}
```

- **GUIApplication.cpp**

```
#include "GUIApplication.h"

#include "imgui.h"

GUIApplication::GUIApplication()
{
}

GUIApplication::~GUIApplication()
{
}

void GUIApplication::startup()
```

```
{

}

void GUIApplication::shutdown()
{
}

void GUIApplication::update(float dt)
{//use the model matrix to  move the square around

}

void GUIApplication::draw()
{
        if(ImGui::Button("Move Left")){}
        if (ImGui::Button("Move Right")) {}
        if (ImGui::Button("Move Up")) {}
        if (ImGui::Button("Move Down")) {}
        if (ImGui::Button("Move Left")) {}


}
```

Read Me