MODERN OPENGL

$WHOAMI

Konstantinos Benos

- Gameplay and Graphics programmer @ Supercourse ELT (2018-2019)
- Security Vulnerabilities Researcher @ Census Labs (2017-2018)
- System Administrator @ IT AUTh (2015-2016)
- CS Undergraduate @ AUTh (2013-2021[?])

DISCLAIMER

This talk ***is not*** a:

- Good graphics coding practices guide
- Tutorial on the mathematics used behind the scenes
- An introduction for someone new to graphics programming

DISCLAIMER

This talk **is** a:

- Transitioning guide from obsolete to modern OpenGL API
- Shaders coding introduction

01

OLD VS MODERN API

OLD VS MODERN API

OLD OPENGL

- Easier to use
- Fixed rendering pipeline
- Immediate mode

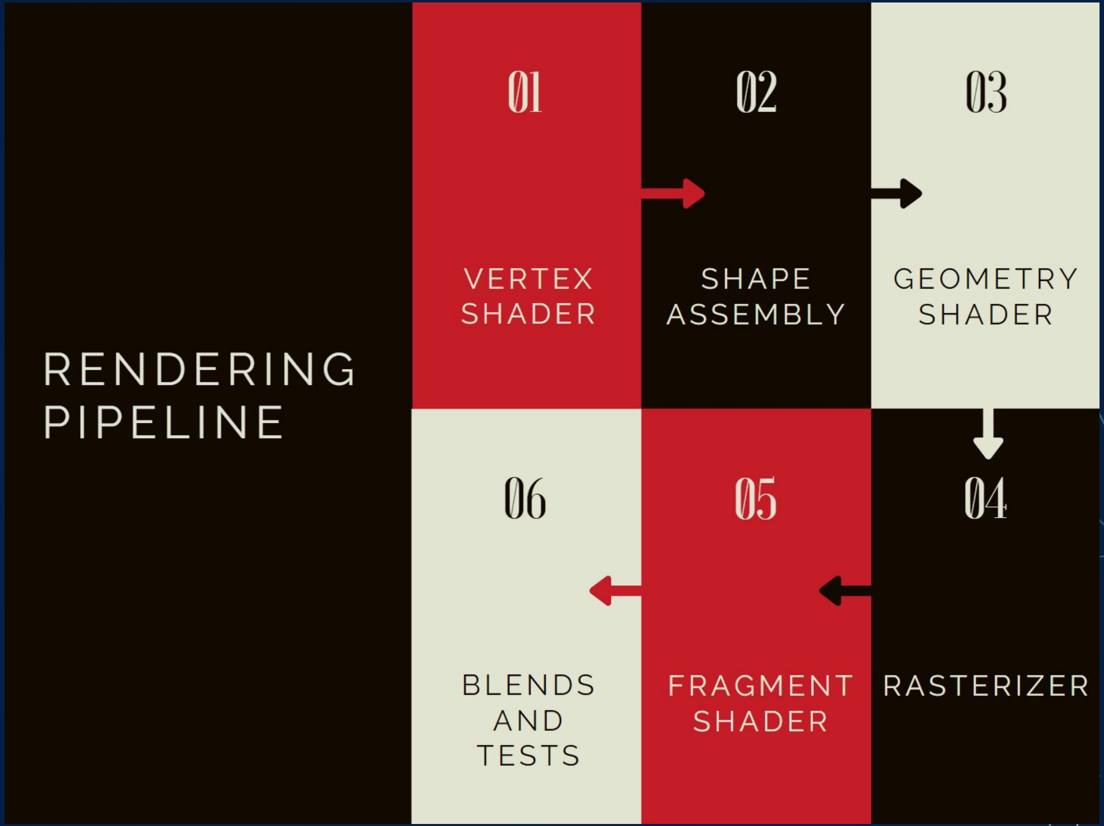
MODERN OPENGL (3.0+)

- Requires a more in-depth understanding of the GPU
- Programmable rendering pipeline
- Retained mode (core profile)

02

RENDERING PIPELINE

RENDERING PIPELINE



1. VERTEX SHADER

The **vertex shader stage** accepts vertex data as input.
Its 2 main responsibilities are:

1. 3D coordinate transformations (ex. MVP transforms)
2. Basic **vertex attribute** processing

QUESTION

What are some commonly used vertex attributes?

ANSWER

Normals, Tangents, UVs, Color etc.

We can also provide provide custom attributes needed by our specific application.

2. SHAPE ASSEMBLY

The **shape/primitive assembly stage** accepts all vertices from the vertex shader and assembles them based on the currently set **primitive state**.

Always remember: OpenGL is a state machine.

3. GEOMETRY SHADER

The **geometry shader** takes the output of the primitive assembly stage, that is, collections of vertices that form a primitive.

It can then operate on the vertices to produce new vertices and primitives.

QUESTION

Why do we need the ability to produce new vertices/shapes?
What are some common use cases?

ANSWER

There are cases where we may prefer to operate on a model geometry in a dynamic way, instead of fixed vertex data.

Example techniques: Tessellation, Subdivision
Example use case: Procedural generation (grass, terrain etc.)

ANSWER

It can also have a significant impact on performance:

1. Memory bandwidth (less vertices stored)
2. Controlled LOD (level of detail)
3. Fewer vertex shader and shape assembly executions

4. RASTERIZATION

The **rasterization stage** takes the output of the geometry shader and maps the resulting primitives to pixels on screen, creating **fragments**.

When we refer to a fragment in graphics, we mean all the data required to render a single pixel.

5. FRAGMENT SHADER

The **fragment shader** takes the fragments created by the rasterization stage and operates on them, with the main responsibility of **calculating the final color** of each fragment on screen.

At this stage we combine all the vertex information related to lighting with the lighting data of the environment based on a **lighting model**.

QUESTION

Any examples of vertex data related to lighting?

ANSWER

- Textures
- Normals
- Height information
- Metallicness
- Roughness
- Emission
- etc.

6. BLENDS AND TESTS

At the last stage of the pipeline, before the fragments are being drawn on screen, some final operations are being performed:

1. Z (depth) testing
2. Alpha testing
3. Stencil testing
4. Color blending

03

CORE PROFILE

IMMEDIATE MODE

Old OpenGL uses functions like

glBegin(), glEnd(), glVertex(), glColor() etc.

This mode of operation is called **immediate mode**.

IMMEDIATE MODE

Immediate mode has some major disadvantages:

- Lots of function call overhead to the graphics driver
- Vertex data must always be sent to the GPU, even if unchanged

RETAINED MODE

In modern OpenGL we use **retained mode** to pass vertices to the GPU.

In retained mode we make use of some GPU preserved objects:

- VBO (Vertex Buffer Object)
- VAO (Vertex Array Object)
- EBO (Element Buffer Object)

VERTEX BUFFER OBJECT

An OpenGL specific buffer object, used to store **vertex data**.

VERTEX ARRAY OBJECT

An OpenGL object, used to store **vertex state information**, needed to correctly access the vertex data, including a reference to a VBO and the data format.

ELEMENT BUFFER OBJECT

An OpenGL object, used for indexed rendering.

(Not mandatory but usually makes things easier)

CORE PROFILE

Core profile OpenGL restricts us in using only custom shaders (at least vertex and fragment) and retained mode for providing vertex data.

04

IN PRACTICE

SETUP

In the following examples we will be using:

- **GLFW** as the windowing library API
- **Core Profile OpenGL** version 3.3
- **GLM** for math operations
- **GLAD** library for correct GPU OpenGL functionality detection

SETUP

We will not explain in detail how to use GLFW and GLM functionality.

For an in-depth setup guide and tutorials for these libraries, search for the respective sections on the official documentation pages.

A SIMPLE RECTANGLE

SOME GLOBALS

```
float rect[12] = {
    -1.0f, -1.0f, 0.0f,
    -1.0f,  1.0f, 0.0f,
     1.0f, -1.0f, 0.0f,
     1.0f,  1.0f, 0.0f,
};

unsigned int rectIndices[6] = {
    0, 1, 2,
    1, 2, 3
};

GLuint rectVAO;
GLuint rectVBO;
GLuint rectEBO;
```

MAIN

```
int main()
{
    // Initialize GLFW, GLAD and OpenGL viewport
    int initCode;
    if ((initCode = initProgram()) != 0)
    {
        std::cout << "[!] Initialization error" << std::endl;

        return initCode;
    }

    // Initialize and compile shader program
    Shader shaderProgram = Shader("./vertex.shader", "./fragment.shader");

    initRectangle();

    windowMainLoopRect(shaderProgram);

    cleanup(&rectVAO, &rectVBO, &rectEBO);

    return 0;
}
```

INITIALIZE RECTANGLE

```
void initRectangle()
{
    glGenVertexArrays(1, &rectVAO);
    glGenBuffers(1, &rectVBO);
    glGenBuffers(1, &rectEBO);

    glBindVertexArray(rectVAO);
    glBindBuffer(GL_ARRAY_BUFFER, rectVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(rect), rect, GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, rectEBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(rectIndices), rectIndices, GL_STATIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
}
```

WINDOW MAIN LOOP

```
void windowMainLoopRect(Shader shaderProgram)
{
    glBindVertexArray(rectVAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glPolygonMode(GL_FRONT_AND_BACK, wireframe ? GL_LINE : GL_FILL);

    glClearColor(0.12f, 0.12f, 0.12f, 1.0f);

    while (!glfwWindowShouldClose(window))
    {
        processInput(window);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        setupShader(shaderProgram,
            glm::vec3(0.0f, 0.0f, 0.0f),
            glm::vec3(0.0f, 0.0f, 0.0f),
            glm::vec3(1.0f, 1.0f, 1.0f),
            glm::vec3(0.0f, 0.0f, 0.0f),
            glm::vec3(0.9f, 0.3f, 0.4f));

        glDrawArrays(GL_TRIANGLES, 0, 3);
        glDrawArrays(GL_TRIANGLES, 1, 3);

        glfwSwapBuffers(window);
        glfwPollEvents();
    }
}
```

SHADER SETUP

```
void setupShader(Shader shaderProgram, glm::vec3 modelPos, glm::vec3 modelRot, glm::vec3 modelScale, glm::vec3 pivot, glm::vec3 color)
{
    glm::mat4x4 model = glm::mat4x4(1.0f);
    glm::mat4x4 view = glm::mat4x4(1.0f);
    glm::mat4x4 proj = glm::mat4x4(1.0f);

    model = glm::translate(model, modelPos);

    model = glm::translate(model, pivot);
    model = glm::rotate(model, glm::radians(modelRot.x), glm::vec3(1.0f, 0.0f, 0.0f));
    model = glm::rotate(model, glm::radians(modelRot.y), glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::rotate(model, glm::radians(modelRot.z), glm::vec3(0.0f, 0.0f, 1.0f));
    model = glm::translate(model, -pivot);

    model = glm::scale(model, modelScale);

    const float radius = 2.0f;
    float camX = 0.0f;
    float camY = 0.0f;
    float camZ;

    switch (camMode)
    {
    case (CameraMode::Static):
        camZ = radius;
        break;
    case (CameraMode::Rotating):
        camX = sin(glfwGetTime()) * radius;
        camZ = cos(glfwGetTime()) * radius;
        break;
    default:
        break;
    }
```

SHADER SETUP (cont.)

```
    view = glm::lookAt(glm::vec3(camX, camY, camZ), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0, 1.0, 0.0));
    proj = glm::perspective(glm::radians(90.0f), (float)windowW / (float)windowH, 0.1f, 1000.0f);

    shaderProgram.Use();

    GLuint colorLoc = glGetUniformLocation(shaderProgram.GetShaderID(), "color");
    glUniform4f(colorLoc, color.x, color.y, color.z, 1.0f);
    GLuint modelLoc = glGetUniformLocation(shaderProgram.GetShaderID(), "model");
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    GLuint viewLoc = glGetUniformLocation(shaderProgram.GetShaderID(), "view");
    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
    GLuint projLoc = glGetUniformLocation(shaderProgram.GetShaderID(), "projection");
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(proj));
}
```

SHADER CLASS

You can find a simplified version of the Shader class here:

https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/shader_s.h

(Implementation by Joey De Vries)

SHADERS

Core profile OpenGL requires us to provide at least a vertex and a fragment shader for our application.

VERTEX SHADER

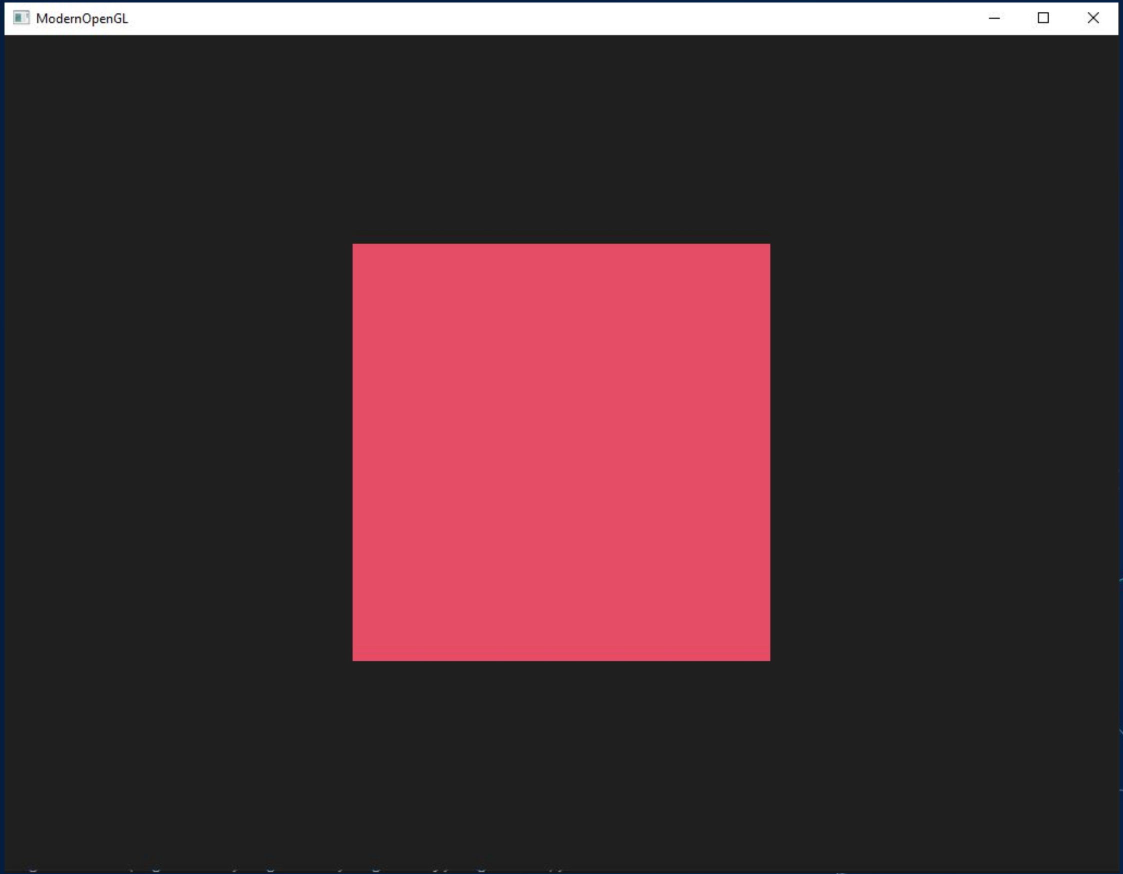
```
#version 330 core

layout(location = 0) in vec3 aPos;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

FRAGMENT SHADER

```
#version 330 core

out vec4 FragColor;

uniform vec4 color;

void main()
{
    FragColor = color;
}
```

(DEMO)

A SIMPLE RECTANGLE



DRAWING A SPHERE

PROCEDURAL GENERATION

Let’s now explore the possibility of dynamically generating the mesh for a sphere object.

DATA GENERATION

```cpp
void Sphere::CreateVertices()
{
    std::vector<float>().swap(vertexData);

    float x;
    float y;
    float z;
    float stackAngle;
    float sectorRadius;

    float stackStep = glm::pi<float>() / (float)stacks;
    float sectorStep = 2.0f * glm::pi<float>() / (float)sectors;

    for (int i = 0; i <= stacks; i++)
    {
        stackAngle = (glm::pi<float>() * 0.5f) - (i * stackStep);
        sectorRadius = radius * glm::cos(stackAngle);
        z = radius * glm::sin(stackAngle);

        for (int j = 0; j <= sectors; j++)
        {
            x = sectorRadius * glm::cos(j * sectorStep);
            y = sectorRadius * glm::sin(j * sectorStep);

            vertexData.push_back(x);
            vertexData.push_back(z);
            vertexData.push_back(y);

            vertexData.push_back(x / radius);
            vertexData.push_back(z / radius);
            vertexData.push_back(y / radius);
        }
    }
}
```

INDICES GENERATION

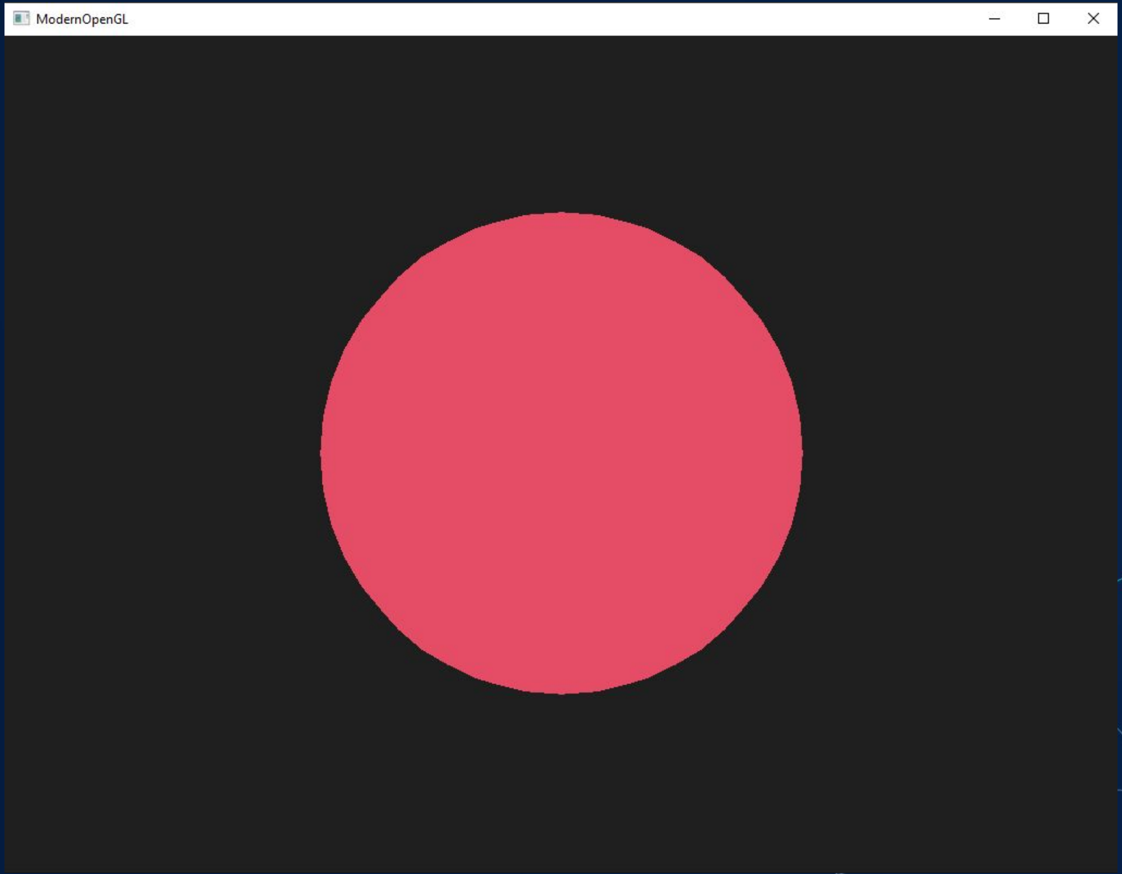
```
void Sphere::CreateIndices()
{
    std::vector<unsigned int>().swap(indices);

    unsigned int sid;
    unsigned int nsid;

    for (int i = 0; i < stacks; i++)
    {
        sid = i * (sectors + 1);
        nsid = (i + 1) * (sectors + 1);

        for (int j = 0; j < sectors; j++)
        {
            if (i != 0)
            {
                indices.push_back(sid);
                indices.push_back(nsid);
                indices.push_back(sid + 1);
            }

            if (i != stacks - 1)
            {
                indices.push_back(sid + 1);
                indices.push_back(nsid);
                indices.push_back(nsid + 1);
            }

            sid++;
            nsid++;
        }
    }
}
```

OPENGL OBJECTS SETUP

```
void Sphere::Init()
{
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, vertexData.size() * sizeof(float), vertexData.data(), GL_DYNAMIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned int), indices.data(), GL_DYNAMIC_DRAW);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
}
```

(DEMO)

DRAWING A SPHERE



LET THERE BE LIGHT

LIGHTING

Simulating lighting is one of the things that really makes shader programming shine (pun intended)

Let’s try to reproduce the phong lighting model in our scene.

VERTEX SHADER

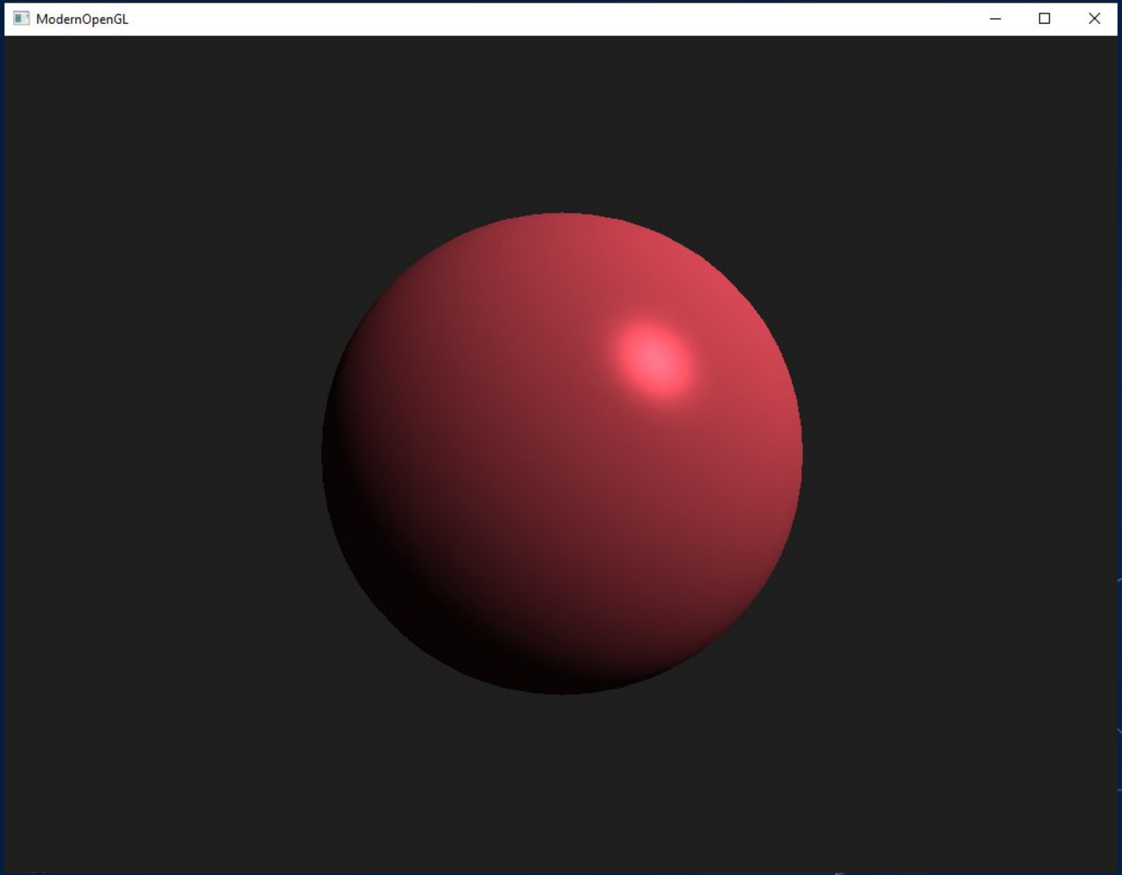
```
#version 330 core

layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;

out vec3 Normal;
out vec3 WorldPos;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform mat4 normalMatrix;

void main()
{
    gl_Position = projection * view * model * vec4(aPos.x, aPos.y, aPos.z, 1.0);
    WorldPos = vec3(model * vec4(aPos.x, aPos.y, aPos.z, 1.0));
    Normal = mat3(normalMatrix) * aNormal;
}
```

FRAGMENT SHADER

```glsl
#version 330 core

in vec3 Normal;
in vec3 WorldPos;

out vec4 FragColor;

uniform vec4 color;
uniform vec4 lightColor;
uniform vec3 lightPos;
uniform vec3 cameraPos;

void main()
{
    float ambientLightStrength = 0.05f;
    vec3 ambient = vec3(ambientLightStrength * lightColor);

    vec3 lightDir = normalize(lightPos - WorldPos);
    vec3 normal = normalize(Normal);

    float diffuseContribution = max(dot(normal, lightDir), 0.0f);
    vec3 diffuse = vec3(diffuseContribution * lightColor);

    vec3 viewDir = normalize(cameraPos - WorldPos);
    vec3 reflectDir = reflect(-lightDir, normal);

    float specularStrength = 0.9f;
    float specularContribution = pow(max(dot(viewDir, reflectDir), 0.0f),32);
    vec3 specular = specularStrength * specularContribution * vec3(lightColor);

    vec3 result = vec3(color) * (ambient + diffuse + specular);

    FragColor = vec4(result, 1.0f);
}
```

(DEMO)

LET THERE BE LIGHT



LET'S PLAY AROUND WITH SOME EFFECTS

(DEMO)

THANKS

Questions?

mpenos.ks@gmail.com

@deg3x

Konstantinos Benos

RESOURCES

- Graphics tutorials by Inigo Quilez: https://iquilezles.org/
- SigGraph (Graphics Conference): https://www.siggraph.org/
- LearnOpenGL (by Joey de Vries): https://learnopengl.com/
- Online shader coding platform: https://www.shadertoy.com/