Graphics Project

Name	ID
Mohamed Adel Mahmoud Taha	20011629
Ziad Reda Saad Ali	19015717
Mostafa Ahmed Abdelhamed	18011774
Mohamed Haroun Saleh	19016520

Color Shader:

The provided code contains a vertex shader and a fragment shader written in GLSL for OpenGL. The vertex shader takes a vertex position (`aPosition`), applies a series of transformations using the model, view, and projection matrices (`u_ModelMat`, `u_ViewMat`, `u_ProjectionMat`), and outputs the transformed position to `gl_Position`. The fragment shader takes a uniform color (`u_Color`) and sets the color of the fragment (`gl_FragColor`) to this value. This setup is commonly used to render 3D objects with transformations and a uniform color.

```
#shader vertex
#version 330 core

attribute vec4 aPosition;

uniform mat4 u_ModelMat;
uniform mat4 u_ViewMat;
uniform mat4 u_ProjectionMat;

void main()
{
	gl_Position = u_ProjectionMat * u_ViewMat * u_ModelMat * aPosition;
}

#shader fragment
#version 330 core

uniform vec4 u_Color;

void main()
{
	gl_FragColor = u_Color;
}
```

Texture Shader:

This code defines a vertex shader and a fragment shader for rendering textured objects with lighting effects.

The vertex shader processes each vertex by applying model, view, and projection transformations to determine the final position (`gl_Position`). It also calculates texture coordinates (`v_TexCoord`), transforms normals (`v_Normal`) for proper lighting calculations, and computes the position in world space (`v_Position`).

The fragment shader handles the color and lighting of each fragment (pixel). It uses a texture (`u_Texture`) and a specified opacity (`u_TexOpacity`). The shader calculates ambient, diffuse, and specular lighting based on a light source at the origin (`lightPos`) with a

specified color (`lightColor`). The final color of the fragment (`gl_FragColor`) is determined by combining these lighting components with the texture color, considering the specified opacity.

```
#shader vertex
#version 330 core

attribute vec4 aPosition;
attribute vec2 aTexCoord;
attribute vec3 aNormal;
uniform mat4 u_ModelMat;
uniform mat4 u_ViewMat;
uniform mat4 u_ProjectionMat;
varying vec2 v_TexCoord;
varying vec3 v_Normal;
varying vec3 v_Position;

void main()
{
	gl_Position = u_ProjectionMat * u_ViewMat * u_ModelMat * aPosition;
	v_TexCoord = aTexCoord;
	v_Normal = mat3(transpose(inverse(u_ModelMat))) * aNormal;
	v_Position = vec3(u_ModelMat * aPosition);
}

#shader fragment
#version 330 core
uniform float u_TexOpacity;
uniform sampler2D u_Texture;
varying vec2 v_TexCoord;
varying vec3 v_Normal;
varying vec3 v_Position;

void main()
{
	vec3 lightPos = vec3(0.0f);
	vec3 lightColor = vec3(1.0f, 0.8f, 0.8f);
	// Ambient
	float ambientStrength = 0.8f;
	vec3 ambient = ambientStrength * lightColor;
	// Diffuse
	vec3 norm = normalize(v_Normal);
	vec3 lightDir = normalize(lightPos - v_Position);
	float diff = max(dot(norm, lightDir), 0.0);
	vec3 diffuse = diff * lightColor;
	// Specular
	float specularStrength = 1.2f;
	vec3 viewDir = normalize(-v_Position);
	vec3 reflectDir = reflect(-lightDir, norm);
	float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
	vec3 specular = specularStrength * spec * lightColor;

	vec4 texColor = texture(u_Texture, v_TexCoord);
	vec4 result = vec4((ambient + diffuse + specular) * texColor.rgb, u_TexOpacity);
	gl_FragColor = result;
}
```

Camera:

This C++ code defines methods for a `Camera` class used in 3D graphics, which handles camera movement and orientation.

- `GetViewMatrix()`: Returns the view matrix using the camera's position, front vector, and up vector with the `glm::lookAt` function.

- `ProcessKeyboard()`: Adjusts the camera's position based on keyboard input for movement directions (FORWARD, BACKWARD, LEFT, RIGHT) and the elapsed time (`deltaTime`).

- `ProcessMouseMovement()`: Updates the camera's yaw and pitch angles based on mouse movement offsets. It optionally constrains the pitch to prevent excessive up or down tilting, then updates the camera's direction vectors.

- `ProcessMouseScroll()`: Adjusts the camera's zoom level based on scroll input, clamping the value between 1.0f and 45.0f.

- `updateCameraVectors()`: Recalculates the camera's front, right, and up vectors based on the current yaw and pitch angles, ensuring the camera's orientation is correctly updated.

```cpp
#include "Camera.h"

glm::mat4 Camera::GetViewMatrix() const
{
	return glm::lookAt(m_Position, m_Position + m_Front, m_Up);
}

void Camera::ProcessKeyboard(CameraMovement direction, float deltaTime)
{
	float velocity = m_MovementSpeed * deltaTime;
	if (direction == FORWARD)
		m_Position += m_Front * velocity;
	if (direction == BACKWARD)
		m_Position -= m_Front * velocity;
	if (direction == LEFT)
		m_Position -= m_Right * velocity;
	if (direction == RIGHT)
		m_Position += m_Right * velocity;
}

void Camera::ProcessMouseMovement(float xOffset, float yOffset, bool constrainPitch)
{
	xOffset *= m_MouseSensitivity;
	yOffset *= m_MouseSensitivity;

	m_Yaw += xOffset;
	m_Pitch += yOffset;

	if (constrainPitch)
	{
		if (m_Pitch > 89.0f)
			m_Pitch = 89.0f;
		if (m_Pitch < -89.0f)
			m_Pitch = -89.0f;
	}
	updateCameraVectors();
}

void Camera::ProcessMouseScroll(float yOffset)
{
	if (m_Zoom >= 1.0f && m_Zoom <= 45.0f)
		m_Zoom -= yOffset;
	if (m_Zoom <= 1.0f)
		m_Zoom = 1.0f;
	if (m_Zoom >= 45.0f)
		m_Zoom = 45.0f;
}

void Camera::updateCameraVectors()
{
	glm::vec3 front = glm::vec3(0.0f);
	front.x = cos(glm::radians(m_Yaw)) * cos(glm::radians(m_Pitch));
	front.y = sin(glm::radians(m_Pitch));
	front.z = sin(glm::radians(m_Yaw)) * cos(glm::radians(m_Pitch));
	m_Front = glm::normalize(front);
	m_Right = glm::normalize(glm::cross(m_Front, m_WorldUp));
	m_Up = glm::normalize(glm::cross(m_Right, m_Front));
}
```

Debugger:

This C++ code provides utility functions for OpenGL error handling and logging, aimed at debugging graphics applications.

- `glClearError()`: Clears all existing OpenGL errors by calling `glGetError()` in a loop until no errors remain (`GL_NO_ERROR`).
- `glLogCall()`: Logs OpenGL errors. It repeatedly checks for errors using `glGetError()` and, if an error is found, prints a detailed error message to the console. The message includes the error code, the function name, the file, and the line number where the error occurred. It returns `false` if an error is detected, otherwise returns `true`.

These functions are useful for identifying and troubleshooting OpenGL issues in your graphics code.

```
#include <iostream>
#include "Debugger.h"
using namespace std;

void glClearError()
{
	while (glGetError() != GL_NO_ERROR); // !glGetError()
}

bool glLogCall(const char* function, const char* file, int line)
{
	while (unsigned int error = glGetError())
	{
		cout << "[OpenGL Error!!] " << "(" << error << "): " << function << " "
<< file << " " << line << endl;
		return false;
	}
	return true;
}
```

Element Buffer:

This C++ code defines the `ElementBuffer` class for managing OpenGL element buffer objects (EBOs). The constructor generates a buffer, binds it, and allocates memory for it using provided data. The destructor deletes the buffer to free resources. The `bind` method binds the buffer, and the `unbind` method unbinds it. Error checking is done using a macro (`glCall`) for each OpenGL call.

```cpp
#include "ElementBuffer.h"
#include "Debugger.h"

ElementBuffer::ElementBuffer(unsigned int count, const void* data) : m_count(count)
{
	ASSERT(sizeof(unsigned int) == sizeof(GLint));

	// (number of buffer object names to be generated, pointer to array in which
the generated object names are stored)
	glCall(glGenBuffers(1, &m_RendererId));

	// (the target in which the buffer object is bound, the name of a buffer ob-
ject)
	glCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererId));

	// (target, data size in bytes, data, usage)
	glCall(glBufferData(GL_ELEMENT_ARRAY_BUFFER, count * sizeof(unsigned int),
data, GL_STATIC_DRAW));
}

ElementBuffer::~ElementBuffer()
{
	glDeleteBuffers(1, &m_RendererId);
}

void ElementBuffer::bind() const
{
	glCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_RendererId));
}

void ElementBuffer::unbind() const
{
	glCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0));
}
```

Renderer:

```cpp
/*
* RENDERER CLASS
* This class is responsible for rendering the objects on the screen.
* It has two methods:
* 1. draw: This method takes a VertexArray, ElementBuffer and Shader as input and
draws the object on the screen.
* 2. clear: This method clears the screen.
* The draw method binds the VertexArray, ElementBuffer and Shader and then calls the
glDrawElements method to draw the object.
* The clear method clears the screen by calling the glClear method.
*/
#include "Debugger.h"
#include "Renderer.h"

void Renderer::draw(const VertexArray& vao, const ElementBuffer& ebo, const Shader&
shader) const
{
	vao.bind();
	ebo.bind();
	shader.bind();
	glCall(glDrawElements(GL_TRIANGLES, ebo.getCount(), GL_UNSIGNED_INT,
nullptr));
}

void Renderer::clear() const
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

Shader:

```
/*
* SHADER CLASS
* This class is responsible for creating and managing the shaders.
* It has the following methods:
* 1. Shader: This is the constructor of the class. It takes the file path of the
shader as input and creates the shader.
* 2. bind: This method binds the shader.
* 3. unbind: This method unbinds the shader.
* 4. deleteProgram: This method deletes the shader program.
* 5. setUniform1i: This method sets the value of an integer uniform in the shader.
* 6. setUniform1f: This method sets the value of a float uniform in the shader.
* 7. setUniform2f: This method sets the value of a vec2 uniform in the shader.
* 8. setUniform3f: This method sets the value of a vec3 uniform in the shader.
* 9. setUniform4f: This method sets the value of a vec4 uniform in the shader.
* 10. setUniformMat4f: This method sets the value of a mat4 uniform in the shader.
* 11. parseBasicShaders: This method parses the basic shaders from the shader file.
* 12. createShader: This method creates a shader.
* 13. createProgram: This method creates a shader program.
* 14. getUniformLocation: This method gets the location of a uniform in the shader.
* The Shader class has a member variable m_RendererId which stores the id of the
shader program.
* The Shader class also has a member variable m_UniformLocationCache which stores
the locations of the uniforms in the shader.
* The Shader class uses the Debugger class to log errors and debug information.
* The Shader class uses the glm library for matrix and vector operations.
* The Shader class uses the OpenGL API for creating and managing the shaders.
* The Shader class uses the string, fstream and sstream libraries for file in-
put/output and string stream operations.
* The Shader class uses the BasicShaders struct to store the basic shaders parsed
from the shader file.
* The Shader class uses the ShaderType enum class to store the type of shader.
*/
#include "Shader.h"
#include "Debugger.h"
#include <iostream> // input/output stream
#include <fstream>  // file stream
#include <sstream>  // string stream

Shader::Shader(const string& filepath) : m_FilePath(filepath)
{
	BasicShaders basicShaders = parseBasicShaders(filepath);
	m_RendererId = createProgram(basicShaders.vertexShaderSrc, basicShaders.frag-
mentShaderSrc);
}

void Shader::bind() const
{
	glCall(glUseProgram(m_RendererId));
}

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

void Shader::deleteProgram() const
{
	glCall(glDeleteProgram(m_RendererId));
```

```
}

void Shader::setUniform1i(const string& name, int v0) const
{
	glCall(glUniform1i(getUniformLocation(name), v0));
}

void Shader::setUniform1f(const string& name, float v0) const
{
	glCall(glUniform1f(getUniformLocation(name), v0));
}

void Shader::setUniform2f(const string& name, float v0, float v1) const
{
	glCall(glUniform2f(getUniformLocation(name), v0, v1));
}

void Shader::setUniform3f(const string& name, float v0, float v1, float v2) const
{
	glCall(glUniform3f(getUniformLocation(name), v0, v1, v2));
}

void Shader::setUniform4f(const string& name, float v0, float v1, float v2, float
v3) const
{
	glCall(glUniform4f(getUniformLocation(name), v0, v1, v2, v3));
}

void Shader::setUniformMat4f(const string& name, const glm::mat4& matrix) const
{
	glCall(glUniformMatrix4fv(getUniformLocation(name), 1, GL_FALSE,
&matrix[0][0]));
}

BasicShaders Shader::parseBasicShaders(const string& filepath) const
{
	// Open the input file
	ifstream stream(filepath);

	enum class ShaderType {
		NONE = -1, VERTEX = 0, FRAGMENT = 1
	};

	ShaderType shaderType = ShaderType::NONE;
	stringstream ss[2];
	string line;
	while (getline(stream, line))
	{
		if (line.find("#shader") != string::npos)
		{
			if (line.find("vertex") != string::npos)
				shaderType = ShaderType::VERTEX;
			else if (line.find("fragment") != string::npos)
				shaderType = ShaderType::FRAGMENT;
		}
		else {
			if (shaderType != ShaderType::NONE)
				ss[(int)shaderType] << line << '\n';
```

```
			}
		}

		return { ss[0].str(), ss[1].str() };
}

unsigned int Shader::createShader(unsigned int type, const string& source) const
{
		// Shader source code creation and compilation step
		glCall(unsigned int shaderId = glCreateShader(type));
		const char* const src = source.c_str();
		glCall(glShaderSource(shaderId, 1, &src, nullptr));
		glCall(glCompileShader(shaderId));

		// Error printing step
		int status;
		glCall(glGetShaderiv(shaderId, GL_COMPILE_STATUS, &status));

		if (status == GL_FALSE) // !status
		{
			// Getting the length of the error message
			int length;
			glCall(glGetShaderiv(shaderId, GL_INFO_LOG_LENGTH, &length));

			// Getting the error message
			char* message = (char*)_malloca(length);
			glCall(glGetShaderInfoLog(shaderId, length, &length, message));

			// Logging the error message
			cout << "Failed to create " << (type == GL_VERTEX_SHADER ? "vertex" :
"fragment") << " shader!" << endl;
			cout << message << endl;

			glCall(glDeleteShader(shaderId));
			return 0;
		}

		return shaderId;
}

unsigned int Shader::createProgram(const string& vertexShaderSrc, const string&
fragmentShaderSrc) const
{
		glCall(unsigned int programId = glCreateProgram()); // Create/Initialize the
program
		unsigned int vs = createShader(GL_VERTEX_SHADER, vertexShaderSrc);
		unsigned int fs = createShader(GL_FRAGMENT_SHADER, fragmentShaderSrc);

		glCall(glAttachShader(programId, vs)); // Attach the vertex shader
		glCall(glAttachShader(programId, fs)); // Attach the fragment shader

		// Two regular steps at the end of creating any program
		glCall(glLinkProgram(programId));
		glCall(glValidateProgram(programId));

		// Delete the unused shaders
		glCall(glDetachShader(programId, vs));
		glCall(glDetachShader(programId, fs));
```

```
	glCall(glDeleteShader(vs));
	glCall(glDeleteShader(fs));
	return programId;
}

int Shader::getUniformLocation(const string& name) const
{
	if (m_UniformLocationCache.count(name))
		return m_UniformLocationCache.at(name);

	glCall(int location = glGetUniformLocation(m_RendererId, name.c_str()));
	if (location == -1)
		cout << "Warning: Uniform '" << name << "' doesn't exist!!" << endl;
	return location;
}
```

Texture:

```
/*
* TEXTURE CLASS
* Texture class is used to load textures from file and bind them to the GPU.
* It uses stb_image.h to load images from file.
* It also provides functions to bind and unbind textures.
* It also provides a function to delete the texture from GPU.
*/
#include "Texture.h"
#include "Debugger.h"
#include "stb_image/stb_image.h"

Texture::Texture(const string& filePath) : m_FilePath(filePath), m_Local-
Buffer(nullptr), m_Width(0), m_Height(0), m_BPP(0)
{
	stbi_set_flip_vertically_on_load(1);
	m_LocalBuffer = stbi_load(filePath.c_str(), &m_Width, &m_Height, &m_BPP, 4);

	glCall(glGenTextures(1, &m_RendererId));
	glCall(glBindTexture(GL_TEXTURE_2D, m_RendererId));

	// always done
	glCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR));
	glCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR));
	glCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
	glCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));

	glCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, m_Width, m_Height, 0,
GL_RGBA, GL_UNSIGNED_BYTE, m_LocalBuffer));
	glCall(glBindTexture(GL_TEXTURE_2D, 0));
	if (m_LocalBuffer)
		stbi_image_free(m_LocalBuffer);
}

void Texture::bind(unsigned int slot) const
{
	glCall(glActiveTexture(GL_TEXTURE0 + slot));
	glCall(glBindTexture(GL_TEXTURE_2D, m_RendererId));
}

void Texture::unbind() const
{
	glCall(glBindTexture(GL_TEXTURE_2D, 0));
}

void Texture::deleteTexture() const
{
	glCall(glDeleteTextures(1, &m_RendererId));
}
```

Vertex Array:

```
/*
* VERTEX ARRAY
* A vertex array object (VAO) is an object which contains one or more vertex buffer
objects and is designed to store the information
* of vertex attributes in a single object. The VAO can store multiple VBOs and their
attribute configurations, and when we bind a VAO,
* all the required VBOs and their configurations are binded automatically.
*/
#include "VertexArray.h"
#include "Debugger.h"

VertexArray::VertexArray()
{
	glCall(glGenVertexArrays(1, &m_RendererId));
}

VertexArray::~VertexArray()
{
	glCall(glDeleteVertexArrays(1, &m_RendererId));
}

void VertexArray::addBuffer(const VertexBuffer& vbo, const VertexBufferLayout& lay-
out) const
{
	bind();
	vbo.bind();
	const vector<VertexAttribute>& attributes = layout.getAttributes();
	for (unsigned int i = 0, offset = 0; i < attributes.size(); i++) {
		VertexAttribute attribute = attributes[i];

		// (the index of the attribute to be enabled (0, 1, 2, ...) (vertex po-
sition has index 0))
		glCall(glEnableVertexAttribArray(i));

		// (attribute index, number of components in attribute, type of attrib-
ute, normalized or not, stride (size of vertex),
		// pointer (offset of attribute)(0 for the first attribute))
		glCall(glVertexAttribPointer(i, attribute.m_Count, attribute.m_Type,
attribute.m_Normalized, layout.getStride(), (const void*)offset));
		offset += attribute.getAttribSize();
	}
}

void VertexArray::bind() const
{
	glCall(glBindVertexArray(m_RendererId));
}

void VertexArray::unbind() const
{
	glCall(glBindVertexArray(0));
}
```

Vertex Buffer:

```
/*
* VERTEX BUFFER
* A vertex buffer object (VBO) is an OpenGL feature that provides methods for up-
loading vertex data (position, normal vector, color, etc.) to the video device for
non-immediate-mode rendering.
* VBOs offer substantial performance gains over immediate mode rendering primarily
because the data resides in the video device memory rather than the system memory
and so it can be rendered directly by the video device.
* VBOs are created and managed using the OpenGL core profile.
* A VBO is created by first generating a buffer object name using glGenBuffers,
binding the buffer object to the GL_ARRAY_BUFFER target using glBindBuffer, and then
allocating memory for the buffer object using glBufferData.
* The buffer object is then filled with vertex data using glBufferSubData or glMap-
Buffer.
* The buffer object is then bound to the GL_ARRAY_BUFFER target using glBindBuffer,
and the vertex data is rendered using glDrawArrays or glDrawElements.
* The buffer object is deleted using glDeleteBuffers.
* The VBO is bound using glBindBuffer with the target GL_ARRAY_BUFFER.
* The VBO is unbound using glBindBuffer with the target GL_ARRAY_BUFFER and the
buffer object name 0.
* The VBO is deleted using glDeleteBuffers.
* The VBO is used to store vertex data in the GPU memory.
*/
#include "VertexBuffer.h"
#include "Debugger.h"

VertexBuffer::VertexBuffer(unsigned int size, const void* data)
{
	// (number of buffer object names to be generated, pointer to array in which
the generated object names are stored)
	glCall(glGenBuffers(1, &m_RendererId));

	// (the target in which the buffer object is bound, the name of a buffer ob-
ject)
	glCall(glBindBuffer(GL_ARRAY_BUFFER, m_RendererId));

	// (target, data size in bytes, data, usage)
	glCall(glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW));
}

VertexBuffer::~VertexBuffer()
{
	glDeleteBuffers(1, &m_RendererId);
}

void VertexBuffer::bind() const
{
	glCall(glBindBuffer(GL_ARRAY_BUFFER, m_RendererId));
}

void VertexBuffer::unbind() const
{
	glCall(glBindBuffer(GL_ARRAY_BUFFER, 0));
}
```

Vertex Buffer Layout:

```
/*
* VERTEX BUFFER LAYOUT CLASS
* VertexBufferLayout class is used to store the layout of the vertex buffer.
* It stores the attributes of the vertex buffer.
* It also stores the stride of the vertex buffer.
*/
#include "VertexBufferLayout.h"

void VertexBufferLayout::addAttrib(VertexAttribute attribute)
{
	m_Attributes.push_back(attribute);
	m_Stride += attribute.getAttribSize();
}

void VertexBufferLayout::addAttributes(vector<VertexAttribute> attributes)
{
	for (VertexAttribute& attribute : attributes)
	{
		addAttrib(attribute);
	}
}
```