Gebze Technical University

Computer Engineering

Computer Graphics(CSE 461)

Homework #2

Berkan AKIN

171044073

OpenGL Application Report for Programming Assignment 2

Introduction

This report describes the implementation of an OpenGL application that meets the requirements outlined in the CSE 461 Programming Assignment 2 document. The application features a bounded 3D world, perspective projection, multiple objects with various textures and materials, mouse and keyboard interactions, multiple light sources, and collision detection.

Minimum Requirements Implementation

1. Bounded 3D World

The application defines a bounded 3D world using **WORLD_BOUNDS_MIN** and **WORLD_BOUNDS_MAX**:

```
const glm::vec3 WORLD_BOUNDS_MIN = glm::vec3(-5.0f, -5.0f, -5.0f);
const glm::vec3 WORLD_BOUNDS_MAX = glm::vec3(5.0f, 5.0f, 5.0f);
```

2. Perspective Projection

Perspective projection is set up using **glm::perspective** in the render loop:

```
// Set up the view and projection matrices
glm::mat4 projection = glm::perspective(glm::radians(fov), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "view"), 1, GL_FALSE, glm::value_ptr(view));
```

3. Realistic and Logical Ordering of Rendered Objects

Z-buffering is enabled to ensure proper ordering of rendered objects:

```
// Configure global OpenGL state
glEnable(GL_DEPTH_TEST);
```

4. Four Different Objects

Four different objects are created: a cube, a pyramid, a sphere, and a cylinder. The vertex data for these objects are set up in separate VAOs and VBOs.

```
// Set up vertex data, buffers, and configure vertex attributes
float cubeVertices[] = {
    // positions          // normals           // texture coords
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f, -1.0f,  0.0f, 0.0f,
```

```
// Pyramid vertex data
float pyramidVertices[] = {
    // positions          // normals           // texture coords
    // Front face
     0.0f,  0.5f,  0.0f,  0.0f,  0.4472f,  0.8944f,  0.5f,  1.0f,
```

5. Object Copies and Rotation

Multiple copies of the cube and pyramid are created at different positions and scales. One instance of the sphere rotates around the cube:

```
// Render cube and its copies
for (int i = 0; i < 3; ++i) {
    glBindVertexArray(VAOs[0]);
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(i * 1.5f, 0.0f, 0.0f));
    model = glm::scale(model, glm::vec3(0.5f, 0.5f, 0.5f));
    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "model"), 1, GL_FALSE, glm::value_ptr(model));
    glDrawArrays(GL_TRIANGLES, 0, 36);
}

// Render pyramid and its copies
for (int i = 0; i < 3; ++i) {
    glBindVertexArray(VAOs[1]);
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, glm::vec3(2.0f, i * 1.5f, 0.0f));
    model = glm::scale(model, glm::vec3(0.5f, 0.5f, 0.5f));
    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "model"), 1, GL_FALSE, glm::value_ptr(model));
    glDrawArrays(GL_TRIANGLES, 0, 12);
}
```

6. Different Textures

Different textures are applied to the objects, and one object has two textures blended:

```
// Load and create textures
unsigned int texture1, texture2;
glGenTextures(1, &texture1);
glBindTexture(GL_TEXTURE_2D, texture1);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
int width, height, nrChannels;
unsigned char *data = stbi_load("wall.jpg", &width, &height, &nrChannels, 0);
if (data)
{
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
    std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(data);

glGenTextures(1, &texture2);
glBindTexture(GL_TEXTURE_2D, texture2);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
data = stbi_load("specular.jpeg", &width, &height, &nrChannels, 0);
if (data)
{
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
    std::cout << "Failed to load texture" << std::endl;
}
stbi_image_free(data);

// Set texture units
glUseProgram(shaderProgram);
glUniform1i(glGetUniformLocation(shaderProgram, "texture1"), 0);
glUniform1i(glGetUniformLocation(shaderProgram, "texture2"), 1);
```

7. Phong Shading

Phong shading is implemented in the fragment shader and blended with textures:

fragment_shader.frag

```glsl
out vec4 FragColor;

struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};

struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;

uniform vec3 viewPos;
uniform Material material;
uniform Light lights[3];
uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
    vec3 ambient = vec3(texture(texture1, TexCoords)) * material.ambient;

    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);

    vec3 result = ambient;
    for (int i = 0; i < 3; i++) {
        vec3 lightDir = normalize(lights[i].position - FragPos);
        float diff = max(dot(norm, lightDir), 0.0);
        vec3 diffuse = lights[i].diffuse * diff * vec3(texture(texture1, TexCoords));

        vec3 reflectDir = reflect(-lightDir, norm);
        float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
        vec3 specular = lights[i].specular * spec * vec3(texture(texture2, TexCoords));

        result += (diffuse + specular);
    }

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

8. Four Different Materials

Four different materials are defined and applied to objects:

```cpp
// Material properties
struct Material {
    glm::vec3 ambient;
    glm::vec3 diffuse;
    glm::vec3 specular;
    float shininess;
};

Material materials[4] = {
    {glm::vec3(1.0f, 0.5f, 0.31f), glm::vec3(1.0f, 0.5f, 0.31f), glm::vec3(0.5f, 0.5f, 0.5f), 32.0f},
    {glm::vec3(0.5f, 0.5f, 0.5f), glm::vec3(0.5f, 0.5f, 0.5f), glm::vec3(0.5f, 0.5f, 0.5f), 64.0f},
    {glm::vec3(0.3f, 0.3f, 0.3f), glm::vec3(0.3f, 0.3f, 0.3f), glm::vec3(0.3f, 0.3f, 0.3f), 16.0f},
    {glm::vec3(0.1f, 0.1f, 0.1f), glm::vec3(0.1f, 0.1f, 0.1f), glm::vec3(0.1f, 0.1f, 0.1f), 8.0f}
};
```

9. Mouse Interaction

Mouse look around and scroll zoom are implemented:

```cpp
void mouse_callback(GLFWwindow* window, double xpos, double ypos) {
    if (firstMouse) {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-coordinates range from bottom to top
    lastX = xpos;
    lastY = ypos;

    float sensitivity = 0.1f;
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

    if (pitch > 89.0f)
        pitch = 89.0f;
    if (pitch < -89.0f)
        pitch = -89.0f;

    glm::vec3 front;
    front.x = cos(glm::radians(yaw)) * cos(glm::radians(pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(pitch));
    cameraFront = glm::normalize(front);
}
```

10. Keyboard Interaction

WASD and arrow keys are used for navigation, and ESC quits the program:

```cpp
// Process all input
void processInput(GLFWwindow *window) {
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    float cameraSpeed = 2.5f * deltaTime;
    glm::vec3 newPos = cameraPos;
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        newPos += cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        newPos -= cameraSpeed * cameraFront;
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        newPos -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        newPos += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
    if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
        newPos += cameraSpeed * cameraUp;
    if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
        newPos -= cameraSpeed * cameraUp;

    // Check boundaries
    if (newPos.x >= WORLD_BOUNDS_MIN.x && newPos.x <= WORLD_BOUNDS_MAX.x &&
        newPos.y >= WORLD_BOUNDS_MIN.y && newPos.y <= WORLD_BOUNDS_MAX.y &&
        newPos.z >= WORLD_BOUNDS_MIN.z && newPos.z <= WORLD_BOUNDS_MAX.z) {
        // Check for collisions with objects
        std::vector<glm::vec3> objectPositions = {
            glm::vec3(0.0f, 0.0f, 0.0f),  // Cube position
            glm::vec3(2.0f, 0.0f, 0.0f),  // Another Cube position
            glm::vec3(2.0f, 1.5f, 0.0f),  // Pyramid position
            glm::vec3(0.0f, 2.0f, -2.0f), // Cylinder position
            glm::vec3(-1.5f, 0.0f, 1.5f)  // Sphere position
        };

        glm::vec3 objectSize = glm::vec3(1.0f); // Approximate size for all objects
        bool collisionDetected = false;

        for (const auto& pos : objectPositions) {
            if (checkCollision(newPos, glm::vec3(0.5f), pos, objectSize)) {
                collisionDetected = true;
                break;
            }
        }

        if (!collisionDetected) {
            cameraPos = newPos;
        }
```

11. Multiple Light Sources

Three light sources are defined, including one directional and one point light source:

```cpp
// Light properties
// Lighting
struct Light {
    glm::vec3 position;
    glm::vec3 direction;
    glm::vec3 ambient;
    glm::vec3 diffuse;
    glm::vec3 specular;
    float constant;
    float linear;
    float quadratic;
    bool isDirectional;
    bool isPoint;
};

std::vector<Light> lights = {
    {glm::vec3(1.2f, 1.0f, 2.0f), glm::vec3(-0.2f, -1.0f, -0.3f), glm::vec3(0.2f, 0.2f, 0.2f), glm::vec3(0.5f, 0.5f, 0.5f), glm::vec3(1.0f, 1.0f, 1.0f), 1.0f, 0.09f, 0.032f, true, false
    {glm::vec3(2.0f, 2.0f, 2.0f), glm::vec3(0.0f), glm::vec3(0.2f, 0.2f, 0.2f), glm::vec3(0.5f, 0.5f, 0.5f), glm::vec3(1.0f, 1.0f, 1.0f), 1.0f, 0.09f, 0.032f, false, true},
    {glm::vec3(0.0f, 2.0f, 0.0f), glm::vec3(0.0f), glm::vec3(0.2f, 0.2f, 0.2f), glm::vec3(0.5f, 0.5f, 0.5f), glm::vec3(1.0f, 1.0f, 1.0f), 1.0f, 0.09f, 0.032f, false, true}
};
```

12. Collision Detection

Collision detection is implemented to prevent passing through objects and bounds:

```
bool checkCollision(glm::vec3 pos1, glm::vec3 size1, glm::vec3 pos2, glm::vec3 size2) {
    bool collisionX = pos1.x + size1.x >= pos2.x && pos2.x + size2.x >= pos1.x;
    bool collisionY = pos1.y + size1.y >= pos2.y && pos2.y + size2.y >= pos1.y;
    bool collisionZ = pos1.z + size1.z >= pos2.z && pos2.z + size2.z >= pos1.z;
    return collisionX && collisionY && collisionZ;
}
```

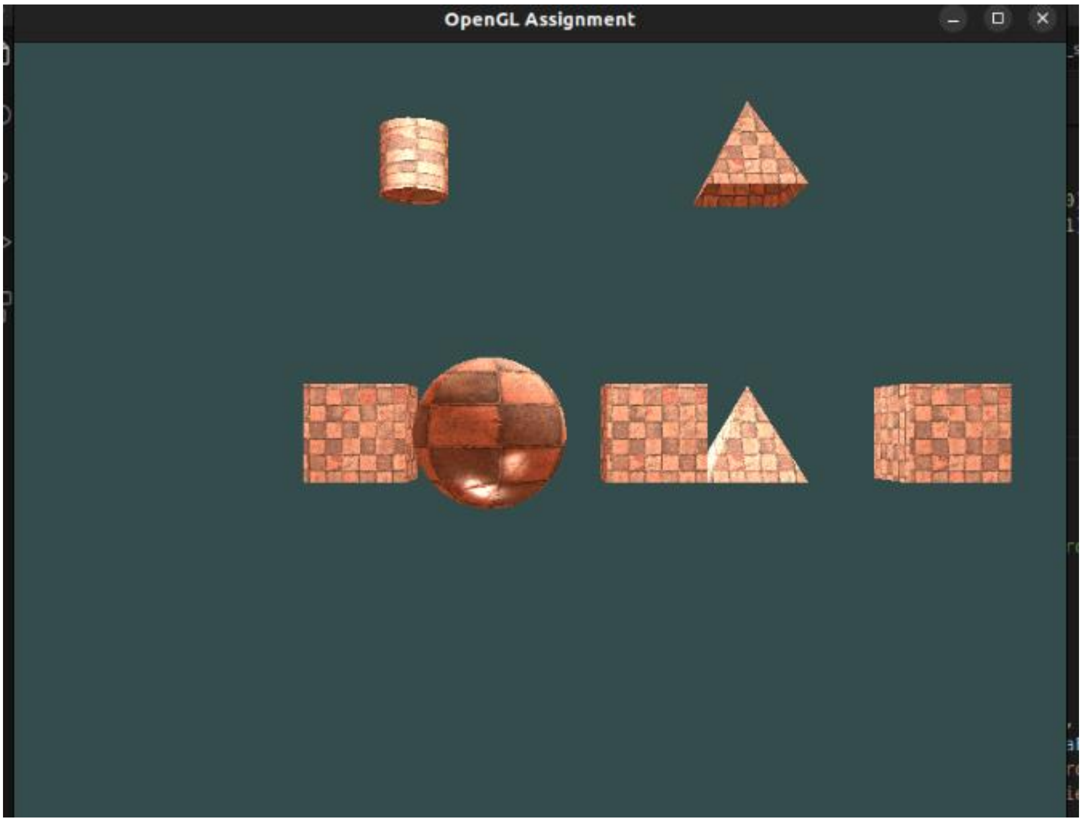
13. Animation and Movement Speed

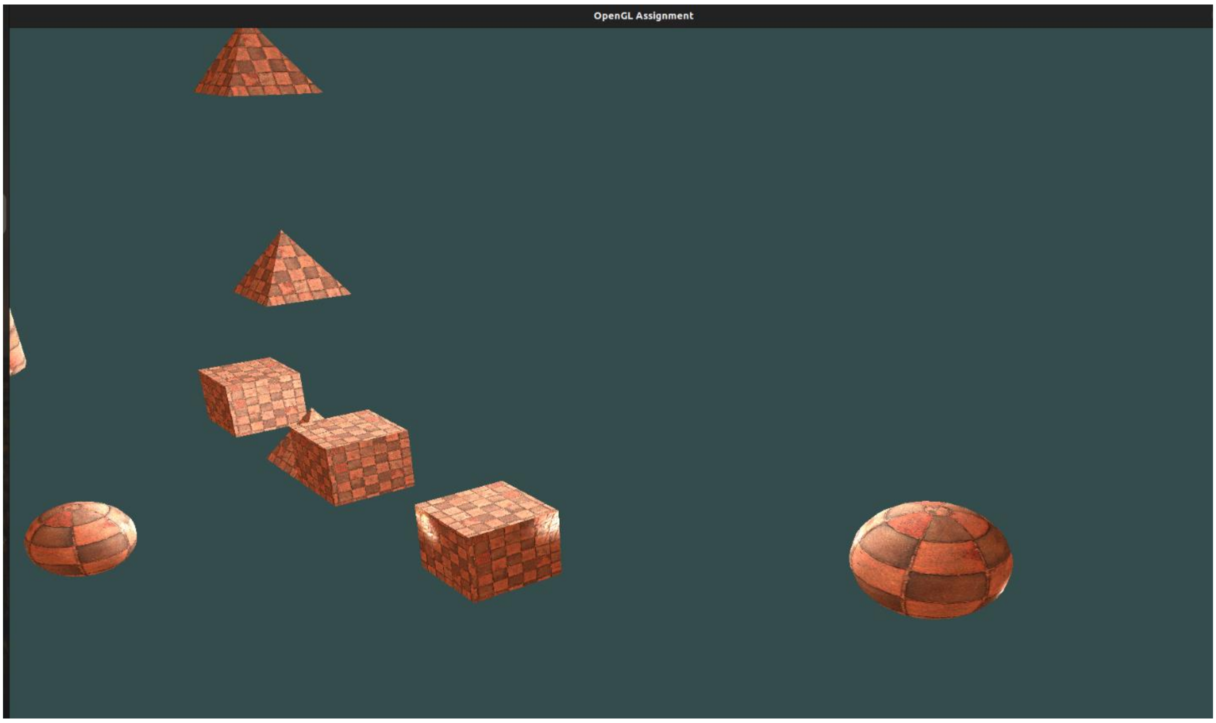
Animation and movement speed are made independent of rendering performance using frame time calculations:

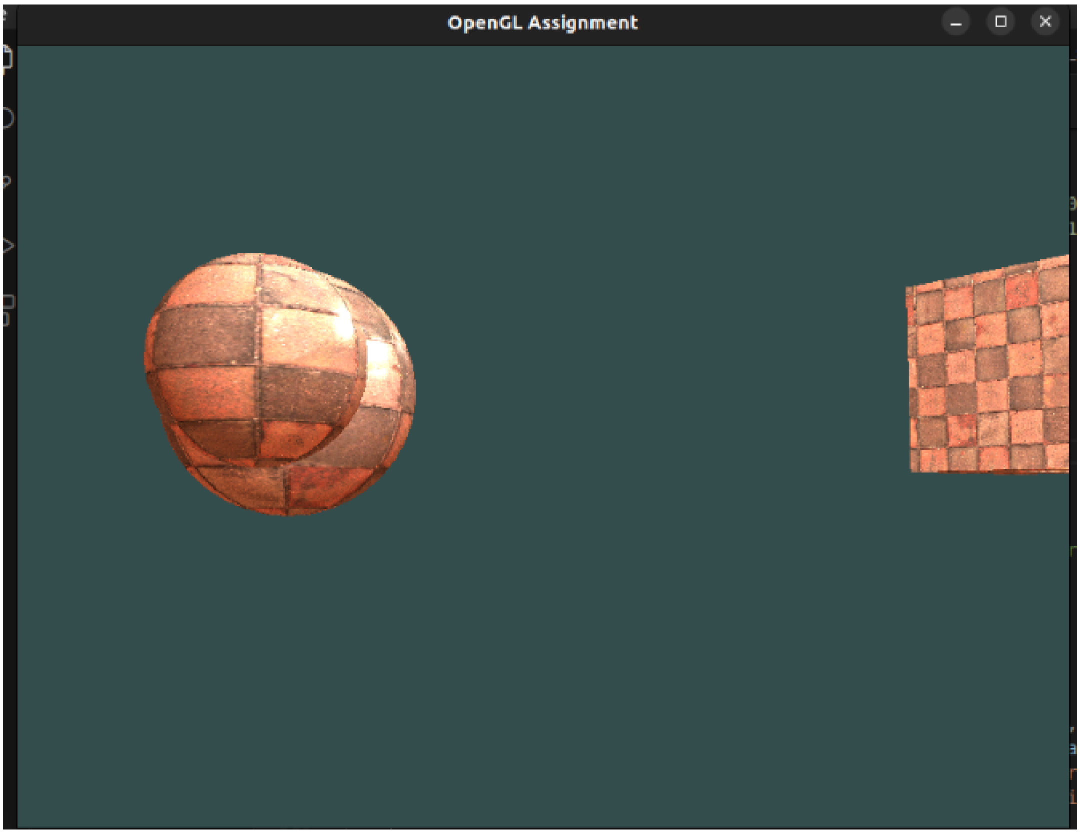
```
// Timing
float deltaTime = 0.0f;
float lastFrame = 0.0f;
```

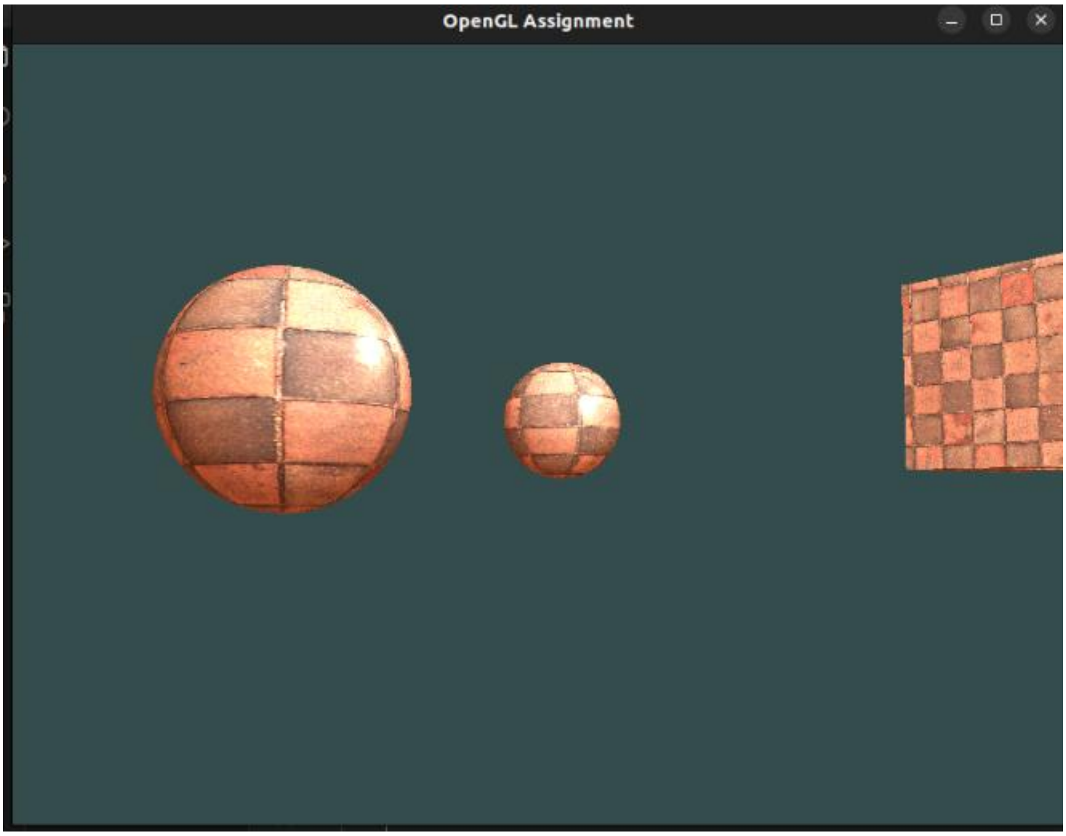
```
// Render loop
while (!glfwWindowShouldClose(window)) {
    // Per-frame time logic
    float currentFrame = glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // Input
    processInput(window);

    // Render
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f); // Change the background color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Use our shader program
    glUseProgram(shaderProgram);

    // Set up the view and projection matrices
    glm::mat4 projection = glm::perspective(glm::radians(fov), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
    glm::mat4 view = glm::lookAt(cameraPos, cameraPos + cameraFront, cameraUp);
    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
    glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "view"), 1, GL_FALSE, glm::value_ptr(view));

    // Bind textures
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture1);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texture2);

    // Render the scene
    renderScene(shaderProgram, VAOs, texture1, texture2);

    // Swap buffers and poll IO events
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Example Screenshoot






OpenGL Assignment


OpenGL Assignment

Conclusion

This OpenGL application meets all the minimum requirements specified in the assignment. The detailed implementation ensures that the 3D world is interactive and visually appealing, with proper lighting, textures, and collision detection. The application runs at a consistent frame rate, providing a smooth user experience.