OpenGL project on the topic: implementation of ray tracing and path tracing technology

BEIJING INSTITUTE OF TECHNOLOGY

GLEB PIMENOV

1820243077

Contents

Overview

The code sets up an OpenGL context using GLFW in Python, and it implements two different rendering techniques: ray tracing and path tracing. It uses vertex and fragment shaders written in GLSL to handle the rendering logic on the GPU. The program also includes features to measure FPS, switch between rendering techniques at intervals, and display the current technique and FPS in the window title.

Detailed Breakdown

1. **GLFW Initialization and Window Creation**:

```
if not glfw.init():
    return

window = glfw.create_window(800, 800, "Ray Tracing vs Path
Tracing", None, None)
if not window:
    glfw.terminate()
    return

glfw.make_context_current(window)
```

This initializes GLFW and creates a window with an OpenGL context. If initialization fails, the program exits.

2. **Vertex Data Setup**:

```
vertices = np.array([
    -1.0, -1.0, 0.0,
    1.0, -1.0, 0.0,
    1.0, 1.0, 0.0,
    -1.0, 1.0, 0.0,
], dtype=np.float32)

indices = np.array([
    0, 1, 2,
    2, 3, 0,
], dtype=np.uint32)
```

This sets up the vertex data for a full-screen quad. The vertices define the corners of the quad, and the indices define the two triangles that make up the quad.

3. **Vertex Array and Buffer Objects**:

```
vao = glGenVertexArrays(1)
glBindVertexArray(vao)

vbo = glGenBuffers(1)
glBindBuffer(GL_ARRAY_BUFFER, vbo)
glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices,
GL_STATIC_DRAW)

ebo = glGenBuffers(1)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
GL_STATIC_DRAW)

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
vertices.itemsize, ctypes.c_void_p(0))
glEnableVertexAttribArray(0)
```

This section generates and binds a Vertex Array Object (VAO) and Vertex Buffer Objects (VBO, EBO). It sets up the vertex attribute pointers to ensure the vertex data is correctly interpreted by the vertex shader.

4. **Shader Programs**:

```
ray_tracing_shader = create_shader_program(vertex_shader,
ray_tracing_fragment_shader)
path_tracing_shader = create_shader_program(vertex_shader,
path_tracing_fragment_shader)
```

This compiles and links the vertex and fragment shaders into shader programs. The create_shader_program function compiles the shaders from the source strings and links them into a program that can be used for rendering.

5. **Render Loop**:

```
start_time = time.time()
fps_counter = 0
fps_time = time.time()
current_shader = ray_tracing_shader
technique_name = "Ray Tracing"
swap_interval = 5  # Swap every 5 seconds

sphere_center = [0.0, 0.0, 0.0]

while not glfw.window_should_close(window):
    glfw.poll_events()
    impl.process_inputs()

    glClear(GL_COLOR_BUFFER_BIT)

    imgui.new_frame()

    imgui.begin("Sphere Position")
    changed, sphere_center[0] = imgui.slider_float("X",
```

```
sphere_center[0], -2.0, 2.0)
    changed, sphere_center[1] = imgui.slider_float("Y",
sphere_center[1], -2.0, 2.0)
    changed, sphere_center[2] = imgui.slider_float("Z",
sphere_center[2], -2.0, 2.0)
    imgui.end()

    imgui.render()
    impl.render(imgui.get_draw_data())

    glUseProgram(current_shader)
    sphere_center_loc = glGetUniformLocation(current_shader,
"sphereCenter")
    glUniform3f(sphere_center_loc, *sphere_center)

    glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT,
None)

    glfw.swap_buffers(window)

    fps_counter += 1
    current_time = time.time()

    if current_time - fps_time >= 1.0:
        fps = fps_counter / (current_time - fps_time)
        fps_time = current_time
        fps_counter = 0
        glfw.set_window_title(window, f"{technique_name} - FPS:
{fps:.2f}")

    if current_time - start_time >= swap_interval:
        start_time = current_time
        if current_shader == ray_tracing_shader:
            current_shader = path_tracing_shader
            technique_name = "Path Tracing"
```

```
        else:
            current_shader = ray_tracing_shader
            technique_name = "Ray Tracing"
```

- The main render loop polls for events, clears the screen, and draws the full-screen quad using the current shader program.
- It updates the FPS counter and window title every second.
- It swaps the shader program between ray tracing and path tracing every swap_interval seconds.

6. **GLFW Termination**:

```
glDeleteVertexArrays(1, vao)
glDeleteBuffers(1, vbo)
glDeleteBuffers(1, ebo)

impl.shutdown()
glfw.terminate()
```

This cleans up the OpenGL resources and terminates GLFW when the window is closed.

Why Use GLSL Code in Strings?

GLSL (OpenGL Shading Language) is a C-like language used to write shaders that run on the GPU. In this program, the vertex and fragment shaders are written in GLSL and embedded as strings in the Python script. The reasons for this approach are:

1. **Integration with Python**:
   - By embedding GLSL code as strings, the shaders can be easily compiled and linked into OpenGL shader programs within the Python script.
2. **Portability**:
   - Keeping the GLSL code in strings within the Python script ensures that the entire program is self-contained and portable. You don't need external shader files.
3. **Flexibility**:
   - It allows for dynamic modifications of the shader code from within the Python script if needed.

Vertex Shader (vertex_shader):

This vertex shader is quite simple. It takes the vertex positions as input and passes them to the fragment shader. The fragPos variable is used to carry the interpolated vertex positions to the fragment shader.

Fragment Shaders:

1. **Ray Tracing Fragment Shader (ray_tracing_fragment_shader)**:

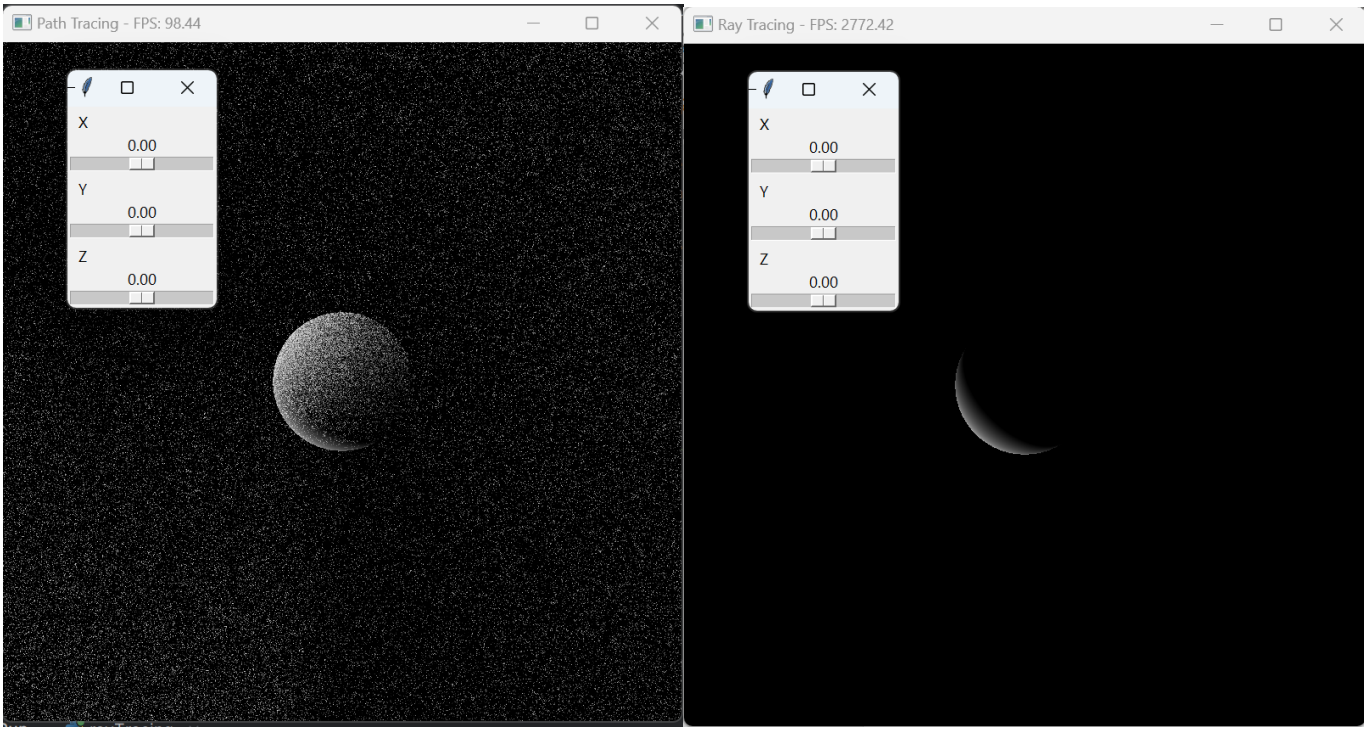
This shader implements basic ray tracing. It calculates the direction of rays, checks for intersections with a sphere, and computes the color based on diffuse shading from a directional light.

2. **Path Tracing Fragment Shader (path_tracing_fragment_shader)**:

This shader extends the ray tracing shader to perform path tracing. It samples multiple rays, computes their contributions, and averages the results to achieve a more realistic rendering. Path tracing involves shooting multiple rays to simulate global illumination effects.

Visual comparison

(Left – path tracing, right – ray tracing)



Summary

The code uses PyOpenGL to interact with OpenGL, GLFW for window management, and GLSL for writing shaders. The shaders are responsible for implementing the ray tracing and path tracing algorithms. The program demonstrates these techniques by rendering a scene and switching between the two methods every few seconds, displaying the FPS and current technique in the window title.

Appendix to the course work

rayTracingVsPathTracing.py

```
import glfw
from OpenGL.GL import *
from OpenGL.GL.shaders import compileProgram, compileShader
import numpy as np
import time
import threading
import tkinter as tk

from vertexShader import vertex_shader
from rayTracingFragmentShader import ray_tracing_fragment_shader
from pathTracingFragmentShader import path_tracing_fragment_shader

def create_shader_program(vertex_src, fragment_src):
    vertex_shader = compileShader(vertex_src, GL_VERTEX_SHADER)
    fragment_shader = compileShader(fragment_src, GL_FRAGMENT_SHADER)
    shader = compileProgram(vertex_shader, fragment_shader)
    return shader


def start_tkinter_app(sphere_center):
    def update_position(event):
        sphere_center[0] = x_slider.get()
        sphere_center[1] = y_slider.get()
        sphere_center[2] = z_slider.get()
```

```
    root = tk.Tk()
    root.title("Sphere Position")

    x_slider = tk.Scale(root, from_=-2.0, to=2.0, resolution=0.01,
orient='horizontal', label='X',
                        command=update_position)
    x_slider.pack(fill='x')
    y_slider = tk.Scale(root, from_=-2.0, to=2.0, resolution=0.01,
orient='horizontal', label='Y',
                        command=update_position)
    y_slider.pack(fill='x')
    z_slider = tk.Scale(root, from_=-2.0, to=2.0, resolution=0.01,
orient='horizontal', label='Z',
                        command=update_position)
    z_slider.pack(fill='x')

    root.mainloop()


def main():
    if not glfw.init():
        return

    window = glfw.create_window(800, 800, "Ray Tracing vs Path
Tracing", None, None)
    if not window:
        glfw.terminate()
        return

    glfw.make_context_current(window)

    vertices = np.array([
        -1.0, -1.0, 0.0,
        1.0, -1.0, 0.0,
        1.0, 1.0, 0.0,
```

```
        -1.0, 1.0, 0.0,
    ], dtype=np.float32)

    indices = np.array([
        0, 1, 2,
        2, 3, 0,
    ], dtype=np.uint32)

    vao = glGenVertexArrays(1)
    glBindVertexArray(vao)

    vbo = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, vbo)
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices,
GL_STATIC_DRAW)

    ebo = glGenBuffers(1)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.nbytes, indices,
GL_STATIC_DRAW)

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
vertices.itemsize, ctypes.c_void_p(0))
    glEnableVertexAttribArray(0)

    ray_tracing_shader = create_shader_program(vertex_shader,
ray_tracing_fragment_shader)
    path_tracing_shader = create_shader_program(vertex_shader,
path_tracing_fragment_shader)

    start_time = time.time()
    fps_counter = 0
    fps_time = time.time()
    current_shader = ray_tracing_shader
    technique_name = "Ray Tracing"
```

```
    swap_interval = 5  # Swap every 5 seconds

    sphere_center = [0.0, 0.0, 0.0]

    # Start the Tkinter GUI in a separate thread
    threading.Thread(target=start_tkinter_app, args=(sphere_center,),
daemon=True).start()

    while not glfw.window_should_close(window):
        glfw.poll_events()

        glClear(GL_COLOR_BUFFER_BIT)

        glUseProgram(current_shader)
        sphere_center_loc = glGetUniformLocation(current_shader,
"sphereCenter")
        glUniform3f(sphere_center_loc, *sphere_center)

        glDrawElements(GL_TRIANGLES, len(indices), GL_UNSIGNED_INT,
None)

        glfw.swap_buffers(window)

        fps_counter += 1
        current_time = time.time()

        if current_time - fps_time >= 1.0:
            fps = fps_counter / (current_time - fps_time)
            fps_time = current_time
            fps_counter = 0
            glfw.set_window_title(window, f"{technique_name} - FPS:
{fps:.2f}")

        if current_time - start_time >= swap_interval:
            start_time = current_time
```

```
            if current_shader == ray_tracing_shader:
                current_shader = path_tracing_shader
                technique_name = "Path Tracing"
            else:
                current_shader = ray_tracing_shader
                technique_name = "Ray Tracing"

    glDeleteVertexArrays(1, vao)
    glDeleteBuffers(1, vbo)
    glDeleteBuffers(1, ebo)

    glfw.terminate()


if __name__ == "__main__":
    main()
```

vertexShader.py

```
vertex_shader = """
#version 330 core
layout(location = 0) in vec3 aPos;
out vec3 fragPos;
void main()
{
    fragPos = aPos;
    gl_Position = vec4(aPos, 1.0);
}
"""
```

rayTracingFragmentShader.py

```
ray_tracing_fragment_shader = """
#version 330 core
out vec4 FragColor;
in vec3 fragPos;
uniform vec3 sphereCenter;
```

```
vec3 rayDirection(vec2 uv, vec3 ro, vec3 rd)
{
    vec3 forward = normalize(rd);
    vec3 right = normalize(cross(forward, vec3(0.0, 1.0, 0.0)));
    vec3 up = cross(right, forward);
    vec3 direction = normalize(forward + uv.x * right + uv.y * up);
    return direction;
}

bool intersectSphere(vec3 ro, vec3 rd, vec3 sphereCenter, float
sphereRadius, out float t)
{
    vec3 oc = ro - sphereCenter;
    float a = dot(rd, rd);
    float b = 2.0 * dot(oc, rd);
    float c = dot(oc, oc) - sphereRadius * sphereRadius;
    float discriminant = b * b - 4.0 * a * c;
    if (discriminant < 0.0) {
        t = -1.0;
        return false;
    } else {
        t = (-b - sqrt(discriminant)) / (2.0 * a);
        return true;
    }
}

vec3 calculateNormal(vec3 p, vec3 sphereCenter)
{
    return normalize(p - sphereCenter);
}

void main()
{
    vec3 ro = vec3(0.0, 0.0, 5.0);  // Ray origin
    vec3 rd = rayDirection(fragPos.xy, ro, vec3(0.0, 0.0, -1.0));  //
```

```
Ray direction

    float sphereRadius = 1.0;
    float t;
    if (intersectSphere(ro, rd, sphereCenter, sphereRadius, t)) {
        vec3 hitPoint = ro + t * rd;
        vec3 normal = calculateNormal(hitPoint, sphereCenter);
        vec3 lightDir = normalize(vec3(-1.0, -1.0, -1.0));
        float diff = max(dot(normal, lightDir), 0.0);
        FragColor = vec4(diff, diff, diff, 1.0);
    } else {
        FragColor = vec4(0.0, 0.0, 0.0, 1.0);
    }
}
"""
```

pathTracingFragmentShader.py

```
path_tracing_fragment_shader = """
#version 330 core
out vec4 FragColor;
in vec3 fragPos;
uniform vec3 sphereCenter;

vec3 rayDirection(vec2 uv, vec3 ro, vec3 rd)
{
    vec3 forward = normalize(rd);
    vec3 right = normalize(cross(forward, vec3(0.0, 1.0, 0.0)));
    vec3 up = cross(right, forward);
    vec3 direction = normalize(forward + uv.x * right + uv.y * up);
    return direction;
}

bool intersectSphere(vec3 ro, vec3 rd, vec3 sphereCenter, float
sphereRadius, out float t)
{
```

```
    vec3 oc = ro - sphereCenter;
    float a = dot(rd, rd);
    float b = 2.0 * dot(oc, rd);
    float c = dot(oc, oc) - sphereRadius * sphereRadius;
    float discriminant = b * b - 4.0 * a * c;
    if (discriminant < 0.0) {
        t = -1.0;
        return false;
    } else {
        t = (-b - sqrt(discriminant)) / (2.0 * a);
        return true;
    }
}

vec3 calculateNormal(vec3 p, vec3 sphereCenter)
{
    return normalize(p - sphereCenter);
}

float rand(vec2 co)
{
    return fract(sin(dot(co.xy, vec2(12.9898, 78.233))) * 43758.5453);
}

void main()
{
    vec3 ro = vec3(0.0, 0.0, 5.0);  // Ray origin
    vec3 rd = rayDirection(fragPos.xy, ro, vec3(0.0, 0.0, -1.0));  //
Ray direction

    float sphereRadius = 1.0;
    float t;
    vec3 color = vec3(0.0);
    int samples = 50;
```

```
    for (int i = 0; i < samples; ++i)
    {
        if (intersectSphere(ro, rd, sphereCenter, sphereRadius, t)) {
            vec3 hitPoint = ro + t * rd;
            vec3 normal = calculateNormal(hitPoint, sphereCenter);
            vec3 lightDir = normalize(vec3(-1.0, -1.0, -1.0));
            float diff = max(dot(normal, lightDir), 0.0);
            color += diff;
        }
        ro += rd * t;
        rd = rayDirection(vec2(rand(ro.xy), rand(ro.yz)), ro, rd);
    }

    FragColor = vec4(color / float(samples), 1.0);
}
"""
```