Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа №8

по курсу «Компьютерная графика»

«Шейдеры OpenGL»

Студент группы ИУ9-42Б Волохов А. В.

Преподаватель Цалкович П. А.

Москва 2024

1 Условие

Задача: Переписать на шейдерах одну из лабораторных: 2, 3 или 6. Была выбрана лаболраторная работа 6.

2 Основная теория

OpenGL и шейдеры: OpenGL — это мощная библиотека для работы с 2D и 3D графикой, которая предоставляет API для взаимодействия с графическим процессором (GPU) для создания высокоэффективных графических приложений. Одним из ключевых элементов OpenGL являются шейдеры — небольшие программы, которые выполняются на GPU и позволяют разработчикам управлять процессом рендеринга на низком уровне.

Шейдеры в OpenGL делятся на несколько типов, но основные из них — это вершинные и фрагментные шейдеры. Вершинные шейдеры обрабатывают вершины, определяя их положение в пространстве, а фрагментные шейдеры определяют цвет каждого пикселя. Шейдеры пишутся на языке GLSL (OpenGL Shading Language), который предоставляет возможности для выполнения математических операций, необходимых для графических вычислений.

В этой лаборатной работе использовались два типа шейдеров: вершинный и фрагментный. Вершинный шейдер преобразует координаты вершин модели из локальной системы координат в экранные координаты, а также передает необходимые данные во фрагментный шейдер, такие как нормали и текстурные координаты. Фрагментный шейдер выполняет освещение с использованием модели фонового, диффузного и зеркального освещения, а также применяет текстуру к поверхности объекта.

Для создания шейдерной программы в OpenGL необходимо скомпилировать исходные коды шейдеров и связать их в единую программу, которая затем используется в процессе рендеринга. В программе, описанной ниже, это выполняется функциями $\mathrm{compile\_shader}$ и $\mathrm{create\_shader\_program}$.

Для текстурирования объекта была использована текстура, загружаемая с помощью библиотеки PIL и применяемая к объекту через шейдеры.

3 Практическая реализация

Код представлен в Листинге 1.

Листинг 1 - lab7.py

```
import math

import glfw
import numpy as np
from OpenGL.GL import *
from math import cos, sin

from PIL import Image

alpha = 0
beta = 0
size = 0.5
fill = True

torus_position = [0.0, 0.0, 0.0]
torus_velocity = [0.001, 0.002, 0.005]


def main():
    if not glfw.init():
        return
    window = glfw.create_window(640, 640, "LAB 8", None, None)
    if not window:
        glfw.terminate()
        return
    glfw.make_context_current(window)
    glfw.set_key_callback(window, key_callback)
    glfw.set_scroll_callback(window, scroll_callback)

    glEnable(GL_DEPTH_TEST)
    glDepthFunc(GL_LESS)

    setup_lighting()
    texture_id = load_texture("texture.jpg")

    vertex_shader_source = """
        attribute vec3 aVert;
        varying vec3 n;
        varying vec3 v;
        varying vec2 uv;
```

```
        void main()
        {
            uv = gl_MultiTexCoord0.xy;
            v = vec3(gl_ModelViewMatrix * vec4(aVert, 1.0));
            n = normalize(gl_NormalMatrix * gl_Normal);
            gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;
            gl_Position = gl_ModelViewProjectionMatrix * vec4(aVert, 1.0);
        }
    """

    fragment_shader_source = """
        varying vec3 n;
        varying vec3 v;
        uniform sampler2D tex;
        void main ()
        {
            vec3 L = normalize(gl_LightSource[0].position.xyz - v);
            vec3 E = normalize(-v);
            vec3 R = normalize(-reflect(L,n));

            vec4 Iamb = gl_FrontLightProduct[0].ambient;
            vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 0.0)
                ;
            Idiff = clamp(Idiff, 0.0, 1.0);

            vec4 Ispec = gl_LightSource[0].specular
                        * pow(max(dot(R,E),0.0),0.7);
            Ispec = clamp(Ispec, 0.0, 1.0);

            vec4 texColor = texture2D(tex, gl_TexCoord[0].st);
            gl_FragColor = (Idiff + Iamb + Ispec) * texColor;
        }
    """

    shader_program = create_shader_program(vertex_shader_source,
        fragment_shader_source)
    glUseProgram(shader_program)

    while not glfw.window_should_close(window):
        display(window, texture_id, shader_program)
    glfw.destroy_window(window)
    glfw.terminate()


def setup_lighting():
    glEnable(GL_LIGHTING)
```

```
    glEnable(GL_LIGHT0)

    light_pos = [10.0, 10.0, 10.0, 1.0]
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos)

    light_diffuse = [1.0, 1.0, 1.0, 1.0]
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse)
    light_specular = [1.0, 1.0, 1.0, 1.0]
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular)

    material_diffuse = [0.8, 0.8, 0.8, 1.0]
    glMaterialfv(GL_FRONT, GL_DIFFUSE, material_diffuse)
    material_specular = [1.0, 1.0, 1.0, 1.0]
    glMaterialfv(GL_FRONT, GL_SPECULAR, material_specular)
    material_shininess = [100.0]
    glMaterialfv(GL_FRONT, GL_SHININESS, material_shininess)


def load_texture(filename):
    img = Image.open(filename)
    img_data = np.array(list(img.getdata()), np.uint8)
    texture_id = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, texture_id)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR)
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR)
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, img.width, img.height, 0, GL_RGB,
        GL_UNSIGNED_BYTE, img_data)
    return texture_id


def compile_shader(source, shader_type):
    shader = glCreateShader(shader_type)
    glShaderSource(shader, source)
    glCompileShader(shader)

    if glGetShaderiv(shader, GL_COMPILE_STATUS) != GL_TRUE:
        raise RuntimeError(glGetShaderInfoLog(shader))

    return shader


def create_shader_program(vertex_source, fragment_source):
    program = glCreateProgram()
    vertex_shader = compile_shader(vertex_source, GL_VERTEX_SHADER)
    fragment_shader = compile_shader(fragment_source, GL_FRAGMENT_SHADER)
```

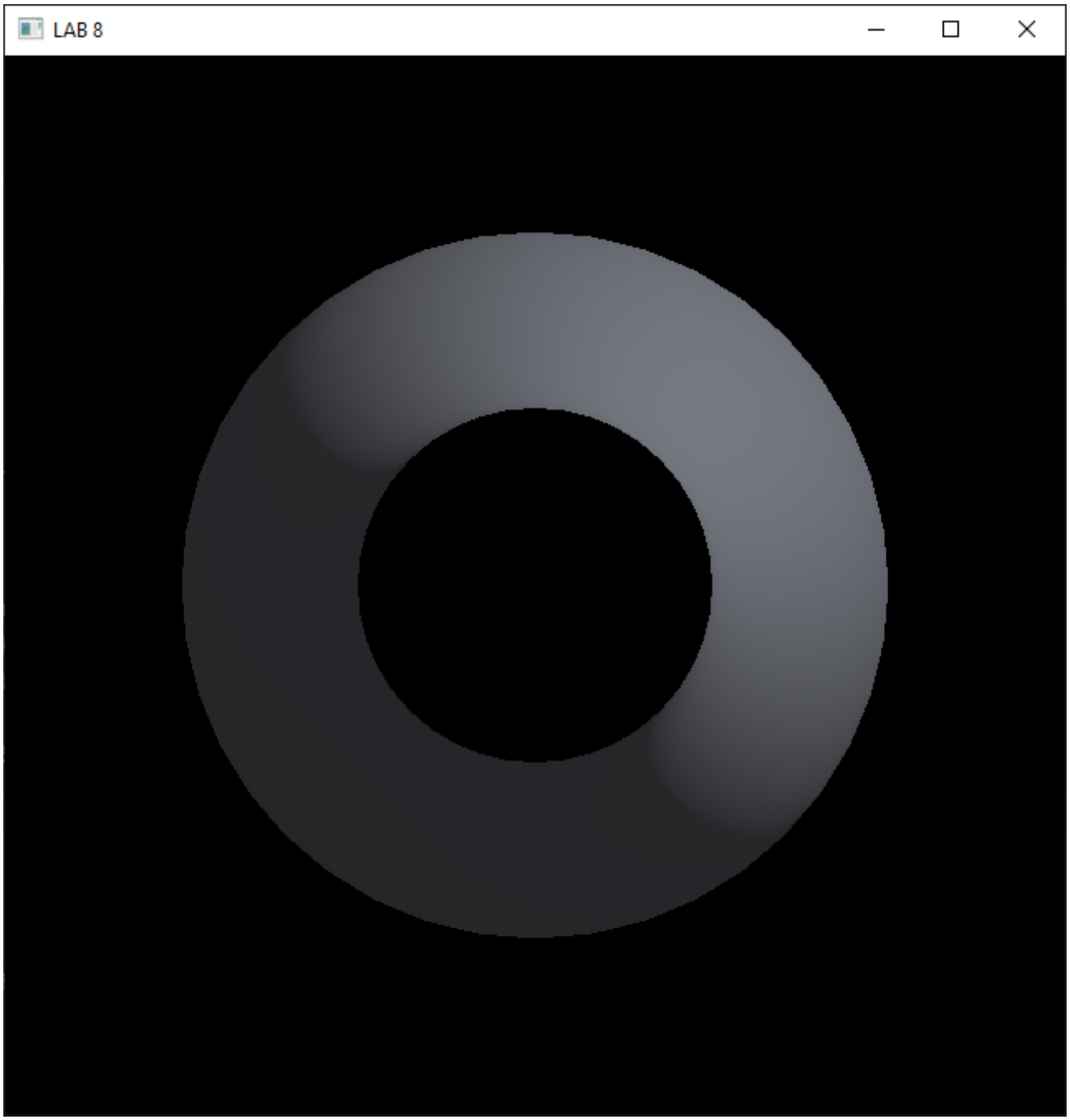
```
    glAttachShader(program, vertex_shader)
    glAttachShader(program, fragment_shader)
    glLinkProgram(program)

    if glGetProgramiv(program, GL_LINK_STATUS) != GL_TRUE:
        raise RuntimeError(glGetProgramInfoLog(program))

    glDeleteShader(vertex_shader)
    glDeleteShader(fragment_shader)

    return program


def display(window, texture_id, shader_program):
    global alpha
    global beta
    global size
    global torus_position
    global torus_velocity

    glLoadIdentity()
    glClear(GL_COLOR_BUFFER_BIT)
    glClear(GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_PROJECTION)

    alpha_rad = np.radians(alpha)
    beta_rad = np.radians(beta)

    rotate_y = np.array([
        [cos(alpha_rad), 0, sin(alpha_rad), 0],
        [0, 1, 0, 0],
        [-sin(alpha_rad), 0, cos(alpha_rad), 0],
        [0, 0, 0, 1]
    ], dtype=np.float32)

    rotate_x = np.array([
        [1, 0, 0, 0],
        [0, cos(beta_rad), -sin(beta_rad), 0],
        [0, sin(beta_rad), cos(beta_rad), 0],
        [0, 0, 0, 1]
    ], dtype=np.float32)

    projection = np.identity(4, dtype=np.float32)
    model = np.dot(rotate_x, rotate_y)
    view = np.identity(4, dtype=np.float32)
```

```
    R = size
    r = size / 3

    for i in range(3):
        torus_position[i] += torus_velocity[i]

    for i in range(3):
        if torus_position[i] + size > 1.0 or torus_position[i] - size < -1.0:
            torus_velocity[i] *= -1.0

    model = np.dot(model, np.array([
        [1, 0, 0, torus_position[0]],
        [0, 1, 0, torus_position[1]],
        [0, 0, 1, torus_position[2]],
        [0, 0, 0, 1]
        ], dtype=np.float32))

    model_loc = glGetUniformLocation(shader_program, "model")
    view_loc = glGetUniformLocation(shader_program, "view")
    projection_loc = glGetUniformLocation(shader_program, "projection")

    glUniformMatrix4fv(model_loc, 1, GL_TRUE, model)
    glUniformMatrix4fv(view_loc, 1, GL_TRUE, view)
    glUniformMatrix4fv(projection_loc, 1, GL_TRUE, projection)

    torus(R, r, 40, 25, shader_program, texture_id)

    glfw.swap_buffers(window)
    glfw.poll_events()

def torus(R, r, N, n, shader_program, texture_id):
    glEnable(GL_TEXTURE_2D)
    glBindTexture(GL_TEXTURE_2D, texture_id)
    vertices = []
    tex_coords = []

    for i in range(N):
        for j in range(n):
            theta = (2 * math.pi / N) * i
            phi = (2 * math.pi / n) * j
            theta_next = (2 * math.pi / N) * (i + 1)
            phi_next = (2 * math.pi / n) * (j + 1)

            x0 = (R + r * cos(phi)) * cos(theta)
            y0 = (R + r * cos(phi)) * sin(theta)
            z0 = r * sin(phi)
```

```
            vertices.extend([x0, y0, z0])
            tex_coords.extend([j / n, i / N])

            x1 = (R + r * cos(phi)) * cos(theta_next)
            y1 = (R + r * cos(phi)) * sin(theta_next)
            z1 = r * sin(phi)
            vertices.extend([x1, y1, z1])
            tex_coords.extend([(j + 1) / n, i / N])

            x2 = (R + r * cos(phi_next)) * cos(theta_next)
            y2 = (R + r * cos(phi_next)) * sin(theta_next)
            z2 = r * sin(phi_next)
            vertices.extend([x2, y2, z2])
            tex_coords.extend([(j + 1) / n, (i + 1) / N])

            x3 = (R + r * cos(phi_next)) * cos(theta)
            y3 = (R + r * cos(phi_next)) * sin(theta)
            z3 = r * sin(phi_next)
            vertices.extend([x3, y3, z3])
            tex_coords.extend([j / n, (i + 1) / N])

    vertices = np.array(vertices, dtype=np.float32)
    tex_coords = np.array(tex_coords, dtype=np.float32)

    vao = glGenVertexArrays(1)
    vbo = glGenBuffers(2)

    glBindVertexArray(vao)

    glBindBuffer(GL_ARRAY_BUFFER, vbo[0])
    glBufferData(GL_ARRAY_BUFFER, vertices.nbytes, vertices, GL_STATIC_DRAW)
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, None)
    glEnableVertexAttribArray(0)

    glBindBuffer(GL_ARRAY_BUFFER, vbo[1])
    glBufferData(GL_ARRAY_BUFFER, tex_coords.nbytes, tex_coords,
        GL_STATIC_DRAW)
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, None)
    glEnableVertexAttribArray(1)

    glBindVertexArray(vao)
    glDrawArrays(GL_QUADS, 0, len(vertices) // 3)

    glDisable(GL_TEXTURE_2D)
    glBindVertexArray(0)
```

```
def key_callback(window, key, scancode, action, mods):
    global alpha
    global beta

    if action == glfw.PRESS or action == glfw.REPEAT:
        if key == glfw.KEY_RIGHT:
            alpha += 3
        elif key == glfw.KEY_LEFT:
            alpha -= 3
        elif key == glfw.KEY_UP:
            beta -= 3
        elif key == glfw.KEY_DOWN:
            beta += 3
        elif key == glfw.KEY_F:
            global fill
            fill = not fill
            if fill:
                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)
            else:
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)
        elif key == glfw.KEY_L and action == glfw.PRESS:
            light_enabled = glIsEnabled(GL_LIGHT0)
            if light_enabled:
                glDisable(GL_LIGHT0)
            else:
                glEnable(GL_LIGHT0)

def scroll_callback(window, xoffset, yoffset):
    global size

    if xoffset > 0:
        size -= yoffset / 10
    else:
        size += yoffset / 10


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

В результате работы программы получился следующий вывод:



4 Заключение

Реализация шейдеров в OpenGL позволяет существенно улучшить качество графики и производительность приложения. В данном проекте были успешно применены вершинные и фрагментные шейдеры для реализации анимации текстурированного тора с освещением. Шейдеры предоставляют гибкие и мощные инструменты для управления графическим процессом, что делает их неотъемлемой частью современной компьютерной графики.