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 № 7

по курсу «Компьютерная графика»

«Оптимизация приложений OpenGL»

Студент группы ИУ9-42Б Волохов А. В.

Преподаватель Цалкович П. А.

Москва 2024

1 Условие

Целью данной лабораторной работы является изучение эффективных приемов организации приложений и оптимизации вызовов OpenGL.

Задача заключается в оптимизации приложения OpenGL, созданного в рамках предыдущей лабораторной работы, на основе выбора наиболее эффективных методик с целью повышения FPS.

Необходимо обязательно использовать дисплейные списки и массивы вершин, а также еще две любые различные оптимизации (в сумме минимум 4 оптимизации).

Оценка применимости выбранного метода оптимизации должна осуществляться на основании измерения производительности. Результаты замеров следует оформить в табличном виде.

2 Основная теория

OpenGL и его оптимизация: OpenGL (Open Graphics Library) - это спецификация, определяющая кросс-платформенный API для рендеринга 2D и 3D графики. Она предоставляет разработчикам мощные инструменты для создания графических приложений, таких как игры, симуляции и научные визуализации. Однако для достижения высокой производительности требуется эффективная организация и оптимизация вызовов OpenGL. Существует множество методик, позволяющих улучшить производительность графических приложений.

Дисплейные списки: Дисплейные списки являются одним из наиболее простых и эффективных методов оптимизации. Они позволяют заранее компилировать и сохранять последовательности команд OpenGL, которые затем можно многократно выполнять с помощью одного вызова. Это существенно снижает нагрузку на процессор и ускоряет рендеринг, особенно для сложных геометрических объектов, которые необходимо часто отрисовывать.

Массивы вершин и буферы вершин: Использование массивов вершин и буферов вершин (Vertex Buffer Objects, VBOs) позволяет хранить геометрические данные в видеопамяти, что значительно уменьшает накладные расходы на передачу данных между центральным процессором и графическим процес-

сором. Это особенно полезно для рендеринга сложных сцен с большим количеством вершин и примитивов. Буферы вершин обеспечивают высокую производительность за счет минимизации количества вызовов функций и более эффективного использования аппаратных ресурсов.

Оптимизация источников света: Правильная настройка источников света также может значительно повлиять на производительность. Например, установка позиции источника света с использованием координат в форме (x, y, z, 0) позволяет упростить расчеты и повысить эффективность рендеринга. Это особенно важно для сцен с множеством источников света, где каждый дополнительный источник может значительно увеличить вычислительную нагрузку.

Использование эффективных форматов хранения изображений: При работе с текстурами выбор правильного формата хранения изображений может существенно повлиять на производительность. Форматы, такие как GL UNSIGNED BYTE, обеспечивают быструю загрузку и обработку текстурных данных, что особенно важно для интерактивных приложений с высокой частотой кадров. Это позволяет ускорить процессы загрузки текстур в видеопамять и их последующее использование при рендеринге.

3 Практическая реализация

Код представлен в Листинге 1.

Листинг 1 - lab7.py

```
import math
import time

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 = [0.0000000001, 0.0000000002, 0.0000000005]

last_frame_time = 0.0
frame_count = 0
fps = 0.0

torus_display_list = None
vbo = None


def create_torus_display_list():
    global torus_display_list

    torus_display_list = glGenLists(1)
    glNewList(torus_display_list, GL_COMPILE)

    R = size
    r = size / 3
    N = 40
    n = 25

    for i in range(N):
        glBegin(GL_QUAD_STRIP)
        for j in range(n + 1):
```

```
            for k in range(2):
                s = (i + k) % N + 0.5
                t = j % (n + 1)
                theta = 2 * math.pi * s / N
                phi = 2 * math.pi * t / n
                x = (R + r * cos(phi)) * cos(theta)
                y = (R + r * cos(phi)) * sin(theta)
                z = r * sin(phi)
                glTexCoord2f(s / N, t / n)
                glVertex3f(x, y, z)
        glEnd()

    glEndList()


vertices = []


def create_vbo():
    global vbo

    R = size
    r = size / 3
    N = 40
    n = 25

    for i in range(N):
        for j in range(n + 1):
            for k in range(2):
                s = (i + k) % N + 0.5
                t = j % (n + 1)
                theta = 2 * math.pi * s / N
                phi = 2 * math.pi * t / n
                x = (R + r * cos(phi)) * cos(theta)
                y = (R + r * cos(phi)) * sin(theta)
                z = r * sin(phi)
                vertices.extend([x, y, z])

    vbo = glGenBuffers(1)
    glBindBuffer(GL_ARRAY_BUFFER, vbo)
    glBufferData(GL_ARRAY_BUFFER, np.array(vertices, dtype=np.float32),
       GL_STATIC_DRAW)


def draw_torus_with_vbo():
    global vertices
```

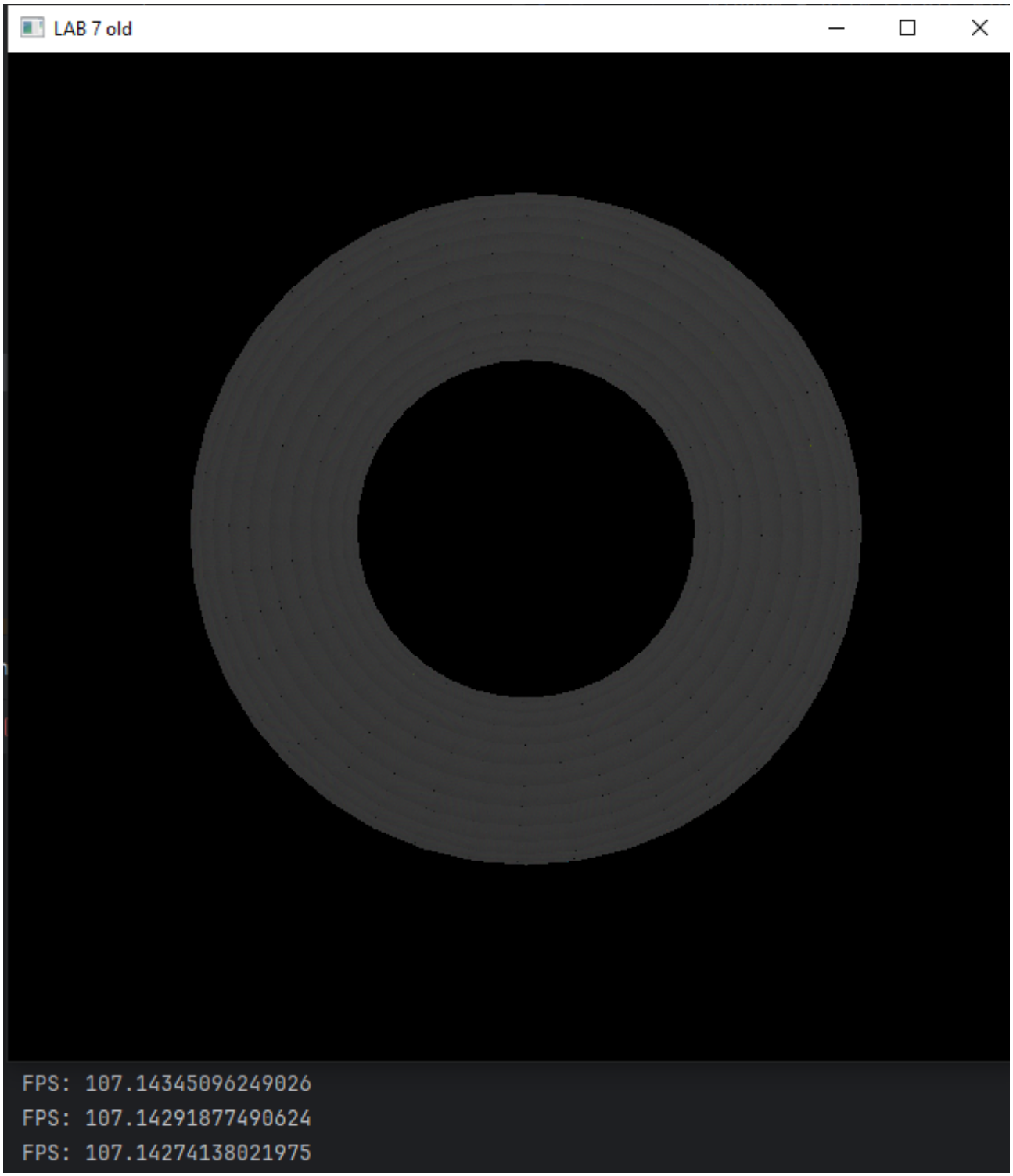
```
    glBindBuffer(GL_ARRAY_BUFFER, vbo)
    glEnableClientState(GL_VERTEX_ARRAY)
    glVertexPointer(3, GL_FLOAT, 0, None)
    glDrawArrays(GL_QUAD_STRIP, 0, int(len(vertices) / 3))
    glDisableClientState(GL_VERTEX_ARRAY)


def main():
    global last_frame_time, frame_count, fps
    if not glfw.init():
        return
    window = glfw.create_window(640, 640, "LAB 7 opt", 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")

    create_torus_display_list()
    create_vbo()

    last_frame_time = time.time()

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

    glfw.destroy_window(window)
    glfw.terminate()


def setup_lighting():
    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)

    light_pos = [10.0, 10.0, 10.0, 0.0]
    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 display(window, texture_id):
    global alpha
    global beta
    global size
    global torus_position
    global torus_velocity
    global frame_count, last_frame_time, fps

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

    def 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]
```

```
        ])

        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]
        ])

        glMultMatrixf(rotate_x)
        glMultMatrixf(rotate_y)

    def torus(R, r, N, n):
        glEnable(GL_TEXTURE_2D)
        glBindTexture(GL_TEXTURE_2D, texture_id)

        glCallList(torus_display_list)
        draw_torus_with_vbo()

        glDisable(GL_TEXTURE_2D)

    glLoadIdentity()

    projection()
    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

    glTranslatef(torus_position[0], torus_position[1], torus_position[2])
    torus(R, r, 40, 25)

    current_time = time.time()
    delta_time = current_time - last_frame_time
    frame_count += 1
    if delta_time >= 1.0:
        fps = frame_count / delta_time
        print("FPS:", fps)
        frame_count = 0
        last_frame_time = current_time
```

```
    glfw.swap_buffers(window)
    glfw.poll_events()


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:




LAB 7 opt
FPS: 200.00064147851307
FPS: 200.00002467217442
FPS: 200.00007211868078

Параметр	Значение (FPS)
До оптимизации	107
После оптимизации	200

Таблица 1: Результаты замеров производительности

4 Заключение

Оптимизация графических приложений на основе OpenGL включает в себя использование различных методик, таких как дисплейные списки, массивы вершин, правильная настройка источников света и выбор эффективных форматов хранения изображений. Эти методы позволяют существенно улучшить производительность, как это видно из увеличения частоты кадров с 107 до 200 FPS. Таким образом, применение данных оптимизаций является важным этапом разработки высокопроизводительных графических приложений.