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 «Алгоритмы компьютерной графики»

Студент группы ИУ9-41Б Горбунов А. Д.

Преподаватель Цалкович П. А.

Москва 2024

1 Задача

• оптимизация приложения OpenGL, созданного в рамках предыдущей лабораторной работы, на основе выбора наиболее эффективных методик. (см. Баяковский Ю.М., Игнатенко А.В. Начальный курс OpenGL.- М.: «Планета Знаний», 2007.- 221с. – Глава 9).

• обязательно использовать дисплейные списки и массивы вершин и еще 2 любые различные оптимизации(в сумме минимум 4 оптимизации).

• оценка применимости выбранного метода оптимизации приложения OpenGL должна осуществляться на основании измерения производительности.

• результаты замеров оформить в табличном виде.

2 Теория

Оптимизация вызовов OpenGL

• стратегии оптимизации:

– передача данных в OpenGL;

– управление обработкой вершин в графическом конвейере;

– управление растеризацией;

– управление текстурированием;

– управление очисткой буферов;

– минимизация числа изменений состояния OpenGL.

Дисплейные списки (display lists)

• если несколько раз производится обращение к одной и той же группе команд, то их можно объединить в дисплейный список, и вызывать его при необходимости;

• дисплейные списки в оптимальном, скомпилированном виде хранятся в памяти сервера, что позволяет рисовать примитивы в такой форме максимально быстро;

• в то же время большие объемы данных занимают много памяти, что в свою очередь влечет падение производительности (большие объемы (больше нескольких десятков тысяч примитивов) лучше рисовать с помощью массивов вершин; • работа с дисплейными списками:

1. создание дисплейного списка: glNewList() / glEndList().

• идентификация списка: целое положительное число

• режим обработки списка:

– GL COMPILE: команды записываются в список без выполнения

– GL COMPILE AND EXECUTE: команды сначала выполняются, а затем записываются в список

2. вызов списка(ов): glCallList() / glCallLists()

3. удаление дисплейного списка: glDeleteList()

Массивы вершин

• задание массивов:

– массив координат вершин:

glVertexPointer (GLint size, GLenum type, GLsizei stride, void* ptr)

– массив координат нормалей:

glNormalPointer (GLenum type, GLsizei stride, void* ptr)

– массив цветов:

glColorPointer (GLint size, GLenum type, GLsizei stride, void* ptr)

– массив текстурных координат:

glTexCoordPointer (GLint size, GLenum type, GLsizei stride, void* ptr)

– одновременно несколько массивов:

glInterleavedArrays(GLenum format, GLsizei stride, const GLvoid * pointer);

• определение используемых массивов:

glEnableClientState(GLenum array) / glDisableClientState(GLenum array)

• использование сформированных массивов для отрисовки:

– отдельного элемента:

void glArrayElement(GLint i)

– набора примитивов по данным из массивов:

void glDrawArrays(GLenum mode, GLint first, GLsizei count);

– набора примитивов по данным из массивов (по индексам):

void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices);

3 Код решения

Программа без оптимизации

Файл main.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <cmath>
#include <chrono>
#include <ctime>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#include "iostream"
#include <fstream>
#include <sstream>
using std::cos, std::sin, std::string;
using namespace std::string_literals;

int mode = 1;
int lightMode = 1;
int degreeMode = 1;
int timeMode = 0;
float degree_y = 0.0;
float degree_x = 0.0;
float move_y = 0.0;
float move_x = 0.0;
float osnov_x = 0.1;
float osnov_y = 0.0;
float flying_speed = 0;
float V = 3.14 * pow(10,-4);
float acl = pow(10,-4);
int width = 1000;
int height = 1000;
```

```
GLuint textureID;
void key_callback(GLFWwindow *window, int key, int scancode, int action, int mods
{
    if (action == GLFW_PRESS || action == GLFW_REPEAT)
    {
        if (key == GLFW_KEY_ESCAPE)
        {
            glfwSetWindowShouldClose(window, GL_TRUE);
        }
        else if (key == GLFW_KEY_UP)
        {
            degree_y += 0.2;
        }
        else if (key == GLFW_KEY_DOWN)
        {
            degree_y -= 0.2;
        }
        else if (key == GLFW_KEY_LEFT)
        {
            degree_x += 0.2;
        }
        else if (key == GLFW_KEY_RIGHT)
        {
            degree_x -= 0.2;
        }
        else if (key == GLFW_KEY_D)
        {
            move_x += 0.2;
        }
        else if (key == GLFW_KEY_A)
        {
            move_x -= 0.2;
        }
```

```
else if (key == GLFW_KEY_W)
{
    move_y += 0.2;
}
else if (key == GLFW_KEY_S)
{
    move_y -= 0.2;
}
else if (key == GLFW_KEY_L)
{
    osnov_x += 0.1;
}
else if (key == GLFW_KEY_K)
{
    osnov_x -= 0.1;
}
else if (key == GLFW_KEY_I)
{
    osnov_y += 0.1;
}
else if (key == GLFW_KEY_O)
{
    osnov_y -= 0.1;
}
else if (key == GLFW_KEY_SPACE)
{
    mode = (mode + 1) % 2;
    if (mode == 0)
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    else
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}
else if (key == GLFW_KEY_1)
```

```
        {
            lightMode = (lightMode + 1) % 2;
            glDisable(GL_LIGHT0);
        }
        else if (key == GLFW_KEY_2)
        {
            degreeMode = (degreeMode + 1) % 2;
        }
        else if (key == GLFW_KEY_3)
        {
            timeMode = (timeMode + 1) % 2;
        }
    }
}
void light()
{
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(1, 1, 1);
    GLfloat material_diffuse[] = {0.75, 0.75, 0.75, 0.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_diffuse);
    GLfloat light2_diffuse[] = {1, 1, 0};
    GLfloat light2_position[] = {0, 0, 0, 1.0};
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light2_position);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.2);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.4);
    glPopMatrix();
}
void texture()
{
    int width_1, height_1, channels;
    unsigned char* image = stbi_load("./../texture.bmp", &width_1, &height_1, &c
    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_N
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NE
    if (image){
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width_1, height_1, 0, GL_
    }
   stbi_image_free(image);
}
void move_object()
{
    flying_speed -= V;
    V += acl;
    if(flying_speed < -2.2 or flying_speed > 2.2)
        V = -V;
}
void render()
{
    glBegin(GL_QUAD_STRIP);
    glColor3f(0.4f, 0.4f, 1.0f);
    for (int i = 0; i <= 360; i += 1)
    {
        float angle = i * M_PI / 180 ;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }
    glEnd();
    glBegin(GL_POLYGON);
    glNormal3f(1, 1, -1);
    glColor3f(1.0f, 0.3f, 0.3f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
    }
    glEnd();
    glBegin(GL_POLYGON);
    glNormal3f(1, 1, 1);
    glColor3f(0.5f, 0.7f, 0.7f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }
    glEnd();
}
void display(GLFWwindow* window)
{
    glClearColor (0.3, 0.3, 0.3, 0.0);
    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glPushMatrix();
    glTranslatef(0.0f + move_x, 0.0f + move_y + flying_speed, 0.0f);
    glRotatef(degree_y * 50.f, 1.f, 0.f, 0.f);
    glRotatef(degree_x * 50.f, 0.f, 1.f, 0.f);
    render();
    glPopMatrix();
    GLfloat spec[] = {1, 1, 1, 1};
    GLfloat emiss[] = {0, 0, 0, 1};
    GLfloat shin = 50;
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,  spec);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, &shin);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION,  emiss);
}
GLuint compileShader(GLuint type, const std::string& source)
{
    GLuint id = glCreateShader(type);
    const char* src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);
    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE)
    {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char* message = (char*)alloca(length * sizeof(char));
        glGetShaderInfoLog(id, length, &length, message);
        std::cout << "Failed to compile " << (type == GL_VERTEX_SHADER ? "
        std::cout << message << std::endl;
        glDeleteShader(id);
        return 0;
    }
    return id;
}
int main()
{
    auto start = std::chrono::high_resolution_clock::now();
    if (!glfwInit()) {
        return -1;
    }
    GLFWwindow* window = glfwCreateWindow(width, height, "Lab 7", NULL, NU
    if (!window) {
        glfwTerminate();
        return -1;
    }
    glViewport(0, 0, width, height);
    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);
    GLenum err = glewInit();
    if (err != GLEW_OK) {
        std::cerr << "Failed to initialize GLEW: " << glewGetErrorString(err) << st
        return -1;
    }
    string vertexShaderSource =
"attribute vec3 aVert; "s+
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec2 uv;"s+
"varying vec4 vertexColor; "s+
"void main() {"s+
"    uv = gl_MultiTexCoord0.xy; "s+
"    v = vec3(gl_ModelViewMatrix * gl_Vertex); "s+
"    n = normalize(gl_NormalMatrix * gl_Normal); "s+
"    gl_TexCoord[0] = gl_TextureMatrix[0]  * gl_MultiTexCoord0; "s+
"    gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.x, gl_Vertex.y,
"    vec4 vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);"s+
"}"s;

    string fragmentShaderSource =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 1.0); "s+
"    Idiff = clamp(Idiff, 2.0, 0.6);     "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R,E),0.0),0.7);"s+
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;
   string fragmentShaderSource_bad =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 0.0); "s+
"    Idiff = clamp(Idiff, 0.0, 1.0);     "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R, E), 0.0), gl_FrontMa
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;
```

```
GLuint vertex = compileShader(GL_VERTEX_SHADER, vertexShaderSource);
GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShader
//GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShad
int program = glCreateProgram();
glAttachShader(program, vertex);
glAttachShader(program, fragment);
glLinkProgram(program);
glScalef(0.25,0.25, 0.25);
glEnable(GL_LIGHTING);
glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
glEnable(GL_NORMALIZE);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
texture();
glUseProgram(program);
for(int i = 0;  i < 300 && !glfwWindowShouldClose(window); i++)
{
    display(window);

    if(degreeMode)
    {
        degree_x += 0.01;
    }
    if(timeMode)
    {
        move_object();
    }
    if(lightMode)
    {
        light();
    }
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

```
    glfwTerminate();
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float> duration = end - start;
    std::cout << "Время выполнения: " << duration.count() << " секунд" << std

    return 0;
}
```

Оптимизация с помощью дисплейного списка

Файл main_display_list.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <cmath>
#include <chrono>
#include <ctime>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#include "iostream"
#include <fstream>
#include <sstream>
using std::cos, std::sin, std::string;
using namespace std::string_literals;

int mode = 1;
int lightMode = 1;
int degreeMode = 1;
int timeMode = 0;

float degree_y = 0.0;
float degree_x = 0.0;
float move_y = 0.0;
float move_x = 0.0;
float osnov_x = 0.1;
float osnov_y = 0.0;

float flying_speed = 0;
float V = 3.14 * pow(10,-4);
float acl = pow(10,-4);

int width = 1000;
int height = 1000;

GLuint prism_display_list = 0;
GLuint textureID;

void drow_figur();

void update_display_list()
{
    if (prism_display_list != 0)
    {
        std::cout << 3;
        glDeleteLists(prism_display_list, 1);
    }
    std::cout << 4 << std::endl;
    drow_figur();
}

void render_display_list()
{
    if(prism_display_list == 0)
    {
        std::cout << 1;
        drow_figur();
    }
    std::cout << 2 << std::endl;
    glCallList(prism_display_list);
}

void key_callback(GLFWwindow *window, int key, int scancode, int action, int mods
{
    if (action == GLFW_PRESS || action == GLFW_REPEAT)
    {
        if (key == GLFW_KEY_ESCAPE)
        {
            glfwSetWindowShouldClose(window, GL_TRUE);
        }
        else if (key == GLFW_KEY_UP)
        {
            degree_y += 0.2;
            update_display_list();
        }
        else if (key == GLFW_KEY_DOWN)
        {
            degree_y -= 0.2;
            update_display_list();
        }
        else if (key == GLFW_KEY_LEFT)
        {
            degree_x += 0.2;
            update_display_list();
        }
        else if (key == GLFW_KEY_RIGHT)
        {
            degree_x -= 0.2;
            update_display_list();
        }
        else if (key == GLFW_KEY_D)
        {
```

```
    move_x += 0.2;
}
else if (key == GLFW_KEY_A)
{
    move_x -= 0.2;
}
else if (key == GLFW_KEY_W)
{
    move_y += 0.2;
}
else if (key == GLFW_KEY_S)
{
    move_y -= 0.2;
}
else if (key == GLFW_KEY_L)
{
    osnov_x += 0.1;
    update_display_list();
}
else if (key == GLFW_KEY_K)
{
    osnov_x -= 0.1;
    update_display_list();
}
else if (key == GLFW_KEY_I)
{
    osnov_y += 0.1;
    update_display_list();
}
else if (key == GLFW_KEY_O)
{
    osnov_y -= 0.1;
    update_display_list();
```

```
        }
        else if (key == GLFW_KEY_SPACE)
        {
            mode = (mode + 1) % 2;
            if (mode == 0)
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            else
                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
            update_display_list();

        }
        else if (key == GLFW_KEY_1)
        {
            lightMode = (lightMode + 1) % 2;
            glDisable(GL_LIGHT0);
        }
        else if (key == GLFW_KEY_2)
        {
            degreeMode = (degreeMode + 1) % 2;
        }
        else if (key == GLFW_KEY_3)
        {
            timeMode = (timeMode + 1) % 2;
        }
    }
}

void light()
{
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(1, 1, 1);
    GLfloat material_diffuse[] = {0.75, 0.75, 0.75, 0.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_diffuse);

    GLfloat light2_diffuse[] = {1, 1, 0};
    GLfloat light2_position[] = {0, 0, 0, 1.0};
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light2_position);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.2);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.4);
    glPopMatrix();
}

void texture()
{
    int width_1, height_1, channels;
    unsigned char* image = stbi_load("./../texture.bmp", &width_1, &height_1, &c

    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_N
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NE
    if (image){
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width_1, height_1, 0, GL_
    }
   stbi_image_free(image);
}
```

```
void move_object()
{
    flying_speed -= V;
    V += acl;
    if(flying_speed < -2.2 or flying_speed > 2.2)
        V = -V;
}

void drow_figur()
{
    prism_display_list = glGenLists(1);
    glNewList(prism_display_list, GL_COMPILE);

    glBegin(GL_QUAD_STRIP);
    glColor3f(0.4f, 0.4f, 1.0f);
    for (int i = 0; i <= 360; i += 1)
    {
        float angle = i * M_PI / 180 ;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }
    glEnd();

    glBegin(GL_POLYGON);
    glNormal3f(1, 1, -1);
    glColor3f(1.0f, 0.3f, 0.3f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
    }
    glEnd();

    glBegin(GL_POLYGON);
    glNormal3f(1, 1, 1);
    glColor3f(0.5f, 0.7f, 0.7f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }

    glEnd();
    glEndList();
}

void display(GLFWwindow* window)
{
    glClearColor (0.3, 0.3, 0.3, 0.0);
    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBindTexture(GL_TEXTURE_2D, textureID);
    glPushMatrix();

    glTranslatef(0.0f + move_x, 0.0f + move_y + flying_speed, 0.0f);
    glRotatef(degree_y * 50.f, 1.f, 0.f, 0.f);
    glRotatef(degree_x * 50.f, 0.f, 1.f, 0.f);

    render_display_list();

    glPopMatrix();
    GLfloat spec[] = {1, 1, 1, 1};
    GLfloat emiss[] = {0, 0, 0, 1};
    GLfloat shin = 50;
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,  spec);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, &shin);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION,  emiss);
}

GLuint compileShader(GLuint type, const std::string& source)
{
    GLuint id = glCreateShader(type);
    const char* src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);

    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE)
    {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char* message = (char*)alloca(length * sizeof(char));
        glGetShaderInfoLog(id, length, &length, message);
        std::cout << "Failed to compile " << (type == GL_VERTEX_SHADER ? "
        std::cout << message << std::endl;
        glDeleteShader(id);
        return 0;
    }

    return id;
}
```

```
int main()
{
    auto start = std::chrono::high_resolution_clock::now();

    if (!glfwInit()) {
        return -1;
    }

    GLFWwindow* window = glfwCreateWindow(width, height, "Lab 7_2", NULL, N
    if (!window) {
        glfwTerminate();
        return -1;
    }
    glViewport(0, 0, width, height);

    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);

    GLenum err = glewInit();
    if (err != GLEW_OK) {
        std::cerr << "Failed to initialize GLEW: " << glewGetErrorString(err) << st
        return -1;
    }

    string vertexShaderSource =
"attribute vec3 aVert; "s+
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec2 uv;"s+
"varying vec4 vertexColor; "s+
"void main() {"s+
"    uv = gl_MultiTexCoord0.xy; "s+
"    v = vec3(gl_ModelViewMatrix * gl_Vertex); "s+
"    n = normalize(gl_NormalMatrix * gl_Normal); "s+
"    gl_TexCoord[0] = gl_TextureMatrix[0]  * gl_MultiTexCoord0; "s+
"    gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.x, gl_Vertex.y,
"    vec4 vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);"s+
"}"s;

   string fragmentShaderSource =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 1.0); "s+
"    Idiff = clamp(Idiff, 2.0, 0.6);     "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R,E),0.0),0.7);"s+
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;

   string fragmentShaderSource_bad =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 0.0); "s+
"    Idiff = clamp(Idiff, 0.0, 1.0);      "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R, E), 0.0), gl_FrontMa
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;

   GLuint vertex = compileShader(GL_VERTEX_SHADER, vertexShaderSource);
   GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShader
   //GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShac

   int program = glCreateProgram();
   glAttachShader(program, vertex);
   glAttachShader(program, fragment);
   glLinkProgram(program);

   glScalef(0.25,0.25, 0.25);
   glEnable(GL_LIGHTING);
   glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
   glEnable(GL_NORMALIZE);
   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
   texture();



   glUseProgram(program);
   for(int i = 0; i < 300 && !glfwWindowShouldClose(window); i++)
   {
      display(window);

      if(degreeMode)
```

```
        {
            degree_x += 0.01;
        }
        if(timeMode)
        {
            move_object();
        }
        if(lightMode)
        {
            light();
        }

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();


    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float> duration = end - start;
    std::cout << "Время выполнения: " << duration.count() << " секунд" << std

    return 0;
}
```

Оптимизация с помощью массива вершин

Файл main_vertex_arrays.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <cmath>
#include <chrono>
#include <ctime>
```

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#include "iostream"
#include <fstream>
#include <sstream>
#include <vector>
using std::cos, std::sin, std::string, std::vector;
using namespace std::string_literals;
int mode = 1;
int lightMode = 1;
int degreeMode = 1;
int timeMode = 0;

float degree_y = 0.0;
float degree_x = 0.0;
float move_y = 0.0;
float move_x = 0.0;
float osnov_x = 0.1;
float osnov_y = 0.0;

float flying_speed = 0;
float V = 3.14 * pow(10,-4);
float acl = pow(10,-4);

int width = 1000;
int height = 1000;

GLuint textureID;

GLuint vbo;
GLfloat quadStripVertices[361 * 2 * 3];
GLfloat polygonVertices[361 * 3];
GLuint quadStripVBO, polygonVBO;
```

```
void key_callback(GLFWwindow *window, int key, int scancode, int action, int mods
{
    if (action == GLFW_PRESS || action == GLFW_REPEAT)
    {
        if (key == GLFW_KEY_ESCAPE)
        {
            glfwSetWindowShouldClose(window, GL_TRUE);
        }
        else if (key == GLFW_KEY_UP)
        {
            degree_y += 0.2;
        }
        else if (key == GLFW_KEY_DOWN)
        {
            degree_y -= 0.2;
        }
        else if (key == GLFW_KEY_LEFT)
        {
            degree_x += 0.2;
        }
        else if (key == GLFW_KEY_RIGHT)
        {
            degree_x -= 0.2;
        }
        else if (key == GLFW_KEY_D)
        {
            move_x += 0.2;
        }
        else if (key == GLFW_KEY_A)
        {
            move_x -= 0.2;
        }
```

```
else if (key == GLFW_KEY_W)
{
    move_y += 0.2;
}
else if (key == GLFW_KEY_S)
{
    move_y -= 0.2;
}
else if (key == GLFW_KEY_L)
{
    osnov_x += 0.1;
}
else if (key == GLFW_KEY_K)
{
    osnov_x -= 0.1;
}
else if (key == GLFW_KEY_I)
{
    osnov_y += 0.1;
}
else if (key == GLFW_KEY_O)
{
    osnov_y -= 0.1;
}
else if (key == GLFW_KEY_SPACE)
{
    mode = (mode + 1) % 2;
    if (mode == 0)
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    else
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

}
```

```
        else if (key == GLFW_KEY_1)
        {
            lightMode = (lightMode + 1) % 2;
            glDisable(GL_LIGHT0);
        }
        else if (key == GLFW_KEY_2)
        {
            degreeMode = (degreeMode + 1) % 2;
        }
        else if (key == GLFW_KEY_3)
        {
            timeMode = (timeMode + 1) % 2;
        }
    }
}

void light()
{
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(1, 1, 1);

    GLfloat material_diffuse[] = {0.75, 0.75, 0.75, 0.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_diffuse);

    GLfloat light2_diffuse[] = {1, 1, 0};
    GLfloat light2_position[] = {0, 0, 0, 1.0};
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light2_position);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.2);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.4);
    glPopMatrix();
}

void texture()
{
    int width_1, height_1, channels;
    unsigned char* image = stbi_load("./../texture.bmp", &width_1, &height_1, &c

    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPE
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_N
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NE
    if (image){
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width_1, height_1, 0, GL_
    }
   stbi_image_free(image);
}

void move_object()
{
    flying_speed -= V;
    V += acl;
    if(flying_speed < -2.2 or flying_speed > 2.2)
        V = -V;
}

void render()
{
    // Рисование с использованием буферов вершин
    glBindBuffer(GL_ARRAY_BUFFER, quadStripVBO);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, NULL);
    glDrawArrays(GL_QUAD_STRIP, 0, 361 * 2);

    glBindBuffer(GL_ARRAY_BUFFER, polygonVBO);
    glVertexPointer(3, GL_FLOAT, 0, NULL);
    glDrawArrays(GL_POLYGON, 0, 361);
}


void fillVertices() {
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        // Для GL_QUAD_STRIP
        quadStripVertices[i * 6] = 1 * cos(angle) + osnov_x;
        quadStripVertices[i * 6 + 1] = 0.5 * sin(angle) + osnov_y;
        quadStripVertices[i * 6 + 2] = 0.0;
        quadStripVertices[i * 6 + 3] = 1 * cos(angle);
        quadStripVertices[i * 6 + 4] = 0.5 * sin(angle);
        quadStripVertices[i * 6 + 5] = 1.0;

        // Для первого GL_POLYGON
        polygonVertices[i * 3] = 1 * cos(angle) + osnov_x;
        polygonVertices[i * 3 + 1] = 0.5 * sin(angle) + osnov_y;
        polygonVertices[i * 3 + 2] = 0.0;
    }
}


void initVBO() {
    fillVertices();
    glGenBuffers(1, &quadStripVBO);
    glBindBuffer(GL_ARRAY_BUFFER, quadStripVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(quadStripVertices), quadStripVertic

    glGenBuffers(1, &polygonVBO);
    glBindBuffer(GL_ARRAY_BUFFER, polygonVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(polygonVertices), polygonVertices, G
}

void display(GLFWwindow* window)
{
    glClearColor (0.3, 0.3, 0.3, 0.0);
    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBindTexture(GL_TEXTURE_2D, textureID);
    glPushMatrix();

    glTranslatef(0.0f + move_x, 0.0f + move_y + flying_speed, 0.0f);
    glRotatef(degree_y * 50.f, 1.f, 0.f, 0.f);
    glRotatef(degree_x * 50.f, 0.f, 1.f, 0.f);

    render();

    glPopMatrix();
    GLfloat spec[] = {1, 1, 1, 1};
    GLfloat emiss[] = {0, 0, 0, 1};
    GLfloat shin = 50;
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,  spec);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, &shin);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION,  emiss);
```

```
}

GLuint compileShader(GLuint type, const std::string& source)
{
    GLuint id = glCreateShader(type);
    const char* src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);

    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE)
    {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char* message = (char*)alloca(length * sizeof(char));
        glGetShaderInfoLog(id, length, &length, message);
        std::cout << "Failed to compile " << (type == GL_VERTEX_SHADER ? "
        std::cout << message << std::endl;
        glDeleteShader(id);
        return 0;
    }

    return id;
}

int main()
{
    auto start = std::chrono::high_resolution_clock::now();

    if (!glfwInit()) {
        return -1;
    }
    GLFWwindow* window = glfwCreateWindow(width, height, "Lab 7", NULL, NU
    if (!window) {
        glfwTerminate();
        return -1;
    }
    glViewport(0, 0, width, height);

    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);

    GLenum err = glewInit();
    if (err != GLEW_OK) {
        std::cerr << "Failed to initialize GLEW: " << glewGetErrorString(err) << std
        return -1;
    }

    string vertexShaderSource =
"attribute vec3 aVert; "s+
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec2 uv;"s+
"varying vec4 vertexColor; "s+
"void main() {"s+
"    uv = gl_MultiTexCoord0.xy; "s+
"    v = vec3(gl_ModelViewMatrix * gl_Vertex); "s+
"    n = normalize(gl_NormalMatrix * gl_Normal); "s+
"    gl_TexCoord[0] = gl_TextureMatrix[0]  * gl_MultiTexCoord0; "s+
"    gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.x, gl_Vertex.y,
"    vec4 vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);"s+
"}"s;

    string fragmentShaderSource =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 1.0); "s+
"    Idiff = clamp(Idiff, 2.0, 0.6);      "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R,E),0.0),0.7);"s+
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;

   string fragmentShaderSource_bad =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 0.0); "s+
"    Idiff = clamp(Idiff, 0.0, 1.0);      "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R, E), 0.0), gl_FrontMa
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;

   GLuint vertex = compileShader(GL_VERTEX_SHADER, vertexShaderSource);
   GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShader
   //GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShad

   int program = glCreateProgram();
   glAttachShader(program, vertex);
   glAttachShader(program, fragment);
   glLinkProgram(program);

   glScalef(0.25,0.25, 0.25);
   glEnable(GL_LIGHTING);
   glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
   glEnable(GL_NORMALIZE);
   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

   initVBO();
   texture();

   glUseProgram(program);

   for(int i = 0;  i < 300 && !glfwWindowShouldClose(window); i++)
   {
       display(window);

       if(degreeMode)
       {
           degree_x += 0.01;
       }
       if(timeMode)
       {
           move_object();
       }
       if(lightMode)
       {
           light();
       }

       glfwSwapBuffers(window);

       glfwPollEvents();

   }
   glDeleteBuffers(1, &quadStripVBO);
   glDeleteBuffers(1, &polygonVBO);
   glfwTerminate();


   auto end = std::chrono::high_resolution_clock::now();
   std::chrono::duration<float> duration = end - start;
   std::cout << "Время выполнения: " << duration.count() << " секунд" << std

   return 0;
}
```

Оптимизация occlusion query

Файл main_occlusion_query.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <cmath>
#include <chrono>
#include <ctime>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#include "iostream"
```

```
#include <fstream>
#include <sstream>
using std::cos, std::sin, std::string;
using namespace std::string_literals;
int mode = 1;
int lightMode = 1;
int degreeMode = 1;
int timeMode = 0;

float degree_y = 0.0;
float degree_x = 0.0;
float move_y = 0.0;
float move_x = 0.0;
float osnov_x = 0.1;
float osnov_y = 0.0;

float flying_speed = 0;
float V = 3.14 * pow(10,-4);
float acl = pow(10,-4);

int width = 1000;
int height = 1000;

GLuint textureID;

void key_callback(GLFWwindow *window, int key, int scancode, int action, int mods
{
    if (action == GLFW_PRESS || action == GLFW_REPEAT)
    {
        if (key == GLFW_KEY_ESCAPE)
        {
            glfwSetWindowShouldClose(window, GL_TRUE);
        }
```

```
else if (key == GLFW_KEY_UP)
{
    degree_y += 0.2;
}
else if (key == GLFW_KEY_DOWN)
{
    degree_y -= 0.2;
}
else if (key == GLFW_KEY_LEFT)
{
    degree_x += 0.2;
}
else if (key == GLFW_KEY_RIGHT)
{
    degree_x -= 0.2;
}
else if (key == GLFW_KEY_D)
{
    move_x += 0.2;
}
else if (key == GLFW_KEY_A)
{
    move_x -= 0.2;
}
else if (key == GLFW_KEY_W)
{
    move_y += 0.2;
}
else if (key == GLFW_KEY_S)
{
    move_y -= 0.2;
}
else if (key == GLFW_KEY_L)
```

```
{
    osnov_x += 0.1;
}
else if (key == GLFW_KEY_K)
{
    osnov_x -= 0.1;
}
else if (key == GLFW_KEY_I)
{
    osnov_y += 0.1;
}
else if (key == GLFW_KEY_O)
{
    osnov_y -= 0.1;
}
else if (key == GLFW_KEY_SPACE)
{
    mode = (mode + 1) % 2;
    if (mode == 0)
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    else
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

}
else if (key == GLFW_KEY_1)
{
    lightMode = (lightMode + 1) % 2;
    glDisable(GL_LIGHT0);
}
else if (key == GLFW_KEY_2)
{
    degreeMode = (degreeMode + 1) % 2;
}
```

```
        else if (key == GLFW_KEY_3)
        {
            timeMode = (timeMode + 1) % 2;
        }
    }
}

void light()
{
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(1, 1, 1);

    GLfloat material_diffuse[] = {0.75, 0.75, 0.75, 0.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_diffuse);

    GLfloat light2_diffuse[] = {1, 1, 0};
    GLfloat light2_position[] = {0, 0, 0, 1.0};
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light2_position);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.2);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.4);
    glPopMatrix();
}

void texture()
{
    int width_1, height_1, channels;
    unsigned char* image = stbi_load("./../texture.bmp", &width_1, &height_1, &c

    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NI
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NE
    if (image){
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width_1, height_1, 0, GL_
    }
   stbi_image_free(image);
}

void move_object()
{
    flying_speed -= V;
    V += acl;
    if(flying_speed < -2.2 or flying_speed > 2.2)
        V = -V;
}

void render()
{
    glBegin(GL_QUAD_STRIP);
    glColor3f(0.4f, 0.4f, 1.0f);
    for (int i = 0; i <= 360; i += 1)
    {
        float angle = i * M_PI / 180 ;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }
    glEnd();

    glBegin(GL_POLYGON);
    glNormal3f(1, 1, -1);
    glColor3f(1.0f, 0.3f, 0.3f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
    }
    glEnd();

    glBegin(GL_POLYGON);
    glNormal3f(1, 1, 1);
    glColor3f(0.5f, 0.7f, 0.7f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }

    glEnd();
}

void display(GLFWwindow* window)
{
    glClearColor (0.3, 0.3, 0.3, 0.0);
    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBindTexture(GL_TEXTURE_2D, textureID);
    glPushMatrix();

    glTranslatef(0.0f + move_x, 0.0f + move_y + flying_speed, 0.0f);
    glRotatef(degree_y * 50.f, 1.f, 0.f, 0.f);
    glRotatef(degree_x * 50.f, 0.f, 1.f, 0.f);

    GLuint* query_id = new GLuint[1];

    glGenQueries(1, query_id);

    glBeginQuery(GL_SAMPLES_PASSED, query_id[0]);



    render();

    glEndQuery(GL_SAMPLES_PASSED);
    GLuint* samples_passed = new GLuint[1];
    glGetQueryObjectuiv(query_id[0], GL_QUERY_RESULT, samples_passed);





    glPopMatrix();
    GLfloat spec[] = {1, 1, 1, 1};
    GLfloat emiss[] = {0, 0, 0, 1};
    GLfloat shin = 50;
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,  spec);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, &shin);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION,  emiss);
}

GLuint compileShader(GLuint type, const std::string& source)
```

```
{
    GLuint id = glCreateShader(type);
    const char* src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);

    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE)
    {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char* message = (char*)alloca(length * sizeof(char));
        glGetShaderInfoLog(id, length, &length, message);
        std::cout << "Failed to compile " << (type == GL_VERTEX_SHADER ? "
        std::cout << message << std::endl;
        glDeleteShader(id);
        return 0;
    }

    return id;
}


int main()
{
    auto start = std::chrono::high_resolution_clock::now();

    if (!glfwInit()) {
        return -1;
    }

    GLFWwindow* window = glfwCreateWindow(width, height, "Lab 7", NULL, NU
    if (!window) {
        glfwTerminate();
        return -1;
    }
    glViewport(0, 0, width, height);

    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);

    GLenum err = glewInit();
    if (err != GLEW_OK) {
        std::cerr << "Failed to initialize GLEW: " << glewGetErrorString(err) << std
        return -1;
    }

    string vertexShaderSource =
"attribute vec3 aVert; "s+
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec2 uv;"s+
"varying vec4 vertexColor; "s+
"void main() {"s+
"    uv = gl_MultiTexCoord0.xy; "s+
"    v = vec3(gl_ModelViewMatrix * gl_Vertex); "s+
"    n = normalize(gl_NormalMatrix * gl_Normal); "s+
"    gl_TexCoord[0] = gl_TextureMatrix[0]  * gl_MultiTexCoord0; "s+
"    gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.x, gl_Vertex.y,
"    vec4 vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);"s+
"}"s;

    string fragmentShaderSource =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 1.0); "s+
"    Idiff = clamp(Idiff, 2.0, 0.6);      "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R,E),0.0),0.7);"s+
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;

   string fragmentShaderSource_bad =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 0.0); "s+
"    Idiff = clamp(Idiff, 0.0, 1.0);      "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R, E), 0.0), gl_FrontMa
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;
```

```
GLuint vertex = compileShader(GL_VERTEX_SHADER, vertexShaderSource);
GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShader
//GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShad

int program = glCreateProgram();
glAttachShader(program, vertex);
glAttachShader(program, fragment);
glLinkProgram(program);

glScalef(0.25,0.25, 0.25);
glEnable(GL_LIGHTING);
glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
glEnable(GL_NORMALIZE);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
texture();

glUseProgram(program);

for(int i = 0;  i < 300 && !glfwWindowShouldClose(window); i++)
{
    display(window);

    if(degreeMode)
    {
        degree_x += 0.01;
    }
    if(timeMode)
    {
        move_object();
    }
    if(lightMode)
    {
        light();
        }

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwTerminate();


    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float> duration = end - start;
    std::cout << "Время выполнения: " << duration.count() << " секунд" << std

    return 0;
}
```

Оптимизация шейдеров

Файл main_shaders.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <cmath>
#include <chrono>
#include <ctime>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#include "iostream"
#include <fstream>
#include <sstream>
using std::cos, std::sin, std::string;
using namespace std::string_literals;
int mode = 1;
int lightMode = 1;
int degreeMode = 1;
int timeMode = 0;

float degree_y = 0.0;
float degree_x = 0.0;
float move_y = 0.0;
float move_x = 0.0;
float osnov_x = 0.1;
float osnov_y = 0.0;

float flying_speed = 0;
float V = 3.14 * pow(10,-4);
float acl = pow(10,-4);

int width = 1000;
int height = 1000;

GLuint textureID;

void key_callback(GLFWwindow *window, int key, int scancode, int action, int mods
{
    if (action == GLFW_PRESS || action == GLFW_REPEAT)
    {
        if (key == GLFW_KEY_ESCAPE)
        {
            glfwSetWindowShouldClose(window, GL_TRUE);
        }
        else if (key == GLFW_KEY_UP)
        {
            degree_y += 0.2;
        }
        else if (key == GLFW_KEY_DOWN)
        {
            degree_y -= 0.2;
        }
        else if (key == GLFW_KEY_LEFT)
        {
            degree_x += 0.2;
        }
        else if (key == GLFW_KEY_RIGHT)
        {
            degree_x -= 0.2;
        }
        else if (key == GLFW_KEY_D)
        {
            move_x += 0.2;
        }
        else if (key == GLFW_KEY_A)
        {
            move_x -= 0.2;
        }
        else if (key == GLFW_KEY_W)
        {
            move_y += 0.2;
        }
        else if (key == GLFW_KEY_S)
        {
            move_y -= 0.2;
        }
        else if (key == GLFW_KEY_L)
        {
            osnov_x += 0.1;
        }
        else if (key == GLFW_KEY_K)
        {
            osnov_x -= 0.1;
        }
```

```
        else if (key == GLFW_KEY_I)
        {
            osnov_y += 0.1;
        }
        else if (key == GLFW_KEY_O)
        {
            osnov_y -= 0.1;
        }
        else if (key == GLFW_KEY_SPACE)
        {
            mode = (mode + 1) % 2;
            if (mode == 0)
                glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            else
                glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

        }
        else if (key == GLFW_KEY_1)
        {
            lightMode = (lightMode + 1) % 2;
            glDisable(GL_LIGHT0);
        }
        else if (key == GLFW_KEY_2)
        {
            degreeMode = (degreeMode + 1) % 2;
        }
        else if (key == GLFW_KEY_3)
        {
            timeMode = (timeMode + 1) % 2;
        }
    }
}
```

```
void light()
{
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(1, 1, 1);

    GLfloat material_diffuse[] = {0.75, 0.75, 0.75, 0.0};
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, material_diffuse);

    GLfloat light2_diffuse[] = {1, 1, 0};
    GLfloat light2_position[] = {0, 0, 0, 1.0};
    glEnable(GL_LIGHT0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light2_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light2_position);
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.2);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.4);
    glPopMatrix();
}

void texture()
{
    int width_1, height_1, channels;
    unsigned char* image = stbi_load("./../texture.bmp", &width_1, &height_1, &c

    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);

    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEA
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_N
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NE
    if (image){
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width_1, height_1, 0, GL_
    }
   stbi_image_free(image);
}

void move_object()
{
    flying_speed -= V;
    V += acl;
    if(flying_speed < -2.2 or flying_speed > 2.2)
        V = -V;
}

void render()
{
    glBegin(GL_QUAD_STRIP);
    glColor3f(0.4f, 0.4f, 1.0f);
    for (int i = 0; i <= 360; i += 1)
    {
        float angle = i * M_PI / 180 ;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }
    glEnd();

    glBegin(GL_POLYGON);
    glNormal3f(1, 1, -1);
    glColor3f(1.0f, 0.3f, 0.3f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y);
        glVertex3f(1 * cos(angle) + osnov_x, 0.5 * sin(angle) + osnov_y, 0.0);
    }
    glEnd();

    glBegin(GL_POLYGON);
    glNormal3f(1, 1, 1);
    glColor3f(0.5f, 0.7f, 0.7f);
    for (int i = 0; i <= 360; i++)
    {
        float angle = i * M_PI / 180;
        glTexCoord2f(1 * cos(angle), 0.5 * sin(angle));
        glVertex3f(1 * cos(angle), 0.5 * sin(angle), 1);
    }

    glEnd();
}

void display(GLFWwindow* window)
{
    glClearColor (0.3, 0.3, 0.3, 0.0);
    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glBindTexture(GL_TEXTURE_2D, textureID);
    glPushMatrix();

    glTranslatef(0.0f + move_x, 0.0f + move_y + flying_speed, 0.0f);
    glRotatef(degree_y * 50.f, 1.f, 0.f, 0.f);
    glRotatef(degree_x * 50.f, 0.f, 1.f, 0.f);

    render();
    glPopMatrix();
    GLfloat spec[] = {1, 1, 1, 1};
    GLfloat emiss[] = {0, 0, 0, 1};
    GLfloat shin = 50;
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE)
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,  spec);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, &shin);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION,  emiss);
}

GLuint compileShader(GLuint type, const std::string& source)
{
    GLuint id = glCreateShader(type);
    const char* src = source.c_str();
    glShaderSource(id, 1, &src, nullptr);
    glCompileShader(id);

    int result;
    glGetShaderiv(id, GL_COMPILE_STATUS, &result);
    if (result == GL_FALSE)
    {
        int length;
        glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
        char* message = (char*)alloca(length * sizeof(char));
        glGetShaderInfoLog(id, length, &length, message);
        std::cout << "Failed to compile " << (type == GL_VERTEX_SHADER ? "
        std::cout << message << std::endl;
        glDeleteShader(id);
        return 0;
    }

    return id;
```

```
}



int main()
{
    auto start = std::chrono::high_resolution_clock::now();

    if (!glfwInit()) {
        return -1;
    }

    GLFWwindow* window = glfwCreateWindow(width, height, "Lab 7", NULL, NU
    if (!window) {
        glfwTerminate();
        return -1;
    }
    glViewport(0, 0, width, height);

    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, key_callback);

    GLenum err = glewInit();
    if (err != GLEW_OK) {
        std::cerr << "Failed to initialize GLEW: " << glewGetErrorString(err) << st
        return -1;
    }

    string vertexShaderSource =
"attribute vec3 aVert; "s+
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec2 uv;"s+
"varying vec4 vertexColor; "s+
"void main() {"s+
"    uv = gl_MultiTexCoord0.xy; "s+
"    v = vec3(gl_ModelViewMatrix * gl_Vertex); "s+
"    n = normalize(gl_NormalMatrix * gl_Normal); "s+
"    gl_TexCoord[0] = gl_TextureMatrix[0]  * gl_MultiTexCoord0; "s+
"    gl_Position = gl_ModelViewProjectionMatrix * vec4(gl_Vertex.x, gl_Vertex.y,
"    vec4 vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f);"s+
"}"s;

   string fragmentShaderSource =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 1.0); "s+
"    Idiff = clamp(Idiff, 2.0, 0.6);     "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R,E),0.0),0.7);"s+
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;

   string fragmentShaderSource_bad =
"varying vec3 n; "s+
"varying vec3 v; "s+
"varying vec4 vertexColor;"s+
"uniform sampler2D tex; "s+
"void main () {  "s+
"    vec3 L = normalize(gl_LightSource[0].position.xyz - v); "s+
"    vec3 E = normalize(-v); "s+
"    vec3 R = normalize(-reflect(L,n)); "s+
"    vec4 Iamb = gl_FrontLightProduct[0].ambient; "s+
"    vec4 Idiff = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 0.0); "s+
"    Idiff = clamp(Idiff, 0.0, 1.0);     "s+
"    vec4 Ispec = gl_LightSource[0].specular * pow(max(dot(R, E), 0.0), gl_FrontMa
"    Ispec = clamp(Ispec, 0.0, 1.0); "s+
"    vec4 texColor = texture2D(tex, gl_TexCoord[0].st); "s+
"    gl_FragColor = (Idiff + Iamb + Ispec) * texColor;"s+
"}"s;

   GLuint vertex = compileShader(GL_VERTEX_SHADER, vertexShaderSource);
   GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShader
   //GLuint fragment = compileShader(GL_FRAGMENT_SHADER, fragmentShad

   int program = glCreateProgram();
   glAttachShader(program, vertex);
   glAttachShader(program, fragment);
   glLinkProgram(program);

   glEnableClientState(GL_VERTEX_ARRAY);
   glEnableClientState(GL_NORMAL_ARRAY);
   glEnableClientState(GL_COLOR_ARRAY);

   glEnable(GL_LIGHTING);
   glLightModelf(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
   glEnable(GL_NORMALIZE);
   glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
   texture();

   render();
```

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glScalef(0.25,0.25, 0.25);
glUseProgram(program);

for(int i = 0;  i < 300 && !glfwWindowShouldClose(window); i++)
{
    display(window);

    if(degreeMode)
    {
        degree_x += 0.01;
    }
    if(timeMode)
    {
        move_object();
    }
    if(lightMode)
    {
        light();
    }

    glfwSwapBuffers(window);
    glfwPollEvents();
}

glfwTerminate();


auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<float> duration = end - start;
std::cout << "Время выполнения: " << duration.count() << " секунд" << std
```

```
    return 0;
}
```

4 Таблица оптимизации

Замеры проводились с помощью 10-ти запуков программы подряд и вычисления их среднего времени.

№	Оптимизация	Время
1	Без оптимизации	5,143874
2	Дисплейный список	5,142373
3	Массива вершин	5,143159
4	Occlusion query	5,144159091
5	Оптимизация шейдеров	5,14004
6	Без оптимизации(просто окно (без вычислений функции render()))	5,138018

5 Заключение

В данной работе я изучил возможности языка C++ в работе с библиотекой OpenGL, а именно научился применять методы оптимизации(хоть, как показали тесты, для небольших проектов на C++ это особо не требуется) используя методы: glGenLists(1), glNewList(prism display list, GL COMPILE), glEndList(), glCallList(prism display list), glDeleteLists(prism display list, 1), glBindBuffer(GL ARRAY BUFFER, quadStripVBO), glEnableClientState(GL VERTEX ARRAY), glVertexPointer(3, GL FLOAT, 0, NULL), glDrawArrays(GL QUAD STRIP, 0, 361 * 2), glGenBuffers(1, quadStripVBO), glBindBuffer(GL ARRAY BUFFER, quadStripVBO), glBufferData(GL ARRAY BUFFER, sizeof(quadStripVertices), quadStripVertices, GL STATIC DRAW), glGenQueries(1, query id), glBeginQuery(GL SAMPLES PASSED, query id[0]), glEndQuery(GL SAMPLES PASSED), glGetQueryObjectuiv(query id[0], GL QUERY RESULT, samples passed), glEnableClientState(GL VERTEX ARRAY).