Санкт-Петербургский Политехнический Университет Петра Великого

Кафедра компьютерных систем и программных технологий

Отчёт по лабораторной работе №3-4

на тему: «Визуализация каркасной модели»

Курс: «Разработка графических приложений»

Выполнил студент:

Волкова М.Д.
Группа: 13541/2

Проверил:

Абрамов Н.А.

Санкт-Петербург
2018 г.

Содержание

1 Лабораторная работа №2 **2**
1.1 Цель работы . . . 2
1.2 Описание программы . . . 2
1.3 Ход работы . . . 3
1.3.1 Алгоритм Брезенхема . . . 3
1.3.2 Настройка фиксированной камеры . . . 4
1.3.3 Z-буферизация . . . 6
1.4 Результаты . . . 8
1.4.1 CvLineDrawer . . . 8
1.4.2 LineDrawer . . . 9
1.4.3 TriangleDrawer . . . 10
1.4.4 Z-буффер . . . 14
1.5 Вывод . . . 15
1.6 Листинги . . . 16
1.6.1 Листинг 1. main.cpp . . . 16
1.6.2 Листинг 2. Drawer.h . . . 19
1.6.3 Листинг 3. render.h . . . 21
1.6.4 Листинг 4. transformers.h . . . 29

Лабораторная работа №2

1.1 Цель работы

Разработать программу на языке C для растеризации загруженной модели на экран

1.2 Описание программы

1. Возможности программы:
   (a) Загрузка трехмерной модели из OBJ-файла
   (b) Растеризация каркаса трехмерной модели
   (c) Обеспечение вращения камеры вокруг трехмерной модели
   (d) Растеризация линий своим алгоритмом
   (e) Растеризация треугольников своим алгоритмом
   (f) Вычисление координат и получение значения глубины для конкретного пикселя
   (g) Использования буффера глубины для отсечения невидимых пикселей
2. Входные параметры программы:
   (a) Ширина и высота окна
   (b) Вертикальный угол обзора камеры для выполненя перспективной проекции
   (c) Ближняя и дальняя плоскости отсечения камеры
   (d) Дистанция от камеры до загруженной модели
   (e) Скорость вращения камеры вокруг модели (градус/сек)
3. Выходные параметры программы:
   (a) Последовательность кадров, выводимая на экран
4. Порядок работы программы:
   (a) Загрузка трехмерной модели в вершинные и индексные буфера
   (b) Определение центра модели (можно считать, что матрица мира для модели – единичная)
   (c) Формирование матрицы проекции
   (d) Далее – для очередного кадра:
      i. Формирование матрицы вида исходя из координат центра модели, дистанции до модели и скорости вращения камеры
      ii. Преобразование вершин модели в экранные координаты

1.3 Ход работы

В дополнение к уже установленной ранее библиотеке OpenCV дополнительно была установлена библиотека GLM, предназначенная для работы с векторами и матрицами размерности до 4-х. Для работы с форматом OBJ использована библиотека TinyObj.

Программа предоставлена в листинге.

1.3.1 Алгоритм Брезенхема

Алгоритм Брезенхема - это алгоритм, определяющий, какие точки двумерного растра нужно закрасить, чтобы получить близкое приближение прямой линии между двумя заданными точками.

Для проволочного рендеринга, сначала нам нужна функция, оторая будет отрисовывать линии:

```
    template<typename F>
    inline void drawline(int x0, int y0, int x1, int y1, F plot) {

        int dx = std::abs(x1 - x0);
        int dy = std::abs(y1 - y0);

        int directionX = x0 < x1 ? 1 : -1;
        int directionY = y0 < y1 ? 1 : -1;
        int err = (dx > dy ? dx : -dy) / 2;

        for (;;) {
            plot(x0, y0);
            if (x0 == x1 && y0 == y1) break;
            int e2 = err;
            if (e2 > -dx) {
                err -= dy;
                x0 += directionX;
            }
            if (e2 < dy) {
                err += dx;
                y0 += directionY;
            }
        }
    }
```

Для хранения модели мы используем формат wavefront obj. Формат файлов OBJ - это простой формат данных, который содержит только 3D геометрию, а именно, позицию каждой вершины, связь координат текстуры с вершиной, нормаль для каждой вершины, а также параметры, которые создают полигоны.Всё, что нам нужно для рендера, это прочитать из файла массив вершин вида:

```
v 0.608654 -0.568839 -0.416318
```

это координаты x,y,z, одна вершина на строку файла и граней:

```
f 7 6 1
```

еще в файле содержиться нормали (нормали могут быть не нормированными):

```
vn -0.966742 -0.255752 9.97231e-09
```

Нас интересуют первое число после каждого пробела, это номер вершины в массиве, который мы прочитали ранее. Таким образом эта строчка говорит, что вершины 7, 6 и 1 образуют треугольник.

Далее пишем функцию, которая принимает объект класса Drawer, вершины и координаты модели и рисует линии:

```
void render(Drawer &drawer, const std::vector<glm::vec3> &vertices, const std::vector<
    unsigned int> &indices) {
    for (auto i = 0; i < indices.size(); i += 3) {
        Triangle triangle{vertices[indices[i]], vertices[indices[i + 1]], vertices[
    indices[i + 2]]};
        drawer.draw(triangle);
    }
}
```

1.3.2 Настройка фиксированной камеры

В OpenGL при использовании фиксированного конвейера есть ровно две матрицы, относящихся к трансформациям точек и объектов:

- GL PROJECTION моделирует ортографическое или перспективное преобразование от трёхмерной усечённой пирамиды (т.е. от области видимости камеры) к трёхмерному кубу с длиной ребра, равной 2 (т.е. к нормализованному пространству).
- GL MODELVIEW сочетает в себе два преобразования: от локальных координат объекта к мировым координатам, а также от мировых координат к координатам камеры.

За рамками фиксированного конвейера можно использовать столько матриц, сколько захочется.

- поведение камеры описывается как ортографическим или перспективным преобразованием, так и положением камеры в мировом пространстве, то есть для моделирования камеры нужны GL PROJECTION и GL MODELVIEW одновременно
- с другой стороны, для трансформаций над телами — вращение предмета с помощью умножения координат на матрицу — нужна матрица GL MODELVIEW.

Настроим матрицу GL PROJECTION один раз для перспективного преобразования, а матрицу GL MODELVIEW будем постоянно модифицировать, когда локальная система координат очередного объекта не совпадает с мировой системой координат.

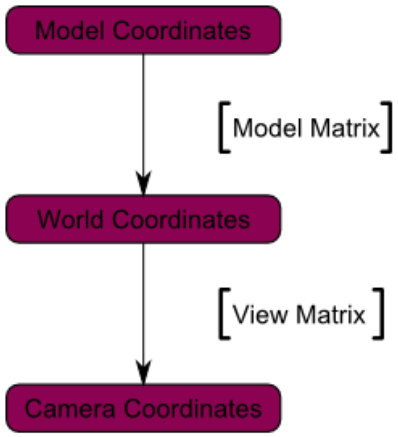
Начнём настройку камеры с GL MODELVIEW: зададим матрицу так, как будто бы камера смотрит с позиции camera position на точку model center, при этом направление “вверх” камеры задаёт вектор glm::vec3(0, 1, 0):

```
camera = glm::lookAt(
                camera_position,
                model_center,
                glm::vec3(0, 1, 0)
        );
```

, где:

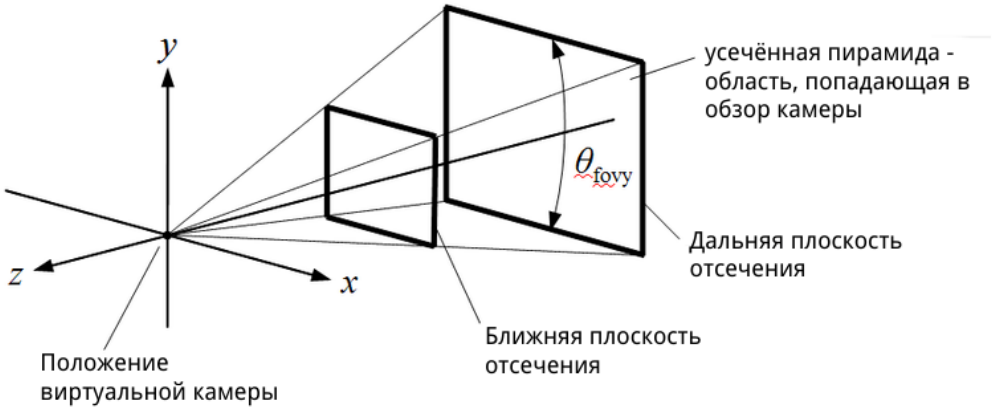
- camera position - Позиция камеры в мировом пространстве
- model center - Указывает куда вы смотрите в мировом пространстве
- glm::vec3(0, 1, 0) - Вектор, указывающий направление вверх

А вот диаграмма, которая показывает то, что мы делаем:



Для перспективного преобразования достаточно создать матрицу с помощью функции glm::perspective. Она принимает на вход несколько параметров преобразования: горизонтальный угол обзора камеры, соотношение ширины и высоты, а также две граничных координаты для отсечения слишком близких к камере и слишком далёких от камеры объектов.

Эти параметры легко увидеть на следующей иллюстрации:



Проекционная матрица

Теперь мы находимся в пространстве камеры, вершина, которая получит координаты x == 0 и y == 0 будет отображаться по центру экрана. Однако, при отображении объекта огромную роль играет также дистанция до камеры. Для двух вершин, с одинаковыми x и y, вершина имеющая большее значение по z будет отображаться ближе, чем другая.

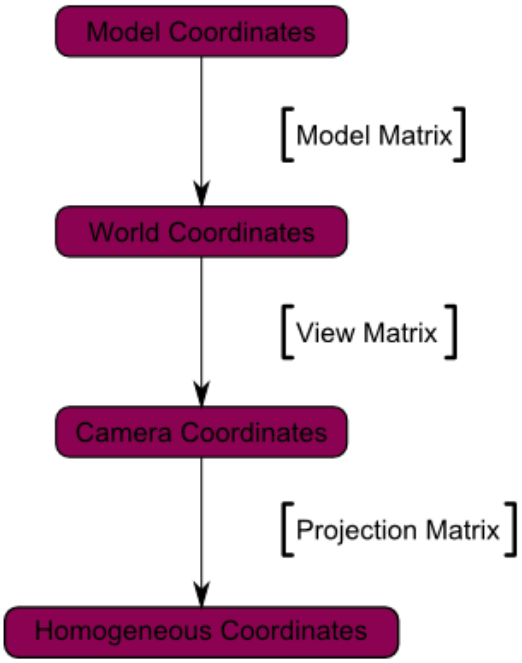
Это называется перспективной проекцией, к счастью, в glm имеем:

```
projection = glm::perspective(
             glm::radians(fovy),
             screen_ratio,
             front,
             back)
```

, где:

- glm::radians(fovy) - Вертикальное поле зрения в радианах.
- screen ratio - Отношение сторон.
- front - Ближняя плоскость отсечения.
- back - Дальняя плоскость отсечения.

Теперь наша диаграмма будет выглядеть так:



Матрица поворота

Функция rotate в glm поворачивает 3D вектор на заданный угол вокруг заданной оси (представленной орт-вектором):

```
glm::rotate(glm::mat4(1), angle, glm::vec3(0, 1, 0));
```

Следующий шагом объединяем трансформации, что реализуется по следующей формуле:

```
camera_position = (rotation_matrix * start_camera_position)
```

1.3.3 Z-буферизация

Z-буферизация - это способ учёта удалённости элемента изображения. Представляет собой один из вариантов решения «проблемы видимости». Очень эффективен и практически не имеет недостатков, если реализуется аппаратно.

Формальное описание алгоритма z-буфера таково:

1. Заполнить буфер кадра фоновым значением интенсивности или цвета.
2. Заполнить z-буфер минимальным значением z.
3. Преобразовать каждый многоугольник в растровую форму в произвольном порядке.
4. Для каждого Пиксел(x,y) в многоугольнике вычислить его глубину z(x,y).
5. Сравнить глубину z(x,y) со значением Zбуфер(x,y), хранящимся в z-буфере в этой же позиции.
6. Если z(x,y) > Zбуфер (x,y), то записать атрибут этого многоугольника (интенсивность, цвет и т. п.) в буфер кадра и заменить Zбуфер(x,y) на z(x,y). В противном случае никаких действий не производить.

Поскольку у нас экран двумерный, то z-буфер тоже должен быть двумерным:

```
    cv::Mat1d zBufferDefaultValue(int height, int width) {
        return cv::Mat::ones(height, width, CV_64F) * 100000;
    }

```

Затем в коде мы проходим по всем треугольникам и делаем вызов растеризатора, передавая ему и картинку, и z-буфер.

```
       void triangleBarycentric(const ExtendedTriangle &t, cv::Mat1d &zBuffer, const
    MipMap<8> &mipMap) {
        auto &&bbox = t.box();
        Coord from = bound(bbox.first);
        Coord to = bound(bbox.second);
        for (auto x = from.x; x <= to.x; ++x) {
            for (auto y = from.y; y <= to.y; ++y) {
                glm::dvec3 barycentric;
                bool pointInTriangle = get_barycentric(t, glm::vec4(x, y, 1, 1),
    barycentric);
                if (!pointInTriangle) continue;
                glm::dvec3 zInterpolated = glm::dvec3(t.a.z, t.b.z, t.c.z) * barycentric;
                double z = zInterpolated[0] + zInterpolated[1] + zInterpolated[2];
                if (z < zBuffer.at<double>(y, x)) {
                    zBuffer.at<double>(y, x) = z;

                    glm::dvec3 barycentricX, barycentricY;
                    get_barycentric(t, glm::vec4(x + 1, y, 1, 1), barycentricX);
                    get_barycentric(t, glm::vec4(x, y + 1, 1, 1), barycentricY);

                    glm::vec2 uv = interpolateTexture(t.texture, barycentric);
                    glm::vec2 uvX = interpolateTexture(t.texture, barycentricX);
                    glm::vec2 uvY = interpolateTexture(t.texture, barycentricY);

                    glm::vec2 dx = uvX - uv;
                    glm::vec2 dy = uvY - uv;

```

```
                    float l = std::max(glm::dot(dx, dx), glm::dot(dy, dy));
                    float d = std::max(0.0f, 0.5f * std::log2(l));

                    cv::Vec3b texel = trilinearFilter(uv, d, mipMap);
                    plot(x, y, texel);

                }
            }
        }
    }

```

В этой функции мы переводим координаты в барицентрические, с помощью следующей функии:

```
    bool get_barycentric(const ExtendedTriangle &t, const glm::vec4 &p, glm::dvec3 &
    barycentric) {
        auto area = edgeFunction(t.a, t.b, t.c);
        auto w0 = edgeFunction(t.b, t.c, p) / area;
        auto w1 = edgeFunction(t.c, t.a, p) / area;
        auto w2 = edgeFunction(t.a, t.b, p) / area;

        barycentric = glm::dvec3(w0, w1, w2);
        barycentric /= barycentric[0] + barycentric[1] + barycentric[2];
        barycentric = glm::abs(barycentric);

        return !(w0 < 0 || w1 < 0 || w2 < 0);
    }

```

Все, что с отрицательным значением - отбрасывается.

```
        double min, max;
        cv::minMaxLoc(drawer.zBuffer, &min, &max);
        cv::Mat zNorm;
        cv::normalize(drawer.zBuffer, zNorm, 0.0, 255.0, cv::NORM_MINMAX, CV_8UC1);

```

Функция minMaxLoc находит минимальное и максимальное значения элементов и их положения.

normalize нормализует src таким образом, что минимальное значение выходного массива равно 0.0, а максимальное значение выходного массива равно 255.0.

1.4 Результаты

В качестве тестовой модели для проверки работоспособности программы использовалась модель чайника, экспортированная стандартными средствами в формат OBJ.

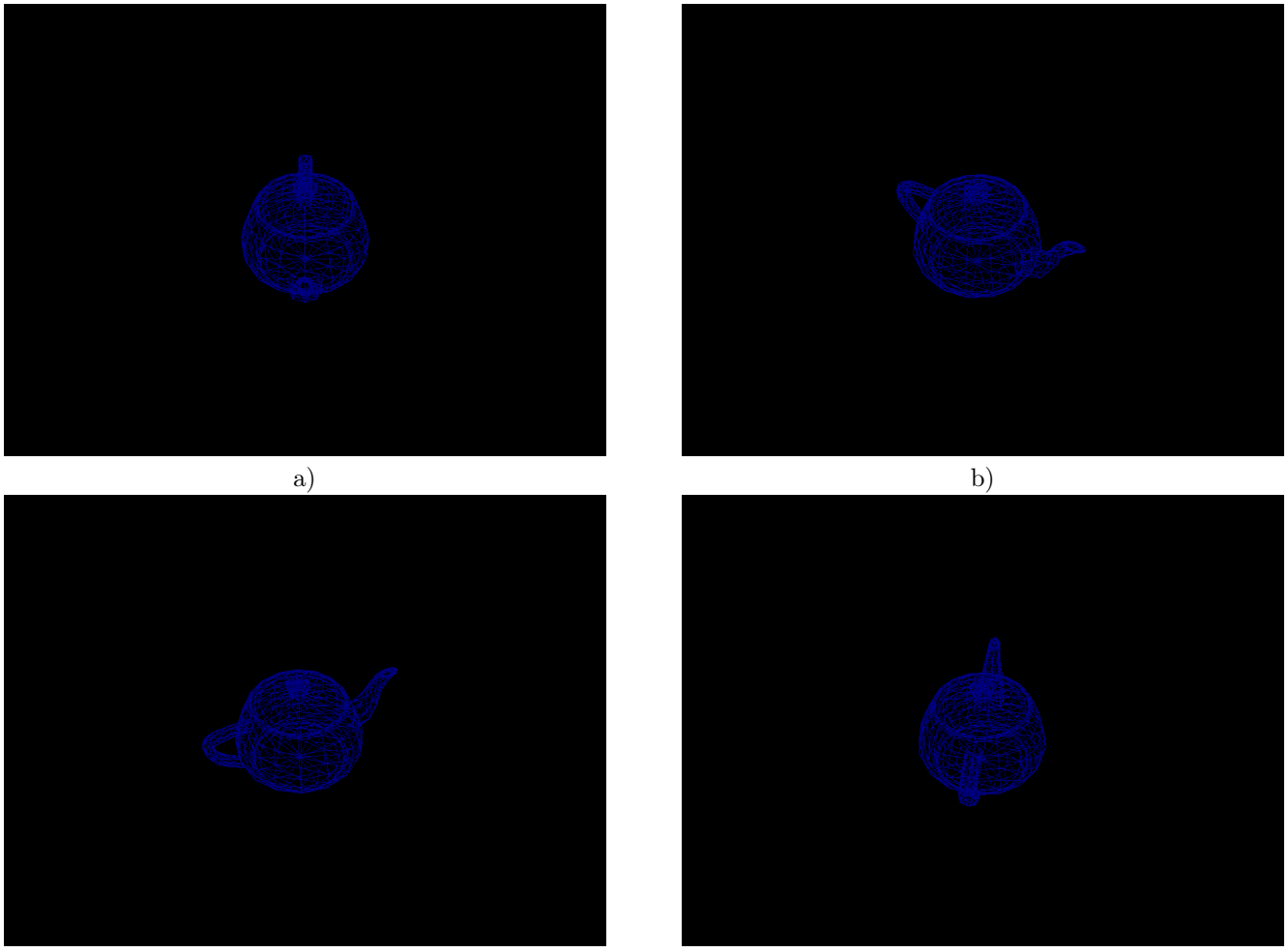
1.4.1 CvLineDrawer

Параметры:

```
options.add_options()
        ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
        ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
        ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("10.0"))
        ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-50.0"))
        ("dx", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("dy", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("1.0"))
        ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("100.0"))
        ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value(default_file_path))
        ("o,out_file", "Output filename ", cxxopts::value<std::string>()->default_value(default_save_path));
```

Рис. 1.1: Параметры

Результат работы:



a) b)

c) d)

Рис. 1.2: Последовательно создаваемые изображения

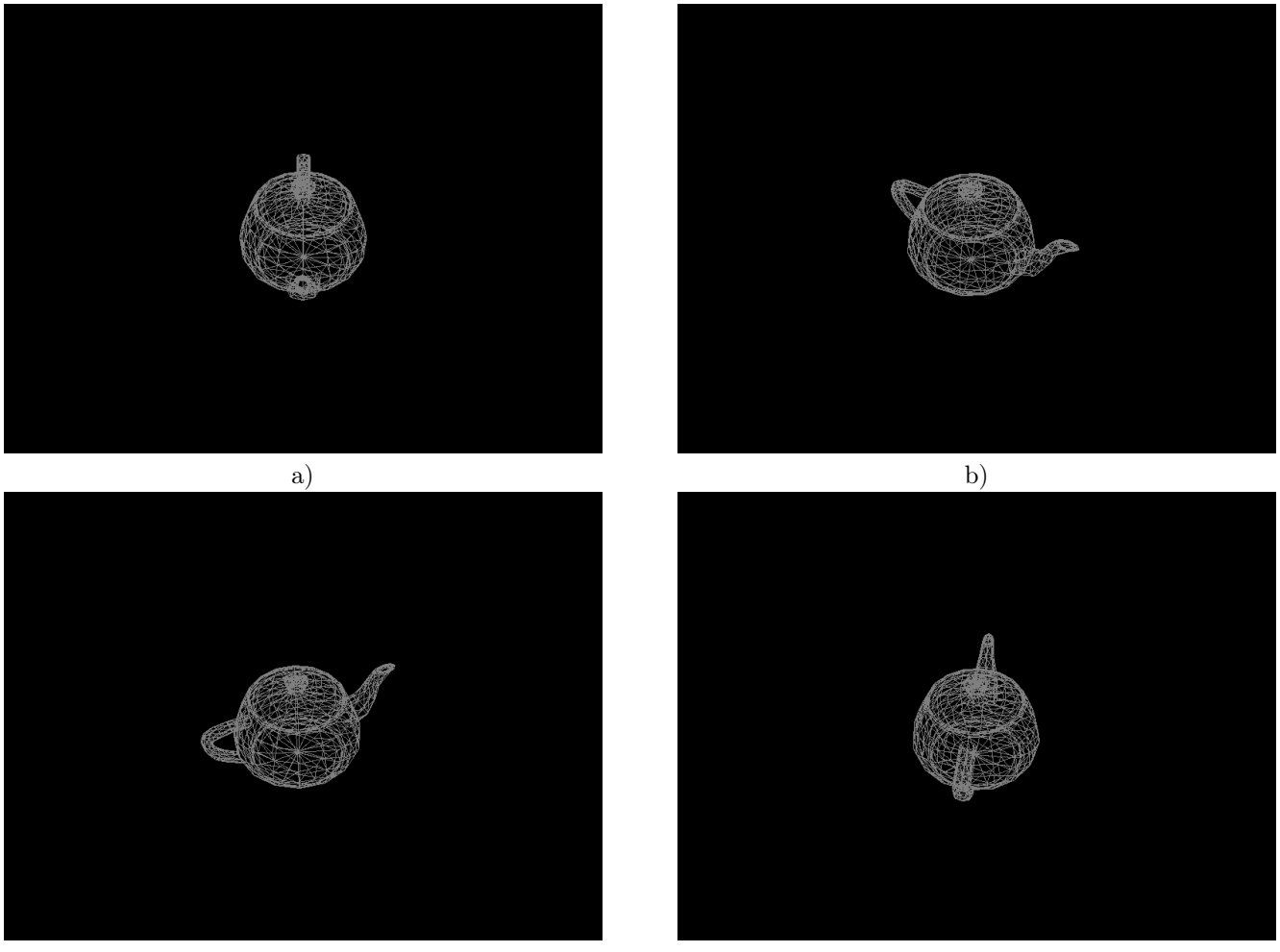
1.4.2 LineDrawer

Параметры:

```
options.add_options()
        ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
        ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
        ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("10.0"))
        ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-50.0"))
        ("dx", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("dy", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("1.0"))
        ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("100.0"))
        ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value(default_file_path))
        ("o,out_file", "Output filename ", cxxopts::value<std::string>()->default_value(default_save_path));
```

Рис. 1.3: Параметры

Результат работы:



a) b)

c) d)

Рис. 1.4: Последовательно создаваемые изображения

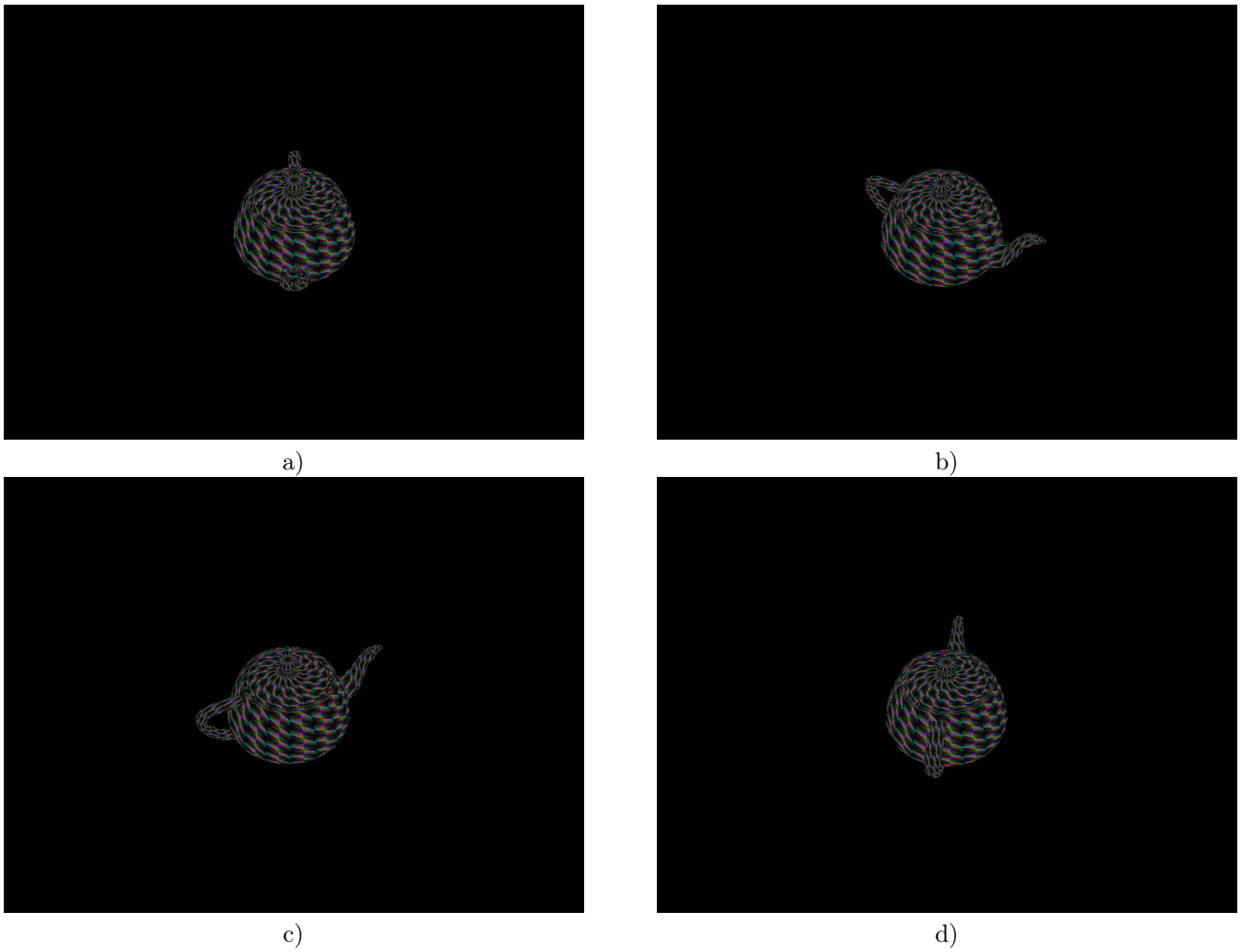
1.4.3 TriangleDrawer

Параметры:

```
options.add_options()
        ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
        ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
        ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("10.0"))
        ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-50.0"))
        ("dx", "Distance to model", cxxopts::value<int>()->default_value("800"))
        ("dy", "Distance to model", cxxopts::value<int>()->default_value("-300"))
        ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("1.0"))
        ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("100.0"))
        ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value(default_file_path))
        ("o,out_file", "Output filename ", cxxopts::value<std::string>()->default_value(default_save_path));
```

Рис. 1.5: Параметры

Результат работы:



a) b)

c) d)

Рис. 1.6: Последовательно создаваемые изображения

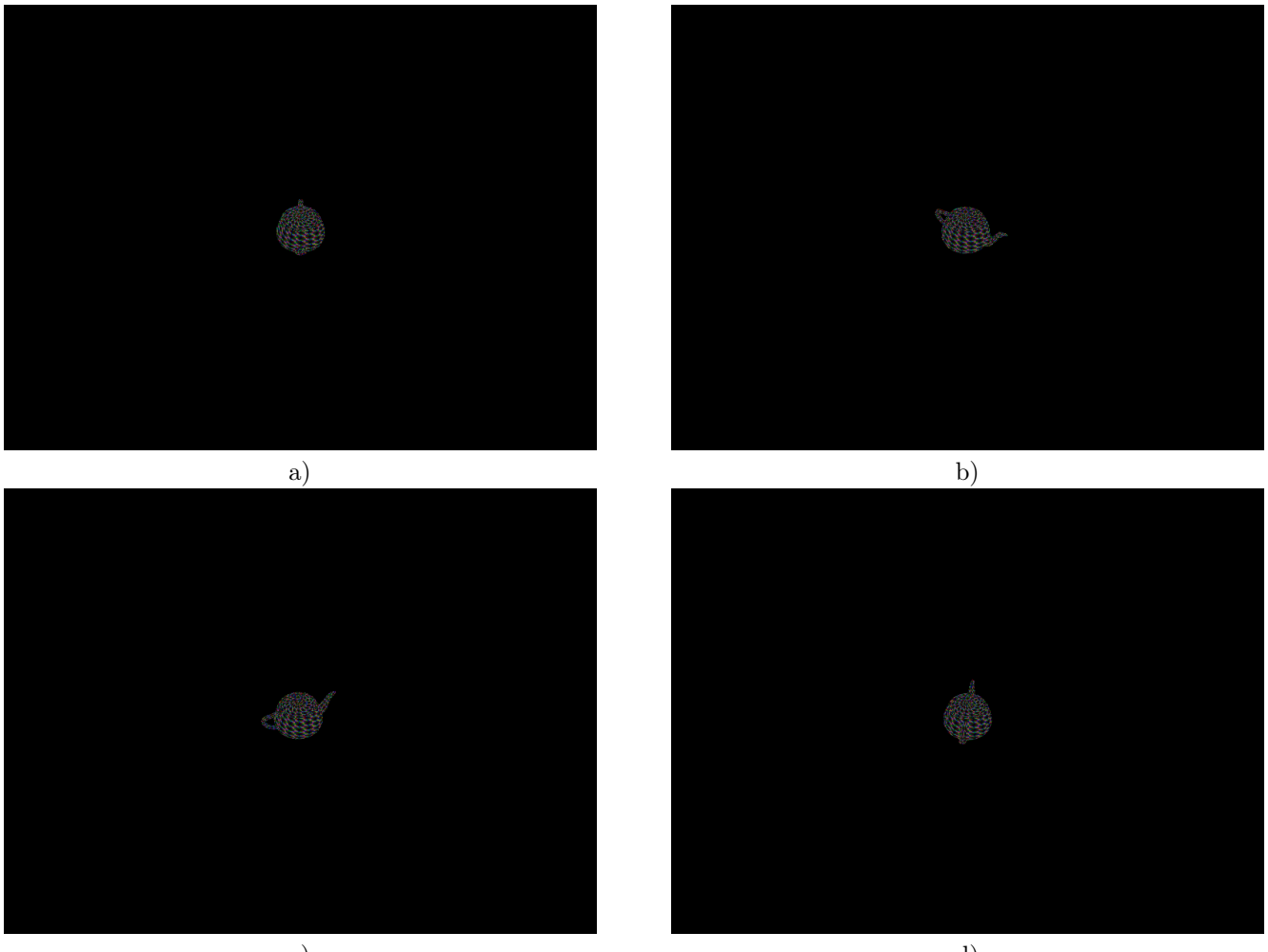
Приведем еще несколько результатов, изменяя параметры камеры:

Параметры:

```
options.add_options()
        ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
        ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
        ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("10.0"))
        ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-100.0"))
        ("dx", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("dy", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("1.0"))
        ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("100.0"))
        ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value(default_file_path))
        ("o,out_file", "Output filename ", cxxopts::value<std::string>()->default_value(default_save_path));
```

Рис. 1.7: Параметры

Результат работы:



a) b)

c) d)

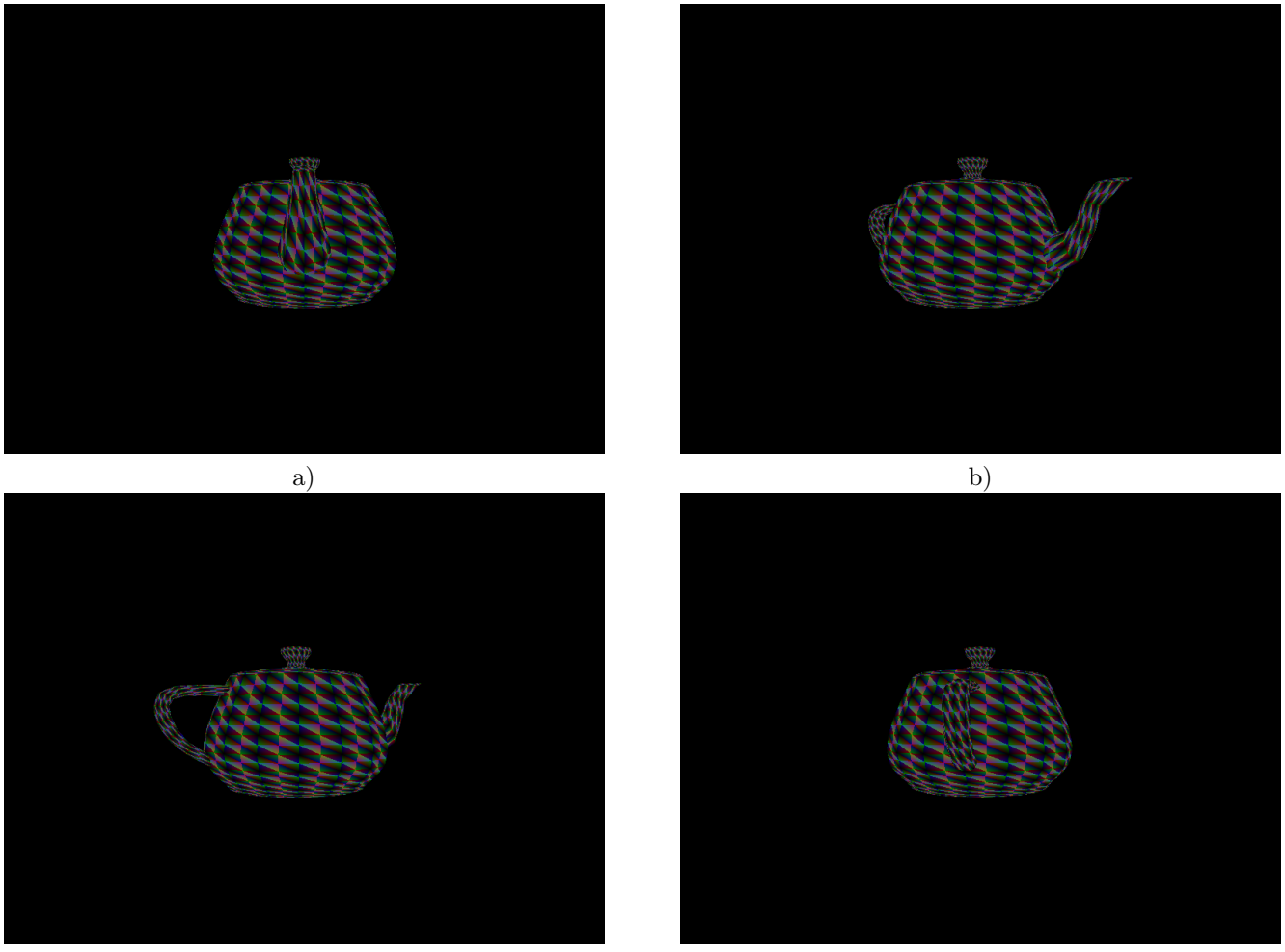
Рис. 1.8: Последовательно создаваемые изображения

Параметры:

```
options.add_options()
        ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
        ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
        ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("10.0"))
        ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-50.0"))
        ("dx", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("dy", "Distance to model", cxxopts::value<int>()->default_value("0"))
        ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("1.0"))
        ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("100.0"))
        ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value(default_file_path))
        ("o,out_file", "Output filename ", cxxopts::value<std::string>()->default_value(default_save_path));
```

Рис. 1.9: Параметры

Результат работы:



a) b)

c) d)

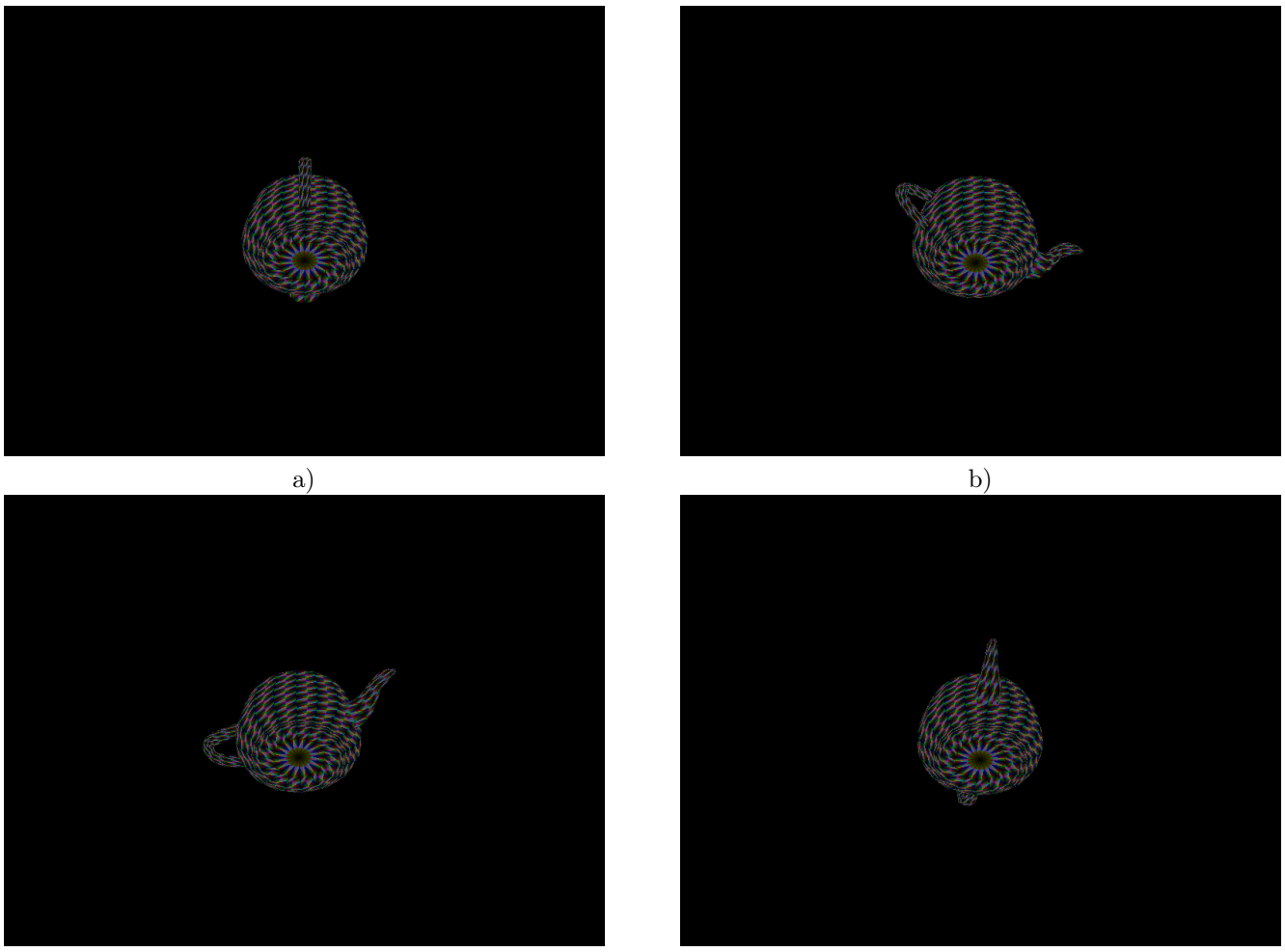
Рис. 1.10: Последовательно создаваемые изображения

Параметры:

```
options.add_options()
        ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
        ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
        ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("10.0"))
        ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-50.0"))
        ("dx", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("dy", "Distance to model", cxxopts::value<int>()->default_value("300"))
        ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("150.0"))
        ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("100.0"))
        ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value(default_file_path))
        ("o,out_file", "Output filename ", cxxopts::value<std::string>()->default_value(default_save_path));
```

Рис. 1.11: Параметры

Результат работы:

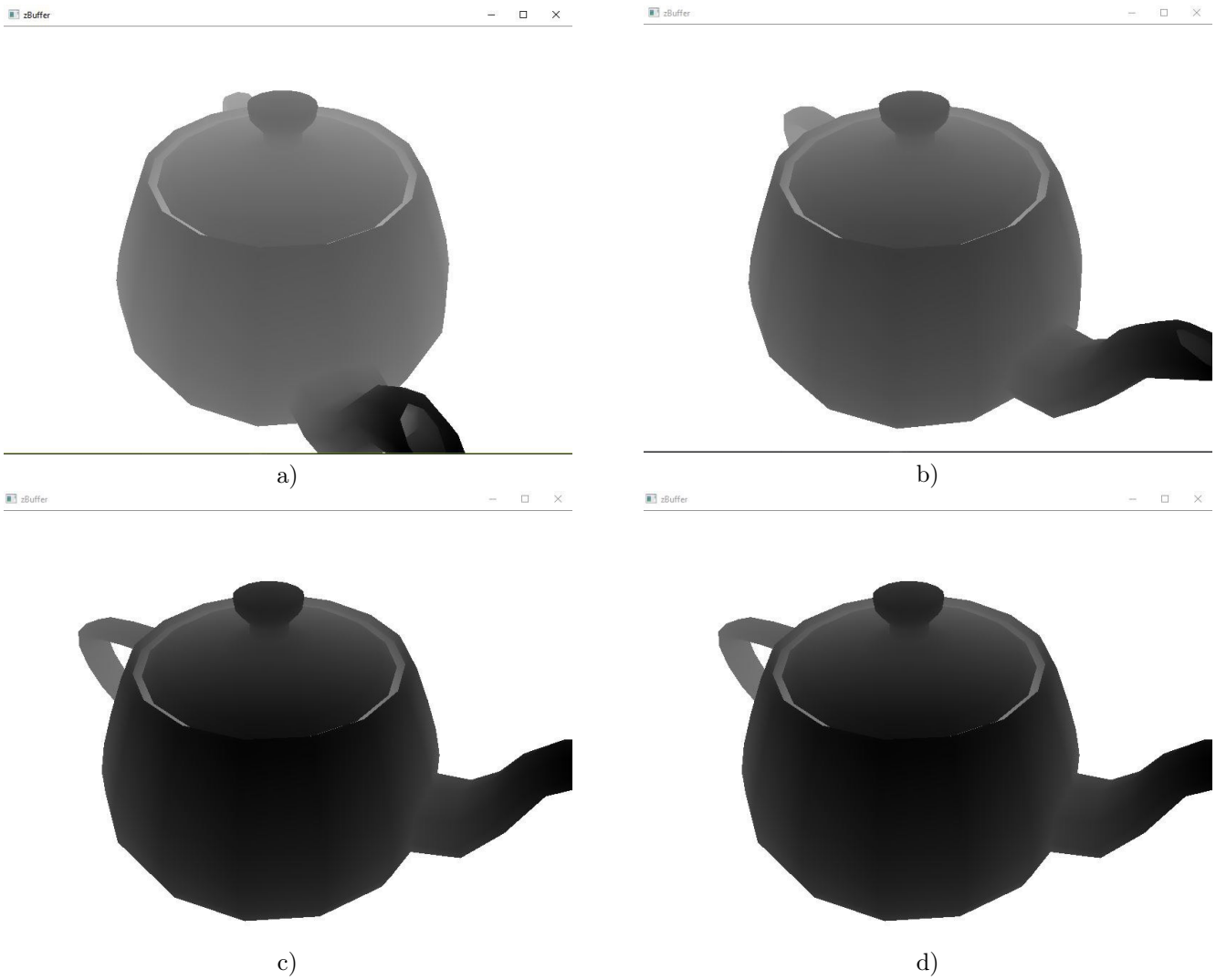


a) b)

c) d)

Рис. 1.12: Последовательно создаваемые изображения

1.4.4 Z-буффер



a) b)

c) d)

Рис. 1.13: Последовательно создаваемые изображения

1.5 Вывод

В данной работе была изучена библиотека GLM и составлена программа для визуализации трехмерной модели в виде проволочного каркаса с использованием средств библиотеки OpenCV.

Результаты визуализации отвечают ожиданиям при заданном смещении, повороте и масштабе модели. Для создания более полного представления наблюдателя о внешнем виде исходной модели, необходимо в дальнейшем реализовать отображение поверхностей модели, посредством треугольников, учитывая, что используемая библиотека TinyObj позволяет проводить разбиение произвольного полигона на треугольники автоматически при чтении файла модели.

1.6 Листинги

1.6.1 Листинг 1. main.cpp

```cpp
#include <utility>

#include <iostream>
#include <any>
#include <OBJ_Loader.h>
#include <glm/vec3.hpp>
#include <glm/geometric.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <cxxopts.hpp>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>

#include "render.h"
#include "Drawer.h"
#include "transformers.h"

const int FRAME_PER_SECOND = 10;
const int FRAME_COUNT = 10000;


template<int index>
float min(const std::vector<glm::vec3> &vertices) {
    float result = FLT_MAX;
    for (auto &&vex : vertices) {
        result = std::min(result, vex[index]);
    }
    return result;
}

template<int index>
float max(const std::vector<glm::vec3> &vertices) {
    float result = FLT_MIN;
    for (auto &&vex : vertices) {
        result = std::max(result, vex[index]);
    }
    return result;
}

template<int index>
float getCenter(const std::vector<glm::vec3> &vertices) {
    auto &&min_point = min<index>(vertices);
    auto &&max_point = max<index>(vertices);
    return (min_point + max_point) / 2;
}

glm::vec3 getModelCenter(const std::vector<glm::vec3> &vertices) {
    auto &&center_x = getCenter<0>(vertices);
    auto &&center_y = getCenter<1>(vertices);
    auto &&center_z = getCenter<2>(vertices);
    return glm::vec3(center_x, center_y, center_z);
}

void render(Drawer &drawer, const std::vector<glm::vec3> &vertices, const std::vector<
    unsigned int> &indices) {
    for (auto i = 0; i < indices.size(); i += 3) {
        Triangle triangle{vertices[indices[i]], vertices[indices[i + 1]], vertices[
    indices[i + 2]]};
        drawer.draw(triangle);
    }
```

```
}

void render(Drawer &drawer,
            const std::vector<glm::vec3> &vertices,
            const std::vector<glm::vec3> &normals,
            const std::vector<glm::vec2> &texture,
            const std::vector<unsigned int> &indices
) {
    for (auto i = 0; i < indices.size(); i += 3) {
        Triangle triangle(vertices[indices[i]], vertices[indices[i + 1]], vertices[indices[i + 2]]);
        TriangleTexture tex{texture[indices[i]], texture[indices[i + 1]], texture[indices[i + 2]]};
        TriangleNormal norm{normals[indices[i]], normals[indices[i + 1]], normals[indices[i + 2]]};
        triangle.normal = norm;
        triangle.texture = tex;
        drawer.draw(triangle);
    }
}

int main(int argc, char **argv) {
    cxxopts::Options options("Lba3", "Render teapot and maybe something else");
    std::string default_file_path = "../Corvette-F3.obj";
    std::string default_texture_path = "../SF_Corvette-F3_diffuse.jpg";
    std::string default_save_path = "../Corvette-F3.avi";

    options.add_options()
            ("w,width", "Width of image", cxxopts::value<int>()->default_value("800"))
            ("h,height", "Height of image", cxxopts::value<int>()->default_value("600"))
            ("s,speed", "Camera speed", cxxopts::value<float>()->default_value("2.0"))
            ("v,fovy", "fovy", cxxopts::value<float>()->default_value("-50.0"))
            ("dx", "Distance to model", cxxopts::value<int>()->default_value("2000"))
            ("dy", "Distance to model", cxxopts::value<int>()->default_value("2000"))
            ("f,front", "Front cut plane", cxxopts::value<float>()->default_value("0.1"))
            ("b,back", "Back cut plane", cxxopts::value<float>()->default_value("30000.0"))
            ("i,in_file", "Input filename ", cxxopts::value<std::string>()->default_value(default_file_path))
            ("t,texture", "Texture filename ", cxxopts::value<std::string>()->default_value(default_texture_path))
            ("o,out_file", "Output filename ", cxxopts::value<std::string>()->default_value(default_save_path));


    auto &&arguments = options.parse(argc, argv);

    auto &&width = arguments["width"].as<int>();
    auto &&height = arguments["height"].as<int>();
    auto &&speed = arguments["speed"].as<float>();
    auto &&fovy = arguments["fovy"].as<float>();
    auto &&distanceX = arguments["dx"].as<int>();
    auto &&distanceY = arguments["dy"].as<int>();
    auto &&front = arguments["front"].as<float>();
    auto &&back = arguments["back"].as<float>();
    auto &&file_name = arguments["in_file"].as<std::string>();
    auto &&texture_file_name = arguments["texture"].as<std::string>();
    auto &&res_file_name = arguments["out_file"].as<std::string>();

    objl::Loader loader;
    loader.LoadFile(file_name);
    auto &&mesh = loader.LoadedMeshes[0];

    auto &&model_vertices = ToGLMVertices().applyList<objl::Vertex, glm::vec3>(mesh.Vertices);
```

```
117     auto &&model_normals = ToGLMNormals().applyList<objl::Vertex, glm::vec3>(mesh.
        Vertices);
118     auto &&model_texture_coordinates = ToGLMTexture().applyList<objl::Vertex, glm::vec2>(
        mesh.Vertices);
119     cv::Mat3b texture = cv::imread(texture_file_name);
120
121     MipMap mipMap = MipMap<8>::compute(texture);
122
123     auto &&model_center = getModelCenter(model_vertices);
124
125     auto &&screen_ratio = static_cast<float>(width) / static_cast<float>(height);
126     auto &&projection = glm::perspective(
127             glm::radians(fovy),
128             screen_ratio,
129             front,
130             back
131     );
132
133     float angle = 0;
134     float angle_per_frame = speed / FRAME_PER_SECOND;
135
136     auto &&start_camera_position = glm::vec4(distanceX, distanceY, 0, 1);
137
138     TriangleDrawer drawer(width, height);
139     drawer.mipMap = mipMap;
140
141     for (auto i = 0; i < FRAME_COUNT; i++) {
142         drawer.resetImage();
143         glm::mat4 rotation_matrix = glm::rotate(glm::mat4(1), angle, glm::vec3(0, 1, 0));
144         glm::vec3 camera_position = (rotation_matrix * start_camera_position);
145         auto &&camera = glm::lookAt(
146                 camera_position,
147                 model_center,
148                 glm::vec3(0, 1, 0)
149         );
150
151         drawer.updatePipeline(std::make_unique<TriangleTransformationPipeline>(camera,
        projection, width, height));
152         render(drawer, model_vertices, model_normals, model_texture_coordinates, mesh.
        Indices);
153
154         cv::imshow("Aaaa", drawer.getImage());
155         cv::waitKey(2000);
156
157         angle += angle_per_frame;
158     }
159     return 0;
160 }
```

1.6.2 Листинг 2. Drawer.h

```
#pragma once

#include "render.h"
#include "transformers.h"


class Drawer {
public:
    explicit Drawer(int width, int height) : renderer(width, height) {};

    virtual void draw(const Triangle &t) = 0;

    cv::Mat getImage() const {
        return renderer.getImage();
    }

    virtual void resetImage() {
        renderer.resetImage();
    }

protected:
    Renderer renderer;
};


class LineDrawer : public Drawer {
public:
    LineDrawer(int width, int height) : Drawer(width, height), pipeline() {}

    void draw(const Triangle &triangle) override {
        auto &&t = pipeline->apply(triangle);
        renderer.line(t.aCoord(), t.bCoord());
        renderer.line(t.bCoord(), t.cCoord());
        renderer.line(t.cCoord(), t.aCoord());
    }

    void updatePipeline(std::unique_ptr<TransformationPipeline<Triangle, Triangle>>
    newPipeline) {
        pipeline = std::move(newPipeline);
    }

protected:
    std::unique_ptr<TransformationPipeline<Triangle, Triangle>> pipeline;

};

class CvLineDrawer : public LineDrawer {
    using LineDrawer::LineDrawer;

    void draw(const Triangle &triangle) override {
        auto &&t = pipeline->apply(triangle);
        renderer.cv_line(t.aCoord(), t.bCoord());
        renderer.cv_line(t.bCoord(), t.cCoord());
        renderer.cv_line(t.cCoord(), t.aCoord());
    }
};

class TriangleDrawer : public Drawer {
public:
    TriangleDrawer(int width, int height) : Drawer(width, height), zBuffer(
    zBufferDefaultValue(height, width)),
                                            pipeline() {}

    void draw(const Triangle &triangle) override {
```

```
63          auto &&t = pipeline->apply(triangle);
64          renderer.triangleBarycentric(t, zBuffer, mipMap);
65      }
66
67      cv::Mat1d zBufferDefaultValue(int height, int width) {
68          return cv::Mat::ones(height, width, CV_64F) * 100000;
69      }
70
71      void resetImage() override {
72          renderer.resetImage();
73          zBuffer = zBufferDefaultValue(renderer.height, renderer.width);
74      }
75
76
77      void updatePipeline(std::unique_ptr<TransformationPipeline<Triangle, ExtendedTriangle
        >> newPipeline) {
78          pipeline = std::move(newPipeline);
79      }
80
81  //protected:
82      std::unique_ptr<TransformationPipeline<Triangle, ExtendedTriangle>> pipeline;
83      cv::Mat1d zBuffer;
84      MipMap<8> mipMap;
85  };
```

1.6.3 Листинг 3. render.h

```cpp
#pragma once

#include <iostream>

#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>

#include <glm/vec3.hpp>
#include <glm/vec4.hpp>
#include <glm/mat3x3.hpp>
#include <glm/geometric.hpp>

struct Coord {
    int x, y;

    cv::Point toCV() const {
        return cv::Point(x, y);
    }
};

struct TriangleNormal {
    glm::vec3 a, b, c;
};

struct TriangleTexture {
    glm::vec2 a, b, c;
};


template<typename T>
struct TriangleBase {
    TriangleBase(const T &aa, const T &bb, const T &cc) : a(aa), b(bb), c(cc) {}

    virtual Coord toCoord(const T &v) const = 0;

    Coord aCoord() const {
        return toCoord(a);
    }

    Coord bCoord() const {
        return toCoord(b);
    }

    Coord cCoord() const {
        return toCoord(c);
    }

    virtual glm::vec3 toVec(const T &v) const = 0;

    glm::vec3 aVec() const {
        return toVec(a);
    }

    glm::vec3 bVec() const {
        return toVec(b);
    }

    glm::vec3 cVec() const {
        return toVec(c);
    }


```

```
65      T a, b, c;
66      TriangleNormal normal;
67      TriangleTexture texture;
68
69  };
70
71  inline cv::Vec3b getFrom(const cv::Mat3b &image, const glm::vec2 &coords) {
72      return image.at<cv::Vec3b>(
73              static_cast<int>(coords.y),
74              static_cast<int>(coords.x)
75      );
76  }
77
78  struct Triangle : public TriangleBase<glm::vec3> {
79      using TriangleBase::TriangleBase;
80
81      Coord toCoord(const glm::vec3 &v) const override {
82          int x = static_cast<int>(v.x);
83          int y = static_cast<int>(v.y);
84          return {x, y};
85      }
86
87      glm::vec3 toVec(const glm::vec3 &v) const override {
88          return v;
89      }
90  };
91
92  struct ExtendedTriangle : public TriangleBase<glm::vec4> {
93      using TriangleBase::TriangleBase;
94
95      Coord toCoord(const glm::vec4 &v) const override {
96          int x = static_cast<int>(v.x);
97          int y = static_cast<int>(v.y);
98          return {x, y};
99      }
100
101     glm::vec3 toVec(const glm::vec4 &v) const override {
102         return glm::vec3(v.x, v.y, v.z);
103     }
104
105
106     std::pair<Coord, Coord> box() const {
107         int minX = static_cast<int>(std::floor(std::min({a.x, b.x, c.x})));
108         int minY = static_cast<int>(std::floor(std::min({a.y, b.y, c.y})));
109         int maxX = static_cast<int>(std::ceil(std::max({a.x, b.x, c.x})));
110         int maxY = static_cast<int>(std::ceil(std::max({a.y, b.y, c.y})));
111         return std::make_pair(Coord{minX, minY}, Coord{maxX, maxY});
112     }
113
114 };
115
116
117 std::ostream &operator<<(std::ostream &res, const glm::vec4 &v) {
118     res << "(" << v.x << ", " << v.y << ", " << v.z << ")";
119     return res;
120 }
121
122 std::ostream &operator<<(std::ostream &res, const glm::vec3 &v) {
123     res << "(" << v.x << ", " << v.y << ", " << v.z << ")";
124     return res;
125 }
126
127
128 std::ostream &operator<<(std::ostream &res, const glm::vec2 &v) {
129     res << "(" << v.x << ", " << v.y << ")";
130     return res;
```

```
}

template<int level = 8>
struct MipMap {
    static MipMap<level> compute(const cv::Mat &original) {
        MipMap<level> mipmap;
        mipmap.atLevel[0] = original;
        for (auto i = 1; i < level; i++) {
            mipmap.atLevel[i] = imageAverage(mipmap.atLevel[i - 1]);
        }
        return mipmap;
    }

    static cv::Mat imageAverage(const cv::Mat &image) {
        auto height = image.size().height, width = image.size().width;
        cv::Mat result(height / 2, width / 2, image.type());
        for (auto x = 0; x < width; x += 2) {
            for (auto y = 0; y < height; y += 2) {
                cv::Mat block = image(cv::Rect(x, y, 2, 2));
                cv::Scalar averaged = cv::mean(block);
                result.at<cv::Vec3b>(y / 2, x / 2) = cv::Vec3b(
                        static_cast<uchar>(averaged[0]),
                        static_cast<uchar>(averaged[1]),
                        static_cast<uchar>(averaged[2])
                );
            }
        }
        return result;
    }

    cv::Mat3b get(int d) const {
        int realD = std::max(0, std::min(d, level - 1));
        return atLevel[realD];
    }

    glm::vec2 scale(int d, const glm::vec2 &uv) const {
        int realD = std::max(0, std::min(d, level - 1));
        auto &&size = atLevel[realD].size();
        return uv * glm::vec2(size.width, size.height);
    }

private:
    cv::Mat3b atLevel[level];
};


class Renderer {
public:
    explicit Renderer(int w, int h) : width(w), height(h), image(cv::Mat::zeros(h, w,
    CV_8U)) {}

    void line(const Coord &from, const Coord &to) {
        drawline(from.x, from.y, to.x, to.y);
    }

    void cv_line(const Coord &from, const Coord &to) {
        cv::line(image, from.toCV(), to.toCV(), cv::Scalar(255, 0, 0));
    }

    template<int index>
    inline float interpolateTextureCoordinate(const TriangleTexture &texture, const glm::
    dvec3 &barycentric) const {
        return static_cast<float>(texture.a[index] * barycentric[0]
                                  + texture.b[index] * barycentric[1]
                                  + texture.c[index] * barycentric[2]);
    }
```

```
195
196     glm::vec2
197     interpolateTexture(const TriangleTexture &texture, const glm::dvec3 &barycentric)
        const {
198         auto u = interpolateTextureCoordinate<0>(texture, barycentric);
199         auto v = interpolateTextureCoordinate<1>(texture, barycentric);
200         return glm::vec2(u, v);
201     }
202
203     void triangleBarycentric(const ExtendedTriangle &t, cv::Mat1d &zBuffer, const MipMap
        <8> &mipMap) {
204         auto &&bbox = t.box();
205         Coord from = bound(bbox.first);
206         Coord to = bound(bbox.second);
207         for (auto x = from.x; x <= to.x; ++x) {
208             for (auto y = from.y; y <= to.y; ++y) {
209                 glm::dvec3 barycentric;
210                 bool pointInTriangle = get_barycentric(t, glm::vec4(x, y, 1, 1),
        barycentric);
211                 if (!pointInTriangle) continue;
212                 glm::dvec3 zInterpolated = glm::dvec3(t.a.z, t.b.z, t.c.z) * barycentric;
213                 double z = zInterpolated[0] + zInterpolated[1] + zInterpolated[2];
214                 if (z < zBuffer.at<double>(y, x)) {
215                     zBuffer.at<double>(y, x) = z;
216
217                     glm::dvec3 barycentricX, barycentricY;
218                     get_barycentric(t, glm::vec4(x + 1, y, 1, 1), barycentricX);
219                     get_barycentric(t, glm::vec4(x, y + 1, 1, 1), barycentricY);
220
221                     glm::vec2 uv = interpolateTexture(t.texture, barycentric);
222                     glm::vec2 uvX = interpolateTexture(t.texture, barycentricX);
223                     glm::vec2 uvY = interpolateTexture(t.texture, barycentricY);
224
225                     glm::vec2 dx = uvX - uv;
226                     glm::vec2 dy = uvY - uv;
227
228                     float l = std::max(glm::dot(dx, dx), glm::dot(dy, dy));
229                     float d = std::max(0.0f, 0.5f * std::log2(l));
230
231                     cv::Vec3b texel = trilinearFilter(uv, d, mipMap);
232                     plot(x, y, texel);
233
234                 }
235             }
236         }
237     }
238
239
240     cv::Mat getImage() const {
241         return image;
242     }
243
244     void resetImage() {
245         image = cv::Mat::zeros(height, width, CV_8UC3);
246     }
247
248     int width, height;
249
250
251 private:
252
253     cv::Vec3b interpolate(const cv::Mat3b &level, const glm::vec2 &xy, const glm::vec2 &
        dxy) const {
254         return (1 - dxy.x) * (1 - dxy.y) * getFrom(level, xy)
255                + dxy.x * (1 - dxy.y) * getFrom(level, xy + glm::vec2(1, 0))
256                + (1 - dxy.x) * dxy.y * getFrom(level, xy + glm::vec2(0, 1))
```

```
257                 + (dxy.x * dxy.y) * getFrom(level, xy + glm::vec2(1, 1));
258
259     }
260
261     cv::Vec3b trilinearFilter(const glm::vec2 &uv, float df, const MipMap<8> &mipmap)
        const {
262         int d = static_cast<int>(std::floor(df));
263         float dd = df - d;
264         glm::vec2 fxy0 = mipmap.scale(d, uv);
265         glm::vec2 fxy1 = mipmap.scale(d + 1, uv);
266
267         glm::vec2 xy0 = glm::floor(fxy0);
268         glm::vec2 xy1 = glm::floor(fxy1);
269
270         glm::vec2 dxy0 = fxy0 - xy0;
271         glm::vec2 dxy1 = fxy1 - xy1;
272
273         cv::Mat3b &&level0 = mipmap.get(d);
274         cv::Mat3b &&level1 = mipmap.get(d + 1);
275
276         cv::Vec3b v0 = interpolate(level0, xy0, dxy0);
277         cv::Vec3b v1 = interpolate(level1, xy1, dxy1);
278
279         return (1 - dd) * v0 + dd * v1;
280     }
281
282     inline int bound(int x, int bnd) {
283         return std::max(0, std::min(bnd - 1, x));
284     }
285
286     inline Coord bound(const Coord &coord) {
287         return Coord{bound(coord.x, width), bound(coord.y, height)};
288     }
289
290     inline double edgeFunction(const glm::vec4 &a, const glm::vec4 &b, const glm::vec4 &c
        ) {
291         return (c.x - a.x) * (b.y - a.y) - (c.y - a.y) * (b.x - a.x);
292     }
293
294     bool get_barycentric(const ExtendedTriangle &t, const glm::vec4 &p, glm::dvec3 &
        barycentric) {
295         auto area = edgeFunction(t.a, t.b, t.c);
296         auto w0 = edgeFunction(t.b, t.c, p) / area;
297         auto w1 = edgeFunction(t.c, t.a, p) / area;
298         auto w2 = edgeFunction(t.a, t.b, p) / area;
299
300         barycentric = glm::dvec3(w0, w1, w2);
301         barycentric /= barycentric[0] + barycentric[1] + barycentric[2];
302         barycentric = glm::abs(barycentric);
303
304         return !(w0 < 0 || w1 < 0 || w2 < 0);
305     }
306
307     inline bool check_coordinates(int x, int y) {
308         if (x < 0) return false;
309         if (x >= width) return false;
310         if (y < 0) return false;
311         if (y >= height) return false;
312         return true;
313     }
314
315     inline void plot(int x, int y, const glm::vec3 &color) {
316         if (!check_coordinates(x, y))
317             return;
318
319         image.at<cv::Vec3b>(y, x) = cv::Vec3b(
```

```
320                 static_cast<uchar>(color[0] * 255),
321                 static_cast<uchar>(color[1] * 255),
322                 static_cast<uchar>(color[2] * 255)
323         );
324     }
325
326     inline void plot(int x, int y, const cv::Vec3b &color) {
327         if (!check_coordinates(x, y))
328             return;
329         image.at<cv::Vec3b>(y, x) = color;
330
331     }
332
333     void line(const Coord &from, const Coord &to, std::vector<Coord> &result) {
334         drawline(from.x, from.y, to.x, to.y, [&](int x, int y) mutable {
335             if (check_coordinates(x, y))
336                 result.emplace_back(Coord{x, y});
337         });
338     }
339
340     void drawline(int x0, int y0, int x1, int y1) {
341         drawline(x0, y0, x1, y1, [&](int x, int y) { plot(x, y, glm::vec3(0.5f)); });
342     }
343
344     template<typename F>
345     inline void drawline(int x0, int y0, int x1, int y1, F plot) {
346
347         int dx = std::abs(x1 - x0);
348         int dy = std::abs(y1 - y0);
349
350         int directionX = x0 < x1 ? 1 : -1;
351         int directionY = y0 < y1 ? 1 : -1;
352         int err = (dx > dy ? dx : -dy) / 2;
353
354         for (;;) {
355             plot(x0, y0);
356             if (x0 == x1 && y0 == y1) break;
357             int e2 = err;
358             if (e2 > -dx) {
359                 err -= dy;
360                 x0 += directionX;
361             }
362             if (e2 < dy) {
363                 err += dx;
364                 y0 += directionY;
365             }
366         }
367     }
368
369     inline void orderVerticesByY(Coord &v1, Coord &v2, Coord &v3) {
370         if (v1.y > v2.y) {
371             std::swap(v1, v2);
372         }
373         if (v2.y > v3.y) {
374             std::swap(v2, v3);
375         }
376         if (v1.y > v2.y) {
377             std::swap(v1, v2);
378         }
379     }
380
381     inline int signum(int x) {
382         if (x == 0) return 0;
383         else if (x > 0) return 1;
384         else return -1;
385     }
```

```
386
387     void fillFlatSideTriangle(const Coord &v1, const Coord &v2, const Coord &v3, std::
        vector<Coord> &trianglePixels) {
388
389         int dx1 = std::abs(v2.x - v1.x);
390         int dy1 = std::abs(v2.y - v1.y);
391
392         int dx2 = std::abs(v3.x - v1.x);
393         int dy2 = std::abs(v3.y - v1.y);
394
395         int signx1 = signum(v2.x - v1.x);
396         int signx2 = signum(v3.x - v1.x);
397
398         int signy1 = signum(v2.y - v1.y);
399         int signy2 = signum(v3.y - v1.y);
400
401         bool edge1Steep = dy1 > dx1;
402         bool edge2Steep = dy2 > dx2;
403
404         if (edge1Steep) {
405             std::swap(dx1, dy1);
406         }
407         if (edge2Steep) {
408             std::swap(dx2, dy2);
409         }
410
411         Coord vTmp1{v1.x, v1.y};
412         Coord vTmp2{v1.x, v1.y};
413
414         int e1 = 2 * dy1 - dx1;
415         int e2 = 2 * dy2 - dx2;
416
417         for (int i = 0; i <= dx1; i++) {
418             line(vTmp1, vTmp2, trianglePixels);
419
420             while (e1 >= 0 && dx1 > 0) {
421                 if (edge1Steep) {
422                     vTmp1.x += signx1;
423                 } else {
424                     vTmp1.y += signy1;
425                 }
426                 e1 = e1 - 2 * dx1;
427             }
428
429             if (edge1Steep) {
430                 vTmp1.y += signy1;
431             } else {
432                 vTmp1.x += signx1;
433             }
434
435             e1 = e1 + 2 * dy1;
436
437             while (vTmp2.y != vTmp1.y) {
438                 while (e2 >= 0 && dx2 > 0) {
439                     if (edge2Steep) {
440                         vTmp2.x += signx2;
441                     } else {
442                         vTmp2.y += signy2;
443                     }
444                     e2 = e2 - 2 * dx2;
445                 }
446
447                 if (edge2Steep) {
448                     vTmp2.y += signy2;
449                 } else {
450                     vTmp2.x += signx2;
```

```
                }

                e2 = e2 + 2 * dy2;
            }

        }
    }


    std::vector<Coord> rasterizeTriangle(Coord v1, Coord v2, Coord v3) {
        std::vector<Coord> trianglePixels;
        orderVerticesByY(v1, v2, v3);
        if (v2.y == v3.y) {
            fillFlatSideTriangle(v1, v2, v3, trianglePixels);
        } else if (v1.y == v2.y) {
            fillFlatSideTriangle(v3, v1, v2, trianglePixels);
        } else {
            float v2mv1 = v2.y - v1.y;
            float v3mv1 = v3.y - v1.y;
            int v4x = static_cast<int>(v1.x + (v2mv1 / v3mv1) * (v3.x - v1.x));
            Coord v4{v4x, v2.y};
            fillFlatSideTriangle(v1, v2, v4, trianglePixels);
            fillFlatSideTriangle(v3, v2, v4, trianglePixels);
        }
        return trianglePixels;
    }


    cv::Mat image;
};
```

1.6.4 Листинг 4. transformers.h

```cpp
#pragma once

#include <any>
#include <utility>
#include <vector>
#include <memory>

#include <OBJ_Loader.h>

#include <glm/vec3.hpp>
#include <glm/geometric.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include "render.h"

class VertexTransformer {
public:
    virtual std::any apply(const std::any &vex) = 0;

    template<typename T, typename R>
    std::vector<R> applyList(const std::vector<T> &vertices) {
        std::vector<R> result;
        for (auto &&v: vertices) {
            result.emplace_back(std::any_cast<R>(apply(v)));
        }
        return result;
    }
};

glm::vec3 to_glm(const objl::Vector3 &vec) {
    return glm::vec3{vec.X, vec.Y, vec.Z};
}

glm::vec2 to_glm(const objl::Vector2 &vec) {
    return glm::vec2{vec.X, vec.Y};
}

class ToGLMVertices : public VertexTransformer {


    std::any apply(const std::any &vertex) override {
        return to_glm(std::any_cast<objl::Vertex>(vertex).Position);
    }
};

class ToGLMNormals : public VertexTransformer {


    std::any apply(const std::any &vertex) override {
        return to_glm(std::any_cast<objl::Vertex>(vertex).Normal);
    }
};

class ToGLMTexture : public VertexTransformer {
    std::any apply(const std::any &vertex) override {
        return to_glm(std::any_cast<objl::Vertex>(vertex).TextureCoordinate);
    }
};

template<typename T, typename R>
class TransformationPipeline {
public:
    virtual R apply(const T &triangle) = 0;
};
```

```
65
66
67  class GLMLineTransformationPipeline : public TransformationPipeline<Triangle, Triangle> {
68
69  public:
70      explicit GLMLineTransformationPipeline(const glm::mat4 &cam, const glm::mat4 &proj,
        int width, int height)
71              : camera(cam), projection(proj), viewport(0, 0, width, height), w(width), h(
        height) {}
72
73      glm::vec3 transformVertex(const glm::vec3 &vertex) {
74          auto &&scaled = glm::project(vertex, camera, projection, viewport);
75          return scaled;
76      }
77
78      Triangle apply(const Triangle &t) override {
79          return {transformVertex(t.a), transformVertex(t.b), transformVertex(t.c)};
80      }
81
82
83  private:
84      glm::mat4 camera, projection;
85      glm::vec4 viewport;
86      int w, h;
87  };
88
89
90  class LineTransformationPipeline : public TransformationPipeline<Triangle, Triangle> {
91  public:
92      explicit LineTransformationPipeline(const glm::mat4 &cam, const glm::mat4 &proj, int
        width, int height)
93              : transformation(proj * cam), w(width), h(height), scale(width, height, 1) {}
94
95      glm::vec3 transformVertex(const glm::vec3 &vertex) {
96          auto homoCoord = glm::vec4(vertex, 1.0);
97          glm::vec4 v = transformation * homoCoord;
98          float w = v[3];
99          glm::vec3 cartesian = glm::vec3(v[0] / w, v[1] / w, v[2] / w);
100         glm::vec3 &&scaled = ((cartesian + 1.0f) / 2.0f) * scale;
101         return scaled;
102     }
103
104     Triangle apply(const Triangle &t) override {
105         return {transformVertex(t.a), transformVertex(t.b), transformVertex(t.c)};
106     }
107
108 private:
109
110     glm::mat4 transformation;
111     glm::vec3 scale;
112     int w, h;
113 };
114
115
116 class TriangleTransformationPipeline : public TransformationPipeline<Triangle,
        ExtendedTriangle> {
117 public:
118     explicit TriangleTransformationPipeline(const glm::mat4 &cam, const glm::mat4 &proj,
        int width, int height)
119             : transformation(proj * cam), w(width), h(height), scale(width, height, 1) {}
120
121     glm::vec4 transformVertex(const glm::vec3 &v) {
122
123         auto &&homoCoord = glm::vec4(v, 1.0);
124         glm::vec4 &&res = transformation * homoCoord;
125         float w = res[3];
```

```
126         glm::vec3 &&cartesian = glm::vec3(res.x / w, res.y / w, res.z / w);
127         glm::vec3 &&scaled = ((cartesian + glm::vec3(1.0f, 1.0f, 0.0f)) / glm::vec3(2.0f,
        2.0f, 1.0f)) * scale;
128         return glm::vec4(scaled, w);
129     }
130
131     ExtendedTriangle apply(const Triangle &triangle) override {
132         ExtendedTriangle tri{transformVertex(triangle.a), transformVertex(triangle.b),
        transformVertex(triangle.c)};
133         tri.texture = triangle.texture;
134         tri.normal = triangle.normal;
135         return tri;
136     }
137
138 private:
139
140     glm::mat4 transformation;
141     glm::vec3 scale;
142     int w, h;
143 };
```