

Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Операционные Системы и Системное Программирование
(ОСиСП)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему:

«Визуализатор 3D объектов»

БГУИР КП 1-40 01 01 612 ПЗ

Студент: гр. 851006 Ковалевский М. Ю.

Руководитель: Жиденко А. Л.

Минск 2020

Учреждение образования
«Белорусский государственный университет информатики и
радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ

Заведующий кафедрой ПОИТ

(подпись)

Лапицкая Н.В. 2020 г.

ЗАДАНИЕ

по курсовому проектированию

Студенту Ковалевскому Михаилу Юрьевичу

1. Тема работы «Визуализатор 3D объектов»
2. Срок сдачи студентом законченной работы 01.12.2020
3. Исходные данные к работе Документация по WinApi
4. Содержание расчётно-пояснительной записки (перечень вопросов, которые подлежат разработке)

Введение.

1. Анализ прототипов, литературных источников и формирование требований к проектируемому ПС;
 2. Разработка алгоритма;
 3. Разработка программного средства;
 4. Тестирование, экспериментальные исследования и анализ полученных результатов;
 5. Руководство пользователя программы;
- Заключение, список литературы, ведомость, приложения.

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

1. Схема программы на А1

6. Консультант по курсовому проекту Жиденко А. Л.

7. Дата выдачи задания 05.09.2020 г.

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и процентом от общего объема работы):

раздел 1 к 20.09.2020 – 15 % готовности работы;

разделы 2, 3 к 13.10.2020 – 30 % готовности работы;

раздел 4 к 02.11.2020 – 60 % готовности работы;

раздел 5 к 26.11.2020 – 90 % готовности работы;

оформление пояснительной записки и графического материала к 01.12.2020 – 100 % готовности работы. Защита курсового проекта с 01.12 по 22.12 2020 г.

РУКОВОДИТЕЛЬ _____ Жиденко А. Л.
(подпись)

Задание принял к исполнению Ковалевский М.Ю _____ 05.09.2020 г.
(дата и подпись студента)

СОДЕРЖАНИЕ

Введение	5
1 Анализ предметной области	6
1.1 Обзор аналогов	6
1.2 Постановка задачи.....	9
2 Разработка программного средства.....	10
2.1 Структура программы.....	10
2.2 Интерфейс программного средства.....	10
2.3 Основной цикл программы	11
2.4 Обработка нажатий	12
2.5 Структура 3D объекта в формате obj	12
2.6 Камера в OpenGL	14
2.7 Алгоритмы OpenGL	15
3 Тестирование программного средства.....	20
4 Руководство пользователя.....	22
4.1 Основные требования для запуска	22
4.2 Руководство по установке	22
4.3 Руководство по использованию.....	22
Заключение	25
Список использованных источников	26
Приложение 1	27

ВВЕДЕНИЕ

Визуализатор – программа, позволяющая просматривать цифровые изображения на компьютере. Он может быть нацелен как на плоские изображения, так и на объемные плоские.

Трёхмерная графика — раздел компьютерной графики, посвящённый методам создания изображений или видео путём моделирования объёмных объектов в трёхмерном пространстве.

Графическое изображение трёхмерных объектов отличается тем, что включает построение геометрической проекции трёхмерной модели сцены на плоскость (например, экран компьютера) с помощью специализированных программ.

Визуальные эффекты, приемы совмещения компьютерной графики с реальным видео поражают воображение и вызывают у многих интерес к трехмерному моделированию и анимации.

Трёхмерная графика активно применяется для создания изображений на плоскости экрана или листа печатной продукции в науке и промышленности, например, в системах автоматизации проектных работ (САПР; для создания твердотельных элементов: зданий, деталей машин, механизмов), архитектурной визуализации (сюда относится и так называемая «виртуальная археология»), в современных системах медицинской визуализации.

Самое широкое применение — во многих современных компьютерных играх, а также как элемент кинематографа, телевидения, печатной продукции.

Трёхмерная графика обычно имеет дело с виртуальным, воображаемым трёхмерным пространством, которое отображается на плоской, двухмерной поверхности дисплея или листа бумаги. В настоящее время известно несколько способов отображения трёхмерной информации в объёмном виде, хотя большинство из них представляет объёмные характеристики весьма условно, поскольку работают со стереоизображением. Из этой области можно отметить стереочки, виртуальные шлемы, 3D-дисплеи, способные демонстрировать трёхмерное изображение.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор аналогов

Существует ряд программ для визуализации 3D объектов, каждая из них имеет свои преимущества и недостатки.

Хорошим выбором для пользователей Windows станет Microsoft 3D Viewer [1]. Она позволяет открыть и отобразить 3D модели в большинстве популярных на сегодняшний день форматах. Программа представлена на рисунке 1.1.



Рисунок 1.1 – Программа Microsoft 3D Viewer

Microsoft 3D Viewer поддерживает fbx, 3mf, obj, stl и многие другие форматы файлов. Преимуществом данной программы является то, что она бесплатна и встроена в операционную систему Windows. Microsoft 3D Viewer имеет интуитивно понятный интерфейс, благодаря чему любой пользователь без каких-либо проблем сможет разобраться и использовать программу. Возможно управление камерой с помощью клавиатуры и мыши. Программа позволяет применить к модели различные материалы, создать несколько источников света и настроить их интенсивность, угол и цвет, добавить фон к просматриваемой модели. Основным минусом является доступность программы только на операционной системе Windows.

Хорошим выбором для пользователей MacOS станет Apple Preview [2]. Она создавалась не исключительно для просмотра 3D объектов. Основным функционалом является просмотр картинок, видео, текста, презентаций, но также присутствует возможность работы с 3D. Программа представлена на рисунке 1.2.

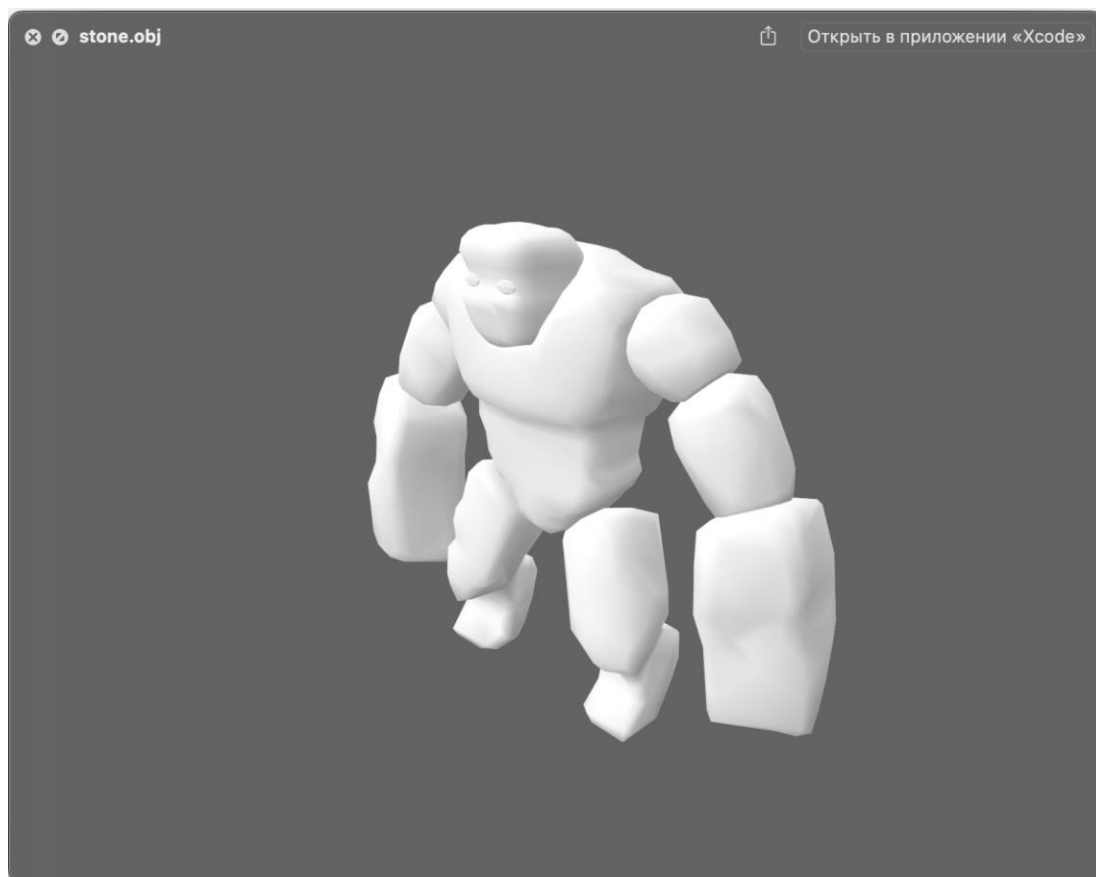


Рисунок 1.2 – Программа Apple Preview

Apple Preview позволяет открыть и отобразить 3D модели в таких форматах как obj, dae, stl и других. Преимуществом данной программы является то, что она бесплатна и встроена в операционную систему MacOS. Preview не имеет богатого пользовательского интерфейса. Управление камерой интуитивно понятно и выполнять при помощи мыши. Присутствует возможность создавать предпросмотр для абсолютно любого файла. Если файл поддерживается программой, то он будет открыт с минимальным интерфейсом для ознакомления с ним, если же не поддерживается, то будет выведена основная информация о нем. Минусом является доступность только на MacOS и невозможность почти никак управлять 3D сценой.

Программа MadView3D - это легкий просмотрщик объектов, совместимый с Windows, Linux, Mac OS, Raspberry Pi и Tinker Board [3]. Она представлена на рисунке 1.3.

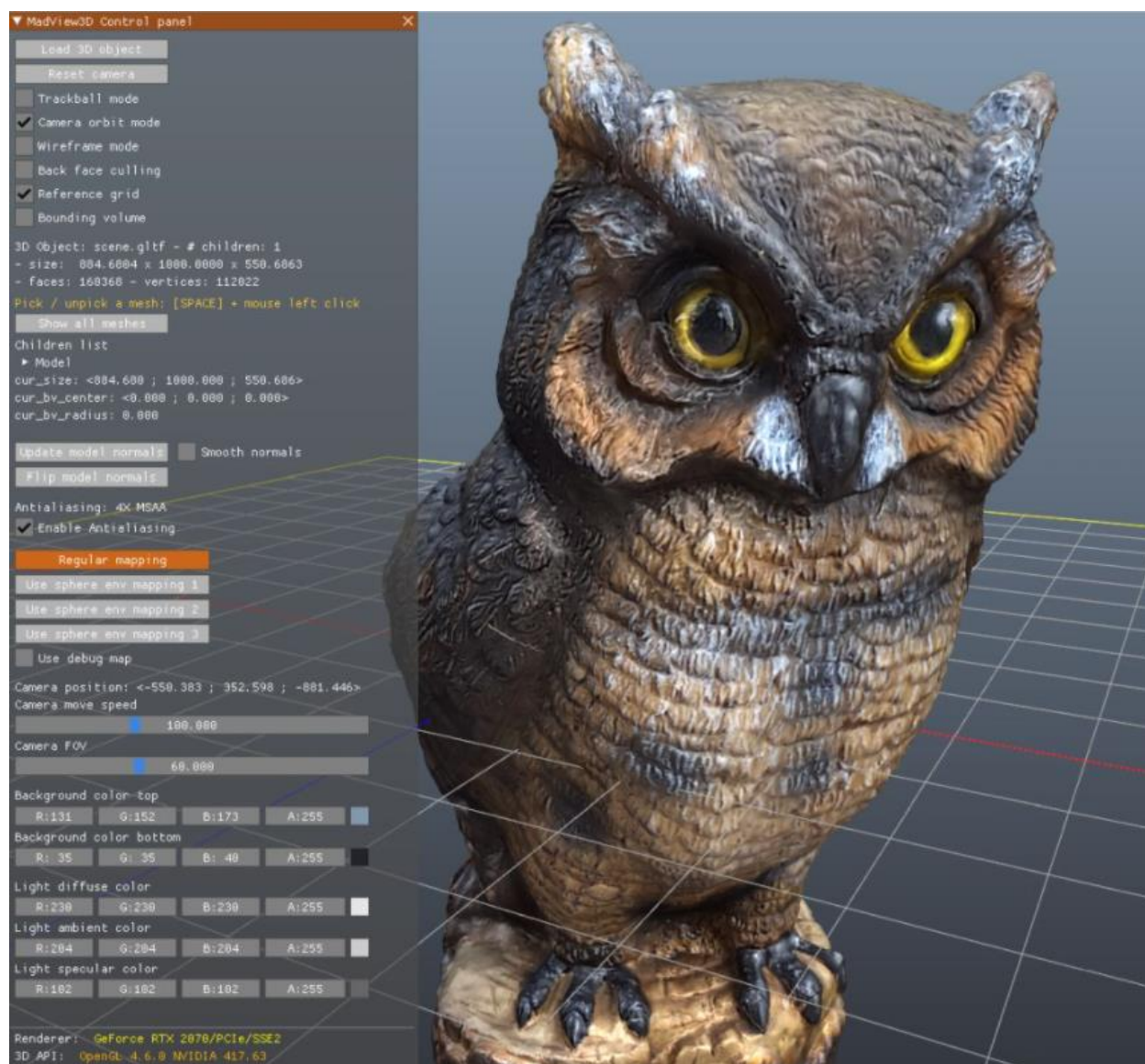


Рисунок 1.3 – Программа MADView3D

MadView3D поддерживает множество форматов 3D файлов, в том числе .3ds, .obj, .fbx, .gltf, и многое другое. Для просмотра файла нужно просто перетащить его на сцену в Mad View 3D. Перемещение камеры происходит с помощью привычных элементов управления мышью. Преимуществом является то, что сцену можно гибко настроить. Также на экран выводится огромное количество информации, связанной с открытым объектом.

1.2 Постановка задачи

В рамках данного курсового проекта планируется разработка программного средства «Визуализатор 3D объектов».

Будут разработаны алгоритмы построения 3D сцены, размещения света в различных позициях, перемещения и поворота камеры, масштабирования объектов к одному размеру по высоте.

В программном средстве планируется реализовать следующие функции и алгоритмы:

- открытие объекта из файла obj;
- добавление нескольких объектов на сцену;
- редактирование параметров света;
- редактирование параметров камеры;
- рендеринг wireframe;
- рендеринг solid;

Главной задачей является создать удобную программу для просмотра 3D моделей.

Для разработки программного средства будет использоваться язык программирования C++, библиотека OpenGL и операционная система Windows 10.

2 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

2.1 Структура программы

Требуется использовать три структурных блока:

- main – модуль, отвечающий за инициализацию OpenGL в среде ОС Windows;
- engine – модуль, отвечающий за рендеринг объектов на сцене и обработку нажатий пользователя без задержки;
- scene – модуль, отвечающий за расположение объектов в 3D пространстве.

2.2 Интерфейс программного средства

Внешний вид и удобность в использовании являются одними из главных критериев качества программного средства. Поэтому взаимодействие приложения с пользователем необходимо организовать максимально просто.

Интерфейс получился компактным и показан на рисунке 2.1.

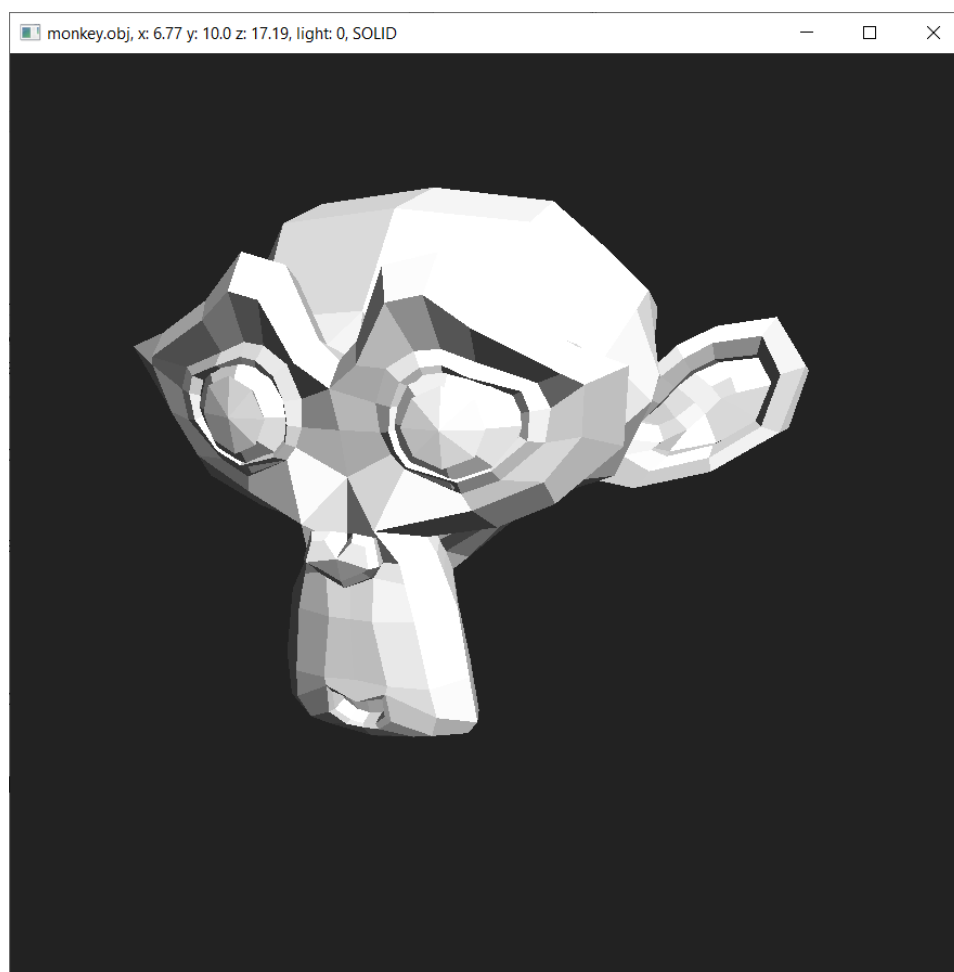


Рисунок 2.1 – Интерфейс программы

При открытии 3D объекта он полностью занимает окно. В заголовке окна выводить основная информация о 3D сцене: названия открытых объектов, текущая позиция камеры, положение света и тип рендеринга.

2.3 Основной цикл программы

Основной цикл программы отличается от стандартного приложения на Windows API. Он представлен на рисунке 2.2.

```
bool bQuit = false;
/* program main loop */
while (!bQuit) {
    /* check for messages */
    if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) {
        /* handle or dispatch messages */
        if (msg.message == WM_QUIT) {
            bQuit = true;
        } else {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    } else {
        /* OpenGL animation code goes here */
        engine.handleKeys(); // Faster than handling messages
        engine.draw(hDC);
        SetWindowTextA(hwnd, engine.description().data());
        Sleep( dwMilliseconds: 1000 / engine.fps);
    }
}
```

Рисунок 2.2 – Основной цикл программы

Основное отличие заключается в том, что в стандартном приложении на Windows API обработка всех событий сопровождается сообщениями, которые обрабатываются в соответствующей функции.

Для отрисовки окна обычно используется сообщение WM_PAINT. Но в данном случае требуется рисовать на окне максимально часто с использованием функций OpenGL. Для того, чтоб добиться этого потребуются отрисовывать сцену не при получении сообщения MW_PAINT, а в те моменты, когда нет никаких других важных сообщений на обработку и есть ресурсы на просчет сцены.

Также указывается желаемая частота кадров для уменьшения потребления программой компьютерных ресурсов.

2.4 Обработка нажатий

При нажатии клавиши в программу приходит сообщение WM_KEYDOWN вместе с информацией о нажатой клавише. В данном случае если зажать клавишу нажатие обрабатывается 1 раз и запускается таймер на проверку того, что клавиша зажата. После срабатывания таймера происходит проверка, зажата ли она, если зажата, то приходит множество сообщений о нажатии этой же клавиши. Данный способ подходит для обработки функций, требующих одно нажатие, например, открытие меню или переключение режимов. Так как если бы не было таймера, то клавиша нажимала бы несколько раз и выполнить действие получалось бы не всегда.

В программе данный способ используется для открытия окна выбора файла, изменения положения света, изменения типа рендеринга, сброса параметров камеры и сброса очистки сцены.

Но некоторые действия требуют, чтобы данной задержки от таймера не было. Например, передвижение камеры стоит выполнять без задержек, так как задержки уничтожают плавность программы. Обработку таких клавиш следует проводить вместе с отрисовкой сцены, то есть постоянно.

В основном цикле программы присутствует вызов `engine.handleKeys()`, в котором происходит проверка, нажата ли клавиша, не дожидаясь сообщения о нажатии.

2.5 Структура 3D объекта в формате obj [5]

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.

Файл `obj` не содержит никакой информации о материалах, применяемых к объекту. Для этого используются отдельные файлы формата `MTL`.

В файле используются следующие основные обозначения:

- `v` для определения координаты вершины;
- `vt` для определения связи координаты и текстуры, находящейся в файле `MTL`;
- `vn` для определения направления нормали в каждой вершине;
- `f` для определения полигонов;

В файле обязательно должна присутствовать информация о вершинах и полигонах. Информация о текстурах и нормалях может отсутствовать, так как не к каждому объекту есть текстура, а нормали можно высчитывать исходя из того, какая последовательность точек в полигонах.

Полигоны могут состоять из любого количества вершин, но принято использовать полигоны из 3х или 4х вершин.

Пример куба в формате obj представлен на рисунке 2.3

```
1 # Blender v2.81 (sub 16) OBJ File: ''
2 # www.blender.org
3 mtllib untitled.mtl
4 o Cube
5 v 1.000000 1.000000 -1.000000
6 v 1.000000 -1.000000 -1.000000
7 v 1.000000 1.000000 1.000000
8 v 1.000000 -1.000000 1.000000
9 v -1.000000 1.000000 -1.000000
10 v -1.000000 -1.000000 -1.000000
11 v -1.000000 1.000000 1.000000
12 v -1.000000 -1.000000 1.000000
13 vt 0.625000 0.500000
14 vt 0.875000 0.500000
15 vt 0.875000 0.750000
16 vt 0.625000 0.750000
17 vt 0.375000 0.750000
18 vt 0.625000 1.000000
19 vt 0.375000 1.000000
20 vt 0.375000 0.000000
21 vt 0.625000 0.000000
22 vt 0.625000 0.250000
23 vt 0.375000 0.250000
24 vt 0.125000 0.500000
25 vt 0.375000 0.500000
26 vt 0.125000 0.750000
27 vn 0.0000 1.0000 0.0000
28 vn 0.0000 0.0000 1.0000
29 vn -1.0000 0.0000 0.0000
30 vn 0.0000 -1.0000 0.0000
31 vn 1.0000 0.0000 0.0000
32 vn 0.0000 0.0000 -1.0000
33 usemtl Material
34 s off
35 f 1/1/1 5/2/1 7/3/1 3/4/1
36 f 4/5/2 3/4/2 7/6/2 8/7/2
37 f 8/8/3 7/9/3 5/10/3 6/11/3
38 f 6/12/4 2/13/4 4/5/4 8/14/4
39 f 2/13/5 1/1/5 3/4/5 4/5/5
40 f 6/11/6 5/10/6 1/1/6 2/13/6
```

Рисунок 2.3 – Пример куба в формате obj

В данном примере файл содержит информацию о текстурах, что делает потенциально возможным наложение текстур на куб.

Также файл содержит информацию о нормалях. В данном случае ее можно было не включать в файл и сэкономить место. Но программа, открывающая файл, должна уметь генерировать нормали автоматически. Также при генерации нормалей в автоматическом режиме не всегда получается нужный результат, так как вершины могут быть записаны не в том порядке, в следствии чего освещение будет работать неправильно.

Полигоны фигуры состоят из 4 вершин в то время, как в 3D мире всё состоит из треугольников. При открытии файла данные полигоны будут разбиты на треугольники таким образом, чтобы внешний вид фигуры не изменился никак. Также следует учитывать, что нормали и точки текстур записаны для полигонов из 4х вершин.

2.6 Камера в OpenGL [5]

Когда мы говорим о пространстве камеры/вида, мы подразумеваем вид всех вершин с точки зрения камеры, положение которой в этом пространстве является базовой точкой начала координат: матрица вида трансформирует мировые координаты в координаты вида, измеряющиеся относительно расположения и направления камеры.

В OpenGL нет такого объекта как камера. Просмотр всегда происходит из точки $(0, 0, 0)$ по отрицательному направлению оси Z. Для имитации камеры вся сцена принимает параметры, противоположные параметрам камеры, благодаря чему она начинает выглядеть так, как будто мы смотрим из какого-то определенного места.

Для однозначного математического описания камеры, нам необходимо ее положение в мировом пространстве, направление в котором она смотрит, вектор указывающий правое направление, и вектор указывающий направление вверх.

Описание камеры показано на рисунке 2.4.

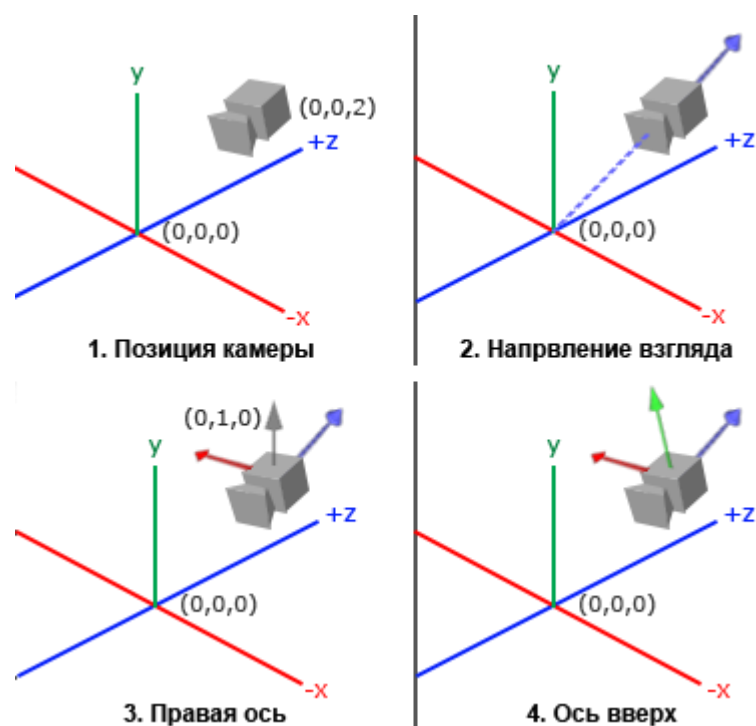


Рисунок 2.4 – Камера в OpenGL

Как пример, для имитации камеры по координатам $(1, 5, 10)$, нам требуется переместиться в точку $(-1, -5, -10)$ и нарисовать всю сцену в данной точке. После чего будет казаться, что камера находится в нужной точке, а не зафиксирована в $(0, 0, 0)$.

2.7 Алгоритмы OpenGL

Для создания проекции 3D мира на 2D экран используются 4х мерные матрицы и 3х мерные векторы. А также алгоритмы векторной алгебры: умножения матрицы на матрицу, вектора на вектор, построение матриц поворота, матриц проекции, матриц пространства в мировых и локальных координатах. Для раскраски полигонов используется алгоритм Scanline. Для правильной объемной отрисовки также используется буфер глубины. Для создания освещения используется алгоритм сравнения угла поворота полигона с углом поворота камеры.

Алгоритм умножения матрицы на матрицу на рисунке 2.5 [4].

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32} & a_{11} \cdot b_{13} + a_{12} \cdot b_{23} + a_{13} \cdot b_{33} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + a_{23} \cdot b_{32} & a_{21} \cdot b_{13} + a_{22} \cdot b_{23} + a_{23} \cdot b_{33} \\ a_{31} \cdot b_{11} + a_{32} \cdot b_{21} + a_{33} \cdot b_{31} & a_{31} \cdot b_{12} + a_{32} \cdot b_{22} + a_{33} \cdot b_{32} & a_{31} \cdot b_{13} + a_{32} \cdot b_{23} + a_{33} \cdot b_{33} \end{pmatrix}$$

Рисунок 2.5 – Алгоритм умножения матрицы на матрицу

Алгоритм умножения вектора на вектор представлен на рисунке 2.6 [4].

$$(a \ b \ c \ d) \cdot \begin{pmatrix} s \\ x \\ y \\ z \end{pmatrix} \rightarrow a \cdot s + b \cdot x + c \cdot y + d \cdot z$$

$$\begin{pmatrix} s \\ x \\ y \\ z \end{pmatrix} \cdot (a \ b \ c \ d) \rightarrow \begin{pmatrix} a \cdot s & s \cdot b & s \cdot c & s \cdot d \\ a \cdot x & b \cdot x & x \cdot c & d \cdot x \\ y \cdot a & b \cdot y & c \cdot y & y \cdot d \\ z \cdot a & z \cdot b & c \cdot z & d \cdot z \end{pmatrix}$$

Рисунок 2.6 – Алгоритм умножения вектора на вектор

Матрица проекций отвечает за то, какой объём пространства будет визуализироваться, каким образом вершины графических примитивов будут спроецированы на двумерную поверхность экрана монитора. Преобразования матрицы проекций ведут к тому, что все изображение будет изменяться (масштабироваться, перемещаться или вращаться). Возможно использование двух режимов матрицы проекций: перспективная и ортогографическая.

При перспективной проекции используется тот факт, что для человеческого глаз работает с предметом дальнего типа, размеры которого имеют угловые размеры. Чем дальше объект, тем меньше он нам кажется. Таким образом, объём пространства, который визуализируется представляет собой пирамиду [4].

$$T_{\text{резперсп}}^{2\text{точ}} = T_{\text{пр}} \cdot T_{\text{перенос}} \cdot T_{\text{персп}} \cdot T_{\text{орт}} \quad (18)$$

Перемножив последовательно матрицы соотношения (18), получим требуемый результат:

$$T_{\text{резперсп}}^{2\text{точ}} = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l_x & m_y & n_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & r \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & 0 & -r\sin\theta \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & 0 & r\cos\theta \\ l_x & m_y & 0 & n_z r + 1 \end{bmatrix} \quad (19)$$

Рисунок 2.7 – Алгоритм построения матрицы проекции

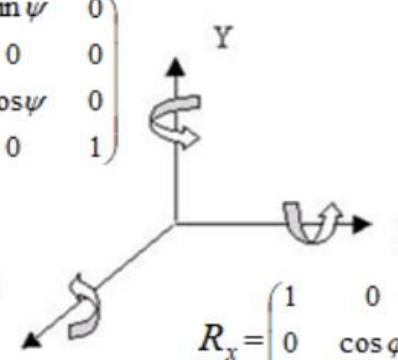
Матрицей поворота (или матрицей направляющих косинусов) называется ортогональная матрица, которая используется для выполнения собственного ортогонального преобразования в евклидовом пространстве. При умножении любого вектора на матрицу поворота длина вектора сохраняется. Определитель матрицы поворота равен единице.

Обычно считают, что в отличие от матрицы перехода при повороте системы координат (базиса), при умножении на матрицу поворота вектора-столбца координаты вектора преобразуются в соответствии с поворотом самого вектора (а не поворотом координатных осей; то есть при этом координаты повернутого вектора получаются в той же, неподвижной системе координат). Однако отличие той и другой матрицы лишь в знаке угла поворота, и одна может быть получена из другой заменой угла поворота на противоположный; та и другая взаимно обратны и могут быть получены друг из друга транспонированием.

Любое вращение в трёхмерном пространстве может быть представлено как композиция поворотов вокруг трёх ортогональных осей (например, вокруг осей декартовых координат). Этой композиции соответствует матрица, равная произведению соответствующих трёх матриц поворота [4].

Алгоритм построения матрицы поворота показана на рисунке 2.8

Матрицы поворота



$$R_y = \begin{pmatrix} \cos \psi & 0 & \sin \psi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \psi & 0 & \cos \psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z = \begin{pmatrix} \cos \chi & -\sin \chi & 0 & 0 \\ \sin \chi & \cos \chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \varphi & -\sin \varphi & 0 \\ 0 & \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Рисунок 2.8 – Алгоритм построения матриц поворота

Мировая матрица – позволяет производить различные матричные преобразования (трансформацию и масштабирование) объекта в мировой системе координат. Мировая система координат – это своя локальная система координат данного объекта, которой наделяется каждый объект, скажем так прошедший через мировую матрицу, поскольку каждая вершина участвует в произведении этой матрицы.

Новая локальная система координат значительно упрощает аффинные преобразования объекта в пространстве. Например, чтобы перенести объект с левого верхнего угла дисплея в нижний правый угол дисплея, то есть переместить игровой объект в пространстве, необходимо просто перенести его локальную точку отсчета, системы координат на новое место. Если бы не было мировой матрицы, то этот объект пришлось переносить по одной вершине. Поэтому любой объект, а точнее все вершины этого объекта проходят через мировую матрицу преобразования [4].

Алгоритм построения матриц пространства в мировых и локальных координатах показан на рисунке 2.9

$$R = \begin{bmatrix} n_1^2 + (1 - n_1^2) \cos \theta & n_1 n_2 (1 - \cos \theta) + n_3 \sin \theta & n_1 n_3 (1 - \cos \theta) - n_2 \sin \theta & 0 \\ n_1 n_2 (1 - \cos \theta) - n_3 \sin \theta & n_2^2 + (1 - n_2^2) \cos \theta & n_2 n_3 (1 - \cos \theta) + n_1 \sin \theta & 0 \\ n_1 n_3 (1 - \cos \theta) + n_2 \sin \theta & n_2 n_3 (1 - \cos \theta) - n_1 \sin \theta & n_3^2 + (1 - n_3^2) \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} n_1 n_3 (1 - \cos \theta) - n_2 \sin \theta & 0 \\ \rightarrow n_2 n_3 (1 - \cos \theta) + n_1 \sin \theta & 0 \\ n_2^2 + (1 - n_2^2) \cos \theta & 0 \\ 0 & 1 \end{bmatrix}.$$

Рисунок 2.9 – Алгоритм построения мировой матрицы

Алгоритм раскраски “Scanline” один из методов рендеринга в компьютерной графике, при котором сцена строится на основе замеров пересечения лучей с визуализируемой поверхностью.

Рейкастинг не является синонимом к рейтрейсингу (трассировке лучей), но он может быть представлен как сокращённая и существенно более быстрая версия алгоритма трассировки лучей. Оба алгоритма являются «image order» и используются в компьютерной графике для рендеринга трёхмерных сцен на двухмерный экран с помощью проекционных лучей, которые проецируются от глаз наблюдателя к источнику света. Метод бросания лучей не вычисляет новые тангенсы лучей света, которые возникнут после того, когда луч, который проецируется от глаза к источнику света, пересечётся с поверхностью. Эта особенность делает невозможным точный рендеринг отражений, преломлений и естественной проекции теней с помощью рейкастинга. Однако все эти особенности могут быть добавлены с помощью «фальшивых» (обманных, аппроксимационных) методик, например, через использование текстурных карт или другие методы. Высокая скорость вычисления сделала рейкастинг удобным методом рендеринга в ранних компьютерных играх с трёхмерной графикой реального времени.

В реальной природе источник света испускает луч света, который, «путешествуя» по пространству, в конечном счёте «натыкается» на какую-либо преграду, которая прерывает распространение этого светового луча. Луч света можно представить в виде потока фотонов, который движется вдоль вектора луча. В какой-либо точке пути с лучом света может случиться любая комбинация трёх вещей: поглощение, отражение и преломление. Поверхность может отразить весь световой луч или только его часть в одном или нескольких направлениях. Поверхность может также поглотить часть светового луча, что приводит к потере интенсивности отраженного и/или преломлённого луча. Если поверхность имеет свойства прозрачности, то она преломляет часть светового луча внутри себя и изменяет его направление распространения, поглощая некоторый (или весь) спектр луча (и, возможно, изменяя цвет). Суммарная интенсивность светового луча, которая была «потеряна» вследствие поглощения, преломления и отражения, должна быть в точности равной исходящей (начальной) интенсивности этого луча. Поверхность не может, например, отразить 66% входящего светового луча, и преломить 50%, так как сумма этих порций будет равной 116%, что больше 100%. Отсюда вытекает, что отраженные и/или преломлённые лучи должны «стыкаться» с другими поверхностями, где их поглощающие, отражающие и преломляющие способности снова вычисляются, основываясь на результатах вычислений входящих лучей. Некоторые из лучей, сгенерированных источником света, распространяются по пространству и, в конечном счете, попадают на область просмотра (глаз человека, объектив фото- или видеокамеры и т.д.). Попытка симулировать физический процесс распространения света путём трассировки световых лучей, используя

компьютер, является чрезмерно расточительной, так как только незначительная доля лучей, сгенерированных источником света, попадает на область просмотра [5].

Алгоритм Scanline представлен на рисунке 2.10.

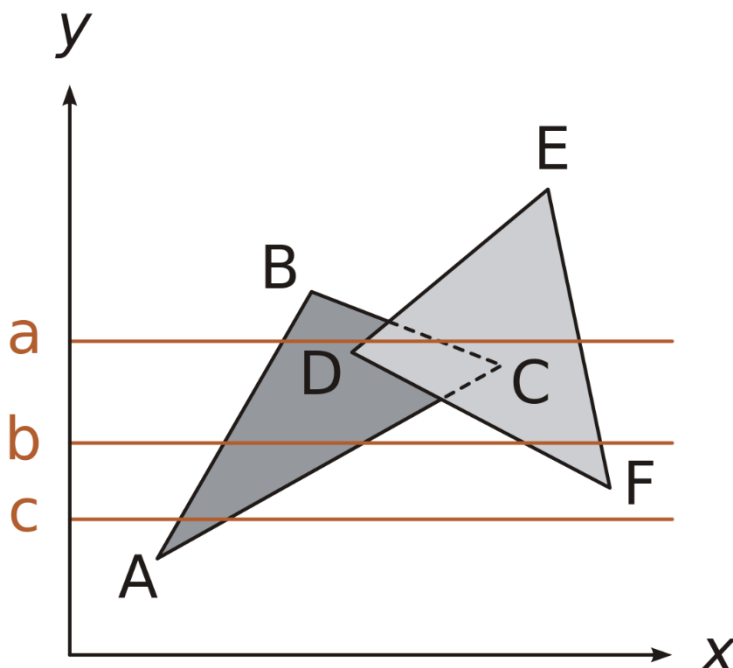


Рисунок 2.10 – Алгоритм Scanline

Буфер глубины (Z-буфер) представляет собой двумерный массив, каждый элемент которого соответствует пикселю на экране. Когда видеокарта рисует пиксель, его удалённость просчитывается и записывается в ячейку Z-буфера. Если пиксели двух рисуемых объектов перекрываются, то их значения глубины сравниваются, и рисуется тот, который ближе, а его значение удалённости сохраняется в буфер. Получаемое при этом графическое изображение носит название z-depth карта, представляющая собой полутоновое графическое изображение, каждый пиксель которого может принимать до 256 значений серого. По ним определяется удалённость от зрителя того или иного объекта трехмерной сцены [5].

3 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

В ходе тестирования приложения не было выявлено недостатков программного средства. Была составлена таблица 3.1, показывающая ожидаемые и реальные результаты, полученные при заданных условиях, она представлена ниже.

Таблица 3.1 – Ожидаемые и реальные результаты тестирования

№	Тестовые случаи	Ожидаемый результат	Полученный результат
1.	Запуск программы	Успешный запуск и инициализация начальных значений параметров программы	Тест пройден
2.	Открытие файла obj с полной информацией о фигуре	Успешное отображение фигуры на экране	Тест пройден
3.	Открытие файла obj с информацией только о вершинах и полигонах	Успешное отображение фигуры на экране	Тест пройден
4.	Выключение программы	Успешное выключение программы без ошибок	Тест пройден
5.	Перемещение камеры по оси X	Камера изменила координату X	Тест пройден
6.	Перемещение камеры по оси Y	Камера изменила координату Y	Тест пройден
7.	Перемещение камеры по оси Z	Камера изменила координату Z	Тест пройден
8.	Поворот камеры относительно оси X	Камера изменила угол эйлера относительно оси X	Тест пройден
9.	Поворот камеры относительно оси Y	Камера изменила угол эйлера относительно оси Y	Тест пройден
10.	Поворот камеры относительно оси Z	Камера изменила угол эйлера относительно оси Z	Тест пройден

Продолжение таблицы 3.1.

11	Сброс параметров камеры	Камера вернулась в исходное положение	Тест пройден
12	Сброс состояния сцены	Все объекты пропали со сцены и камера приняла исходное положение	Тест пройден
13	Изменение положения источника света	Освещение изменилось	Тест пройден
14	Включение wireframe рендеринга	Сцена отрисовалась в стиле wireframe	Тест пройден
15	Включение solid рендеринга	Сцена отрисовалась в стиле solid	Тест пройден

Разработка приложения велась с использованием системы контроля версий GitHub, позволившая сохранять состояние программы на каждом отдельном этапе по ходу добавления нового функционала или изменения уже существующего. Появление новых точек возврата происходит посредством группировки изменённых файлов, затем они объединяются под общим именем «коммита», в котором кратко изложена суть изменений. Также можно добавлять к каждому этапу новые файлы, или удалять устаревшие варианты. После накопления определённого количества групп изменений, их следует отправить на удалённый репозиторий, где видна вся история приложения и разница между каждым новым «коммитом».

Путем тщательной проверки тестами, было выявлено, что ошибок в работе программы нет.

4 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

4.1 Основные требования для запуска

- ОС: Версия Microsoft Windows от 8 и выше;
- Процессор: не менее 1 ГГц;
- RAM: От 1 Гб;
- Место на диске: от 3 Мб свободного места.

Исходя из данных требований следует, что данная программа может запускаться практически на любом компьютере.

4.2 Руководство по установке

Установка программы не требуется, т.к она является переносимым приложением.

Переносимое приложение — программное обеспечение, которое для своего запуска не требует процедуры установки и может полностью храниться на съёмных носителях информации, что позволяет использовать данное ПО на многих компьютерах.

4.3 Руководство по использованию

1) Запустить программу

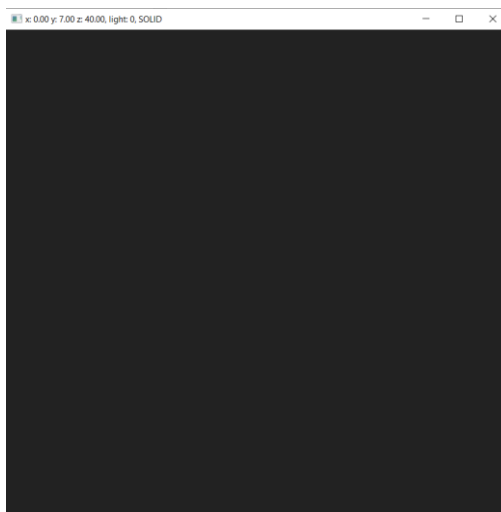


Рисунок 4.1 – Скриншот запущенной программы

2) Открыть несколько объектов (клавиша O)

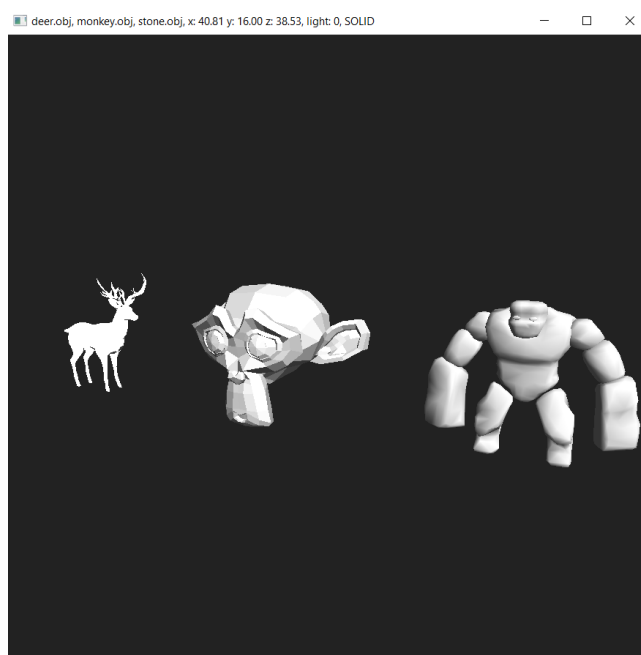


Рисунок 4.2 – Скриншот открытых фигур

3) Изменение положения света (клавиша E)



Рисунок 4.3 – Скриншот открытых фигур с измененным светом

4) Изменение типа рендеринга (клавиша Q)

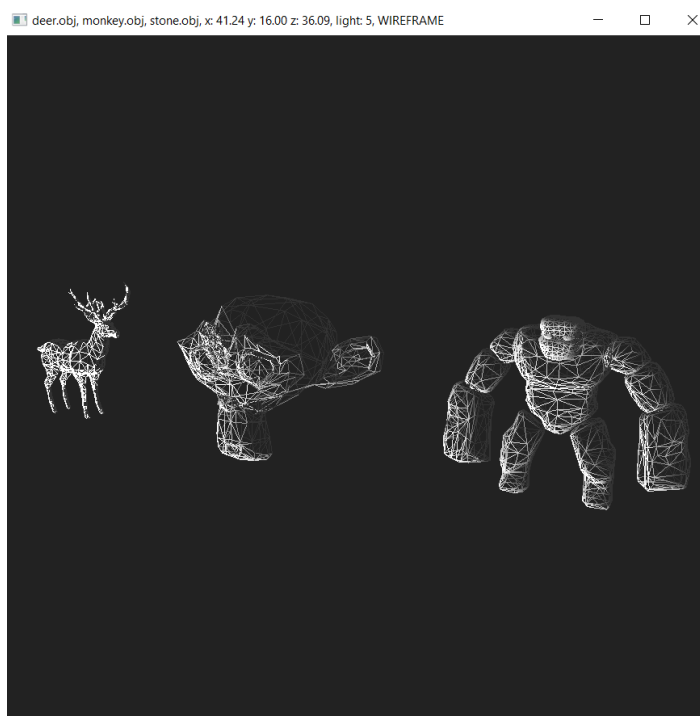


Рисунок 4.4 – Скриншот сцены с wireframe рендерингом

5) Закрывать программу

ЗАКЛЮЧЕНИЕ

В настоящее время вычислительные мощности компьютеров растут невероятно быстро. Благодаря столь быстрому росту компьютеры обрели возможность выйти за рамки 2D пространства. Сегодня самый простой персональный компьютер способен на несложные 3D вычисления. Мощные же компьютеры способны проводить космические симуляции с высокой точностью.

У людей появилась возможность создавать трехмерные объекты, которые могут быть использованы в кино и играх, а также могут быть перенесены в реальный мир с помощью 3D принтера.

В рамках данного курсового проекта было разработано программное средство, позволяющее просматривать 3D фигуры.

Для воплощения проекта в жизнь потребовалось изучить основы компьютерной графики и векторную алгебру.

Существует много возможностей для дальнейшего развития приложения. Некоторыми из них являются добавление функции изменения фона сцены, отрисовка текстур, изменение цвета света и добавление дополнительных форматов 3D-моделей.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] www.microsoft.com [Электронный портал]. – Электронные данные. – Режим доступа: <https://www.microsoft.com/>
- [2] www.apple.com [Электронный портал]. – Электронные данные. – Режим доступа: <https://www.apple.com/>
- [3] www.geeks3d.com [Электронный портал]. – Электронные данные. – Режим доступа: <https://www.geeks3d.com/madview3d/>
- [4] www.matesha.ru [Электронный портал]. – Электронные данные. – Режим доступа: <https://www.matesha.ru/3dgraphics/>
- [5] www.khronos.org [Электронный портал]. – Электронные данные. – Режим доступа: <https://www.khronos.org/opengl/>

ПРИЛОЖЕНИЕ 1

Исходный код программы

Файл main.cpp

```
#ifndef _UNICODE
#define _UNICODE
#endif
#ifndef UNICODE
#define UNICODE
#endif

#include <iostream>
#include <windows.h>
#include <tchar.h>
#include "gl/gl.h"
#include "engine/Engine.h"
#include "engine/utils/MyPolygon.h"
#include "engine/object/ObjectLoader.h"

const SIZE MIN_WINDOW_SIZE = SIZE{900, 900};
const SIZE FIRST_WINDOW_SIZE = SIZE{900, 900};
const COLORREF BACKGROUND_COLOR = RGB(255, 255, 255);

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);

void EnableOpenGL(HWND hwnd, HDC *, HGLRC *);

void DisableOpenGL(HWND, HDC, HGLRC);

Engine engine = Engine();

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR
pCmdLine, int nCmdShow) {

    const wchar_t WINDOW_CLASS[] = L"MAIN_WINDOW_CLASS";
    const wchar_t WINDOW_TITLE[] = L"V-3D ^_^";

    HDC hDC;
    HGLRC hRC;
    MSG msg;
    WNDCLASSEX wc;
    HWND hwnd;
```

```

wc.cbSize = sizeof(WNDCLASSEXW);
wc.style = CS_OWNDC;
wc.lpfnWndProc = WindowProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = LoadIconW(nullptr, IDI_APPLICATION);
wc.hCursor = LoadCursorW(nullptr, IDC_ARROW);
wc.hbrBackground = CreateSolidBrush(BACKGROUND_COLOR);
wc.lpszMenuName = nullptr;
wc.lpszClassName = WINDOW_CLASS;
wc.hIconSm = LoadIconW(nullptr, IDI_APPLICATION);

if (!RegisterClassExW(&wc)) {
    MessageBoxW(nullptr, L"Error registering window class", L"Attention",
MB_OK);
    return 0;
}

hwnd = CreateWindowExW(
    // Optional window styles.
    0x0,
    // Window class
    WINDOW_CLASS,
    // Window text
    WINDOW_TITLE,
    // Window style
    WS_OVERLAPPEDWINDOW,
    // Size and position
    CW_USEDEFAULT, CW_USEDEFAULT, FIRST_WINDOW_SIZE.cx,
FIRST_WINDOW_SIZE.cy,
    // Parent window
    nullptr,
    // Menu
    nullptr,
    // Instance handle
    hInstance,
    // Additional application data
    nullptr
);

if (hwnd == nullptr) {

```

```

    MessageBoxW(nullptr, L"Error creating window", L"Attention", MB_OK);
    return 0;
}

ShowWindow(hwnd, nCmdShow);

/* enable OpenGL for the window */
EnableOpenGL(hwnd, &hDC, &hRC);

glEnable(GL_DEPTH_TEST);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(-0.1, 0.1, -0.1, 0.1, 0.2f, 1000.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_COLOR_MATERIAL);

bool bQuit = false;
/* program main loop */
while (!bQuit) {
    /* check for messages */
    if (PeekMessage(&msg, nullptr, 0, 0, PM_REMOVE)) {
        /* handle or dispatch messages */
        if (msg.message == WM_QUIT) {
            bQuit = true;
        } else {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    } else {
        /* OpenGL animation code goes here */
        engine.handleKeys(); // Faster than handling messages
        engine.draw(hDC);
        SetWindowTextA(hwnd, engine.description().data());
        Sleep(1000 / engine.fps);
    }
}

/* shutdown OpenGL */

```

```

DisableOpenGL(hwnd, hDC, hRC);

/* destroy the window explicitly */
DestroyWindow(hwnd);

return msg.wParam;
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM
wParam, LPARAM lParam) {
    switch (uMsg) {
        case WM_CLOSE: {
            PostQuitMessage(0);
            break;
        }
        case WM_DESTROY: {
            return 0;
        }
        case WM_KEYDOWN: {
            switch (wParam) {
                case 0x4F: { // O
                    OPENFILENAME ofn;
                    TCHAR szFile[260] = {0};

                    ZeroMemory(&ofn, sizeof(ofn));
                    ofn.lStructSize = sizeof(ofn);
                    ofn.hwndOwner = hwnd;
                    ofn.lpstrFile = szFile;
                    ofn.nMaxFile = sizeof(szFile);
                    ofn.lpstrFilter = _T("obj\\0*.obj\\0");
                    ofn.nFilterIndex = 1;
                    ofn.lpstrFileTitle = nullptr;
                    ofn.nMaxFileTitle = 0;
                    ofn.lpstrInitialDir = nullptr;
                    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

                    if (GetOpenFileName(&ofn) == TRUE) {
                        std::wstring wpath = ofn.lpstrFile;
                        std::string path(wpath.begin(), wpath.end());

                        Object obj = ObjectLoader::LoadObjModel(path);
                        obj.centerPolygonVertices();
                        obj.resizeToHeight(10);
                    }
                }
            }
        }
    }
}

```

```

        engine.scene.add(obj);
    }

    break;
}
case 0x51: { // Q
    switch (engine.renderMode) {
        case Engine::RenderMode::SOLID:
            engine.renderMode = Engine::RenderMode::WIREFRAME;
            break;
        case Engine::RenderMode::WIREFRAME:
            engine.renderMode = Engine::RenderMode::SOLID;
            break;
    }
    break;
}
case 0x52: { // R
    engine.scene = Scene();
    break;
}
case 0x46: { // F
    engine.scene.resetCamera();
    break;
}
case 0x45: { // E
    engine.scene.light.nextPosition();
    break;
}
case VK_ESCAPE: {
    PostQuitMessage(0);
    break;
}
default: {
    break;
}
}
break;
}
case WM_GETMINMAXINFO: {
    auto lpMMI = (LPMINMAXINFO) lParam;
    lpMMI->ptMinTrackSize.x = MIN_WINDOW_SIZE.cx;
    lpMMI->ptMinTrackSize.y = MIN_WINDOW_SIZE.cy;
    break;
}

```

```

    }
    default: {
        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
}

return 0;
}

void EnableOpenGL(HWND hwnd, HDC *hDC, HGLRC *hRC) {
    PIXELFORMATDESCRIPTOR pfd;

    int iFormat;

    /* get the device context (DC) */
    *hDC = GetDC(hwnd);

    /* set the pixel format for the DC */
    ZeroMemory(&pfd, sizeof(pfd));

    pfd.nSize = sizeof(pfd);
    pfd.nVersion = 1;
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
PFD_DOUBLEBUFFER;
    pfd.iPixelFormat = PFD_TYPE_RGBA;
    pfd.cColorBits = 24;
    pfd.cDepthBits = 16;
    pfd.iLayerType = PFD_MAIN_PLANE;

    iFormat = ChoosePixelFormat(*hDC, &pfd);

    SetPixelFormat(*hDC, iFormat, &pfd);

    /* create and enable the render context (RC) */
    *hRC = wglCreateContext(*hDC);

    wglMakeCurrent(*hDC, *hRC);
}

void DisableOpenGL(HWND hwnd, HDC hDC, HGLRC hRC) {
    wglMakeCurrent(nullptr, nullptr);
    wglDeleteContext(hRC);
    ReleaseDC(hwnd, hDC);
}

```



```
}
```

Файл Engine.h

```
//
```

```
// Created by iXasthur on 15.11.2020.
```

```
//
```

```
#ifndef V_3D_ENGINE_H
```

```
#define V_3D_ENGINE_H
```

```
#include <array>
```

```
#include "gl/gl.h"
```

```
#include "scene/Scene.h"
```

```
#include <cmath>
```

```
class Engine {
```

```
public:
```

```
    enum class RenderMode {
```

```
        SOLID,
```

```
        WIREFRAME
```

```
    };
```

```
    const int fps = 120;
```

```
    const float cameraMoveDeltaAbsolute = 1.0f;
```

```
    const float cameraRotationDelta = 1.5f;
```

```
    RenderMode renderMode = RenderMode::SOLID;
```

```
    Scene scene = Scene();
```

```
    Engine() = default;
```

```
    void handleKeys() {
```

```
        float moveSpeed = 0;
```

```
        if (GetKeyState(0x57) < 0) { // W
```

```
            moveSpeed = cameraMoveDeltaAbsolute;
```

```
        }
```

```
        if (GetKeyState(0x53) < 0) { // S
```

```
            moveSpeed = -cameraMoveDeltaAbsolute;
```

```
        }
```

```
        if (moveSpeed != 0) {
```

```
            float alphaY = scene.camera.eulerRotation.y;
```

```
            scene.camera.position.x -= sin(-alphaY / 180.0f * M_PI) * moveSpeed;
```

```
            scene.camera.position.z -= cos(-alphaY / 180.0f * M_PI) * moveSpeed;
```

```

}

moveSpeed = 0;
if (GetKeyState(0x44) < 0) { // D
    moveSpeed = cameraMoveDeltaAbsolute;
}
if (GetKeyState(0x41) < 0) { // A
    moveSpeed = -cameraMoveDeltaAbsolute;
}
if (moveSpeed != 0) {
    float alphaY = scene.camera.eulerRotation.y;
    scene.camera.position.x += cos(-alphaY / 180.0f * M_PI) * moveSpeed;
    scene.camera.position.z -= sin(-alphaY / 180.0f * M_PI) * moveSpeed;
}

if (GetKeyState(VK_SHIFT) < 0) {
    scene.camera.position.y -= cameraMoveDeltaAbsolute;
}
if (GetKeyState(VK_SPACE) < 0) {
    scene.camera.position.y += cameraMoveDeltaAbsolute;
}

if (GetKeyState(VK_LEFT) < 0) {
    scene.camera.eulerRotation.y -= cameraRotationDelta;
}
if (GetKeyState(VK_RIGHT) < 0) {
    scene.camera.eulerRotation.y += cameraRotationDelta;
}
if (GetKeyState(VK_UP) < 0) {
    if (scene.camera.eulerRotation.x > -90.0f) {
        scene.camera.eulerRotation.x -= cameraRotationDelta;
    } else {
        scene.camera.eulerRotation.x = -90.0f;
    }
}
if (GetKeyState(VK_DOWN) < 0) {
    if (scene.camera.eulerRotation.x < 90.0f) {
        scene.camera.eulerRotation.x += cameraRotationDelta;
    } else {
        scene.camera.eulerRotation.x = 90.0f;
    }
}
}

```

```

void draw(HDC hdc) {
    switch (renderMode) {
        case RenderMode::SOLID:
            glPolygonMode(GL_FRONT, GL_FILL);
            glPolygonMode(GL_BACK, GL_FILL);
            break;
        case RenderMode::WIREFRAME:
            glPolygonMode(GL_FRONT, GL_LINE);
            glPolygonMode(GL_BACK, GL_LINE);
            break;
    }

    // Clear screen
    glClearColor(scene.backgroundColor.Rf,          scene.backgroundColor.Gf,
scene.backgroundColor.Bf,
                scene.backgroundColor.Af);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Push to save perspective projection
    glPushMatrix();

    // Rotate and scene (simulate camera)
    glRotatef(scene.camera.eulerRotation.x, 1, 0, 0);
    glRotatef(scene.camera.eulerRotation.y, 0, 1, 0);
    glRotatef(scene.camera.eulerRotation.z, 0, 0, 1);
    glTranslatef(-scene.camera.position.x,          -scene.camera.position.y,          -
scene.camera.position.z);

    // Set light
    glLightfv(GL_LIGHT0,                                GL_POSITION,
scene.light.position.toArray().data());

    // Rotate scene
    glRotatef(scene.eulerRotation.x, 1, 0, 0);
    glRotatef(scene.eulerRotation.y, 0, 1, 0);
    glRotatef(scene.eulerRotation.z, 0, 0, 1);

    // Draw
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    for (Object &obj : scene.objects) {
        std::vector<float> vertices = obj.getVertices();

```

```

std::vector<float> normals = obj.getNormals();

glPushMatrix();

glTranslatef(obj.position.x, obj.position.y, obj.position.z);
glColor3f(obj.color.Rf, obj.color.Gf, obj.color.Bf);

glNormalPointer(GL_FLOAT, 0, normals.data());
glVertexPointer(3, GL_FLOAT, 0, vertices.data());
glDrawArrays(GL_TRIANGLES, 0, vertices.size() / 3);

glPopMatrix();
}
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);

glPopMatrix();

SwapBuffers(hdc);
}

std::string description() {
    std::string s;
    for (const Object &obj : scene.objects) {
        s += obj.name + ", ";
    }
    s += scene.camera.position.toString();
    s += ", light: " + std::to_string(scene.light.currentLightPositionIndex);
    switch (renderMode) {
        case RenderMode::SOLID:
            s += ", SOLID";
            break;
        case Engine::RenderMode::WIREFRAME:
            s += ", WIREFRAME";
            break;
    }
    return s;
}
};

#endif //V_3D_ENGINE_H
Файл Camera.h

```

```
//  
// Created by iXasthur on 14.11.2020.  
//
```

```
#ifndef V_3D_CAMERA_H  
#define V_3D_CAMERA_H  
  
#include "../utils/Vector3.h"  
#include "../utils/EulerAngle.h"
```

```
class Camera {  
public:  
    Vector3 position;  
    EulerAngle eulerRotation;  
};
```

```
#endif //V_3D_CAMERA_H
```

Файл Light.h

```
//  
// Created by iXasthur on 16.11.2020.  
//
```

```
#ifndef V_3D_LIGHT_H  
#define V_3D_LIGHT_H  
  
#include <vector>  
#include "../utils/Vector4.h"
```

```
class Light {  
public:  
    Vector4 position;  
    std::vector<Vector4> lightPositions;  
    int currentLightPositionIndex = -1;  
  
    void nextPosition() {  
        if (!lightPositions.empty()) {  
            currentLightPositionIndex++;  
            if (currentLightPositionIndex > lightPositions.size() - 1) {  
                currentLightPositionIndex = 0;  
            }  
        }  
    }  
};
```

```

        position = lightPositions[currentLightPositionIndex];
    } else {
        currentLightPositionIndex = -1;
    }
}
};

```

```

#endif //V_3D_LIGHT_H

```

Файл Object.h

```

//
// Created by iXasthur on 14.11.2020.
//

#ifndef V_3D_OBJECT_H
#define V_3D_OBJECT_H

#include <vector>
#include "../utils/MyPolygon.h"
#include "../utils/Color.h"

class Object {
public:
    Vector3 position;
    std::vector<MyPolygon> polygons;
    std::string name;

    Color color = Color(255, 255, 255, 255);

    Object() = default;

    std::vector<float> getVertices() {
        std::vector<float> vertices;
        for (MyPolygon polygon : polygons) {
            for (Vector3 vertex : polygon.vertices) {
                for (float v : vertex.toArray()) {
                    vertices.emplace_back(v);
                }
            }
        }
        return vertices;
    }
}

```

```

std::vector<float> getNormals() {
    std::vector<float> normals;
    for (MyPolygon polygon : polygons) {
        for (Vector3 normal : polygon.normals) {
            for (float n : normal.toArray()) {
                normals.emplace_back(n);
            }
        }
    }
    return normals;
}

```

```

void normalizePolygonVertices() {
    for (MyPolygon &polygon : polygons) {
        for (Vector3 &vertex : polygon.vertices) {
            vertex.normalize();
        }
    }
}

```

```

void resizeToHeight(float h) {
    Vector3 dim = dimension();
    float delta = h / dim.y;

    for (MyPolygon &polygon : polygons) {
        for (Vector3 &vertex : polygon.vertices) {
            vertex.x *= delta;
            vertex.y *= delta;
            vertex.z *= delta;
        }
    }
}

```

```

void centerPolygonVertices() {
    Vector3 lowestVertex;

    for (MyPolygon &polygon : polygons) {
        for (Vector3 &vertex : polygon.vertices) {
            if (vertex.y < lowestVertex.y) {
                lowestVertex = vertex;
            }
        }
    }
}

```

```

for (MyPolygon &polygon : polygons) {
    for (Vector3 &vertex : polygon.vertices) {
        vertex.y -= lowestVertex.y;
    }
}

Vector3 dimension() {
    float minX = 0;
    float maxX = 0;
    float minY = 0;
    float maxY = 0;
    float minZ = 0;
    float maxZ = 0;

    if (!polygons.empty()) {
        Vector3 fv = polygons.front().vertices.front();
        minX = fv.x;
        maxX = fv.x;
        minY = fv.y;
        maxY = fv.y;
        minZ = fv.z;
        maxZ = fv.z;

        for (MyPolygon polygon : polygons) {
            for (Vector3 vertex : polygon.vertices) {
                if (vertex.x < minX) {
                    minX = vertex.x;
                } else if (vertex.x > maxX) {
                    maxX = vertex.x;
                }

                if (vertex.y < minY) {
                    minY = vertex.y;
                } else if (vertex.y > maxY) {
                    maxY = vertex.y;
                }

                if (vertex.z < minZ) {
                    minZ = vertex.z;
                } else if (vertex.z > maxZ) {
                    maxZ = vertex.z;
                }
            }
        }
    }
}

```



```

    }
    }
    }
}

return {maxX - minX, maxY - minY, maxZ - minZ};
}
};

```

```
#endif //V_3D_OBJECT_H
```

Файл ObjectLoader.h

```

//
// Created by iXasthur on 16.11.2020.
//

#ifndef V_3D_OBJECTLOADER_H
#define V_3D_OBJECTLOADER_H

#include <iostream>
#include <sstream>
#include <fstream>
#include "Object.h"

class ObjectLoader {
private:
    static std::vector<std::string> splitString(const std::string &s, char delim) {
        std::vector<std::string> result;
        std::stringstream ss(s);
        std::string item;

        while (getline(ss, item, delim)) {
            result.push_back(item);
        }

        return result;
    }

public:
    static Object LoadObjModel(const std::string &filename) {
        Object obj = Object();
        obj.name = filename;
    }
}

```

```

int find = obj.name.find_last_of('/');
if (find != std::string::npos) {
    obj.name = obj.name.substr(find + 1);
}

find = obj.name.find_last_of("\\");
if (find != std::string::npos) {
    obj.name = obj.name.substr(find + 1);
}

std::vector<Vector3> vertices;
std::vector<MyPolygon> polygons;
std::vector<Vector3> normals;

std::ifstream in(filename, std::ios::in);
if (in) {
    std::string line;
    while (std::getline(in, line)) {
        if (line.substr(0, 2) == "v ") {
            std::istringstream v(line.substr(2));
            Vector3 vertex;

            v >> vertex.x;
            v >> vertex.y;
            v >> vertex.z;

            vertices.push_back(vertex);
        } else if (line.substr(0, 3) == "vn ") {
            std::istringstream v(line.substr(3));
            Vector3 normal;

            v >> normal.x;
            v >> normal.y;
            v >> normal.z;

            normals.push_back(normal);
        } else if (line.substr(0, 2) == "f ") {
            std::vector<std::string> faceStrs = splitString(line, ' ');
            faceStrs.erase(faceStrs.begin());

            std::vector<int> f;
            std::vector<int> t;

```

```

std::vector<int> n;

for (auto &faceStr : faceStrs) {
    std::vector<std::string> faceElementStrs = splitString(faceStr, '/');

    for (int j = 0; j < faceElementStrs.size(); ++j) {
        if (!faceElementStrs[j].empty()) {
            std::istringstream v(faceElementStrs[j]);
            int fv;
            v >> fv;

            switch (j) {
                case 0: {
                    f.emplace_back(fv);
                    break;
                }
                case 1: {
                    t.emplace_back(fv);
                    break;
                }
                case 2: {
                    n.emplace_back(fv);
                    break;
                }
                default: {
                    break;
                }
            }
        }
    }
}

std::array<Vector3, 3> pVertices;
std::array<Vector3, 3> pNormals;
if (f.size() == 3) {
    pVertices[0] = vertices[f[0] - 1];
    pVertices[1] = vertices[f[1] - 1];
    pVertices[2] = vertices[f[2] - 1];

    if (!normals.empty()) {
        pNormals[0] = normals[n[0] - 1];
        pNormals[1] = normals[n[1] - 1];
        pNormals[2] = normals[n[2] - 1];
    }
}

```

```

        polygons.emplace_back(MyPolygon(pVertices, pNormals));
    } else {
        polygons.emplace_back(MyPolygon(pVertices));
    }

    } else if (f.size() == 4) {
        pVertices[0] = vertices[f[0] - 1];
        pVertices[1] = vertices[f[1] - 1];
        pVertices[2] = vertices[f[2] - 1];

        if (!normals.empty()) {
            pNormals[0] = normals[n[0] - 1];
            pNormals[1] = normals[n[1] - 1];
            pNormals[2] = normals[n[2] - 1];
            polygons.emplace_back(MyPolygon(pVertices, pNormals));
        } else {
            polygons.emplace_back(MyPolygon(pVertices));
        }

        pVertices[0] = vertices[f[0] - 1];
        pVertices[1] = vertices[f[2] - 1];
        pVertices[2] = vertices[f[3] - 1];

        if (!normals.empty()) {
            pNormals[0] = normals[n[0] - 1];
            pNormals[1] = normals[n[2] - 1];
            pNormals[2] = normals[n[3] - 1];
            polygons.emplace_back(MyPolygon(pVertices, pNormals));
        } else {
            polygons.emplace_back(MyPolygon(pVertices));
        }
    } else {
        std::cerr << "Object face must have 3 or 4 vertices" << std::endl;
    }
}

}

} else {
    std::cerr << "Cannot open " << filename << std::endl;
}

obj.polygons = polygons;
return obj;
}

```

```
};
```

```
#endif //V_3D_OBJECTLOADER_H
```

Файл Scene.h

```
//
```

```
// Created by iXasthur on 14.11.2020.
```

```
//
```

```
#ifndef V_3D_SCENE_H
```

```
#define V_3D_SCENE_H
```

```
#include <vector>
```

```
#include <array>
```

```
#include <cstdlib>
```

```
#include "../object/Object.h"
```

```
#include "../light/Light.h"
```

```
#include "../camera/Camera.h"
```

```
#include "../utils/Color.h"
```

```
class Scene {
```

```
public:
```

```
    std::vector<Object> objects;
```

```
    Camera camera;
```

```
    EulerAngle eulerRotation;
```

```
    Light light;
```

```
    Color backgroundColor = Color(34, 34, 34, 255);
```

```
    Scene() {
```

```
        resetLight();
```

```
        resetCamera();
```

```
    };
```

```
    void resetCamera() {
```

```
        camera.position = {0, 10, 40};
```

```
        camera.eulerRotation = EulerAngle();
```

```
    }
```

```
    void resetLight() {
```

```
        light.lightPositions.clear();
```

```
        light.lightPositions.emplace_back(Vector4(500, 500, 500, 0));
```

```

        light.lightPositions.emplace_back(Vector4(-500, 500, 500, 0));
        light.lightPositions.emplace_back(Vector4(-500, 500, -500, 0));
        light.lightPositions.emplace_back(Vector4(500, 500, -500, 0));
        light.lightPositions.emplace_back(Vector4(500, -500, 500, 0));
        light.lightPositions.emplace_back(Vector4(-500, -500, 500, 0));
        light.lightPositions.emplace_back(Vector4(-500, -500, -500, 0));
        light.lightPositions.emplace_back(Vector4(500, -500, -500, 0));
        light.nextPosition();
    }

    void add(const Object &obj) {
        objects.emplace_back(obj);
        reorderObjects();
    }

    void reorderObjects() {
        Vector3 pos;
        for (int i = 0; i < objects.size(); ++i) {
            if (i > 0) {
                Vector3 lastDim = objects[i - 1].dimension();
                Vector3 dim = objects[i].dimension();
                pos.x += lastDim.x / 2 + dim.x / 2;
                pos.x += 5;
            }

            objects[i].position.x = pos.x;
        }
    }
};

```

```

#endif //V_3D_SCENE_H

```

Файл Color.h

```

//
// Created by iXasthur on 15.11.2020.
//

```

```

#ifndef V_3D_COLOR_H
#define V_3D_COLOR_H

```

```

class Color {

```

```

public:
    int R;
    int G;
    int B;
    int A;

    float Rf;
    float Gf;
    float Bf;
    float Af;

    Color(int r, int g, int b, int a) {
        R = r;
        G = g;
        B = b;
        A = a;

        Rf = (float) R / 255.0f;
        Gf = (float) G / 255.0f;
        Bf = (float) B / 255.0f;
        Af = (float) A / 255.0f;
    }
};

```

```

#endif //V_3D_COLOR_H

```

Файл EulerAngle.h

```

//
// Created by iXasthur on 15.11.2020.
//

```

```

#ifndef V_3D_EULERANGLE_H
#define V_3D_EULERANGLE_H

```

```

class EulerAngle {
public:
    float x;
    float y;
    float z;

    EulerAngle(float x, float y, float z) : x(x), y(y), z(z) {

```

```

    }

    EulerAngle() : x(0), y(0), z(0) {

    }
};

#endif //V_3D_EULERANGLE_H

```

Файл MyPolygon.h

```

//
// Created by iXasthur on 14.11.2020.
//

#ifndef V_3D_MYPOLYGON_H
#define V_3D_MYPOLYGON_H

#include <array>
#include "Vector3.h"

class MyPolygon {
private:
    Vector3 getNormal() {
        Vector3 a = vertices[0] - vertices[1];
        Vector3 b = vertices[0] - vertices[2];
        return Vector3::cross(a, b);
    }

public:
    std::array<Vector3, 3> vertices;
    std::array<Vector3, 3> normals;

    explicit MyPolygon(std::array<Vector3, 3> vertices) {
        this->vertices = vertices;
        for (auto &normal : normals) {
            normal = getNormal();
        }
    }

    MyPolygon(std::array<Vector3, 3> vertices, std::array<Vector3, 3> normals) {
        this->vertices = vertices;
    }
}

```



```

        this->normals = normals;
    }
};

#endif //V_3D_MYPOLYGON_H

```

Файл Vector3.h

```

//
// Created by iXasthur on 15.11.2020.
//

#ifndef V_3D_VECTOR3_H
#define V_3D_VECTOR3_H

#include <array>
#include <cmath>

class Vector3 {
public:
    float x;
    float y;
    float z;

    Vector3(float x, float y, float z) : x(x), y(y), z(z) {

    }

    Vector3() : x(0), y(0), z(0) {

    }

    Vector3 operator-(const Vector3 &b) const {
        Vector3 vector;
        vector.x = this->x - b.x;
        vector.y = this->y - b.y;
        vector.z = this->z - b.z;
        return vector;
    }

    Vector3 operator+(const Vector3 &b) const {
        Vector3 vector;
        vector.x = this->x + b.x;

```

```

    vector.y = this->y - b.y;
    vector.z = this->z - b.z;
    return vector;
}

std::array<float, 3> toArray() {
    return {x, y, z};
}

[[nodiscard]] float length() const {
    return std::sqrt((x * x) + (y * y) + (z * z));
}

void normalize() {
    float length = this->length();
    x = x / length;
    y = y / length;
    z = z / length;
}

[[nodiscard]] std::string toString() const {
    std::string s;

    std::string xs = std::to_string(x);
    xs = xs.substr(0, xs.find('.') + 3);

    std::string xy = std::to_string(y);
    xy = xy.substr(0, xs.find('.') + 3);

    std::string xz = std::to_string(z);
    xz = xz.substr(0, xz.find('.') + 3);

    s = "x: " + xs + " y: " + xy + " z: " + xz;
    return s;
}

static Vector3 cross(Vector3 v1, Vector3 v2) {
    Vector3 v;
    v.x = v1.y * v2.z - v1.z * v2.y;
    v.y = v1.z * v2.x - v1.x * v2.z;
    v.z = v1.x * v2.y - v1.y * v2.x;
    return v;
}

```

```
};
```

```
#endif //V_3D_VECTOR3_H
```

Файл Vector4.h

```
//
```

```
// Created by iXasthur on 15.11.2020.
```

```
//
```

```
#ifndef V_3D_VECTOR4_H
```

```
#define V_3D_VECTOR4_H
```

```
#include <array>
```

```
class Vector4 {
```

```
public:
```

```
    float x;
```

```
    float y;
```

```
    float z;
```

```
    float w;
```

```
    Vector4(float x, float y, float z, float w) : x(x), y(y), z(z), w(w) {
```

```
    }
```

```
    Vector4() : x(0), y(0), z(0), w(0) {
```

```
    }
```

```
    std::array<float, 4> toArray() {
```

```
        return {x, y, z, w};
```

```
    }
```

```
};
```

```
#endif //V_3D_VECTOR4_H
```

Обозначение					Наименование					Дополнительные сведения				
					Текстовые документы									
БГУИР КП 1–40 01 01 612 ПЗ					Пояснительная записка					52 с.				
					Графические документы									
ГУИР 851006 01 ПД					Схема программы на А1					Формат А1				