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

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Графический редактор композиций тел вращения»

Студент ИУ7-55Б _____ Романов С. К.
(Группа) (Подпись, дата) (Фамилия И.О.)

Руководитель курсовой работы _____ Майков К. А.
(Подпись, дата) (Фамилия И.О.)

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

УТВЕРЖДАЮ

Заведующий кафедрой
ИУ7
(Индекс)
_____ И.В.Рудаков
(И.О.Фамилия)
« ___ » _____ 2022 г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине _____Компьютерная графика_____

_____Графический редактор композиций тел вращения_____
(Тема курсового проекта)

Студент _____Романов Семен Константинович, ИУ7-45Б_____
(ФИО, индекс группы)

График выполнения проекта: 25% к _4_ нед., 50% к _7_ нед., 75% к _11_нед., 100% к 14 нед.

1. Техническое задание

Разработать программное обеспечение для моделирования и редактирования композиции тел вращения. Интерфейс должен позволить выбрать модель для построения из предложенного набора, задавать параметры для построения тела вращения, должна быть возможность изменений физических параметров (ширина, высота), положения в пространстве (перемещения, поворот, масштабирование), редактирования (отсечение, закраска нужным цветом) тел. Проектируемый программный продукт должен позволять производить булевы операции над телами вращения, размещение одного источника света, просмотра сцены (состоит не более чем из 4-ех моделей) путем перемещения и поворота камеры.

2. Оформление курсовой работы

2.1. Расчетно-пояснительная записка на 25-30 листах формата А4.

Расчетно-пояснительная записка должна содержать постановку введение, аналитическую часть, конструкторскую часть, технологическую часть, экспериментально-исследовательская часть, заключение, список литературы, приложения.

2.2. Перечень графического материала (плакаты, схемы, чертежи и т.п.). На защиту проекта должна быть представлена презентация, состоящая из 10-15 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО, результаты проведенных исследований.

Дата выдачи задания «____» _____ 20__ г.

Руководитель курсовой работы _____ _____Майков К.А._____
(Подпись, дата) (Фамилия И.О.)

Студент _____ _____Романов С .К.__
(Подпись, дата) (Фамилия И.О.)

РЕФЕРАТ

Расчетно-пояснительная записка 56 с., 15 рис., 23 ист., 1 прил.

КОМПЬЮТЕРНАЯ ГРАФИКА, АЛГОРИТМЫ УДАЛЕНИЯ НЕВИДИМЫХ ЛИНИЙ, Z-БУФЕР, ЗАКРАСКА, ОСВЕЩЕНИЕ, СЦЕНА, ТЕЛА ВРАЩЕНИЯ.

Целью работы является разработка программы, позволяющее создавать композиции тел вращения.

Для визуализации сцены использовался алгоритм с Z-буфером, а для представления модели тела вращения использовалась поверхностная модель, представленная в виде списка ребер.

В процессе работы были проанализированы различные алгоритмы, методы представления, закраски геометрических моделей на сцене.

Проведено исследование быстродействия программы при различном количестве создания объектов на сцене с фиксированным количеством ребер. Также с фиксированным количеством объектов, но с различным количеством ребер. Из результатов исследования следует, что время отрисовки сцены увеличивается как при увеличении количества объектов на сцене, так и при увеличении количества ребер.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие термины с соответствующими определениями.

Graphical user interface — «Графический интерфейс пользователя» обеспечивает возможность управления поведением вычислительной системы через визуальные элементы управления — окна, списки, кнопки, гиперссылки и т.д. [1]

Буфер кадра — часть графической памяти для хранения массива кодов, определяющих засветку пикселей на экране [1].

Визуализация (англ. Rendering) — создание плоских изображений трехмерных (3D) моделей [1].

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие сокращения и обозначения.

GUI — «Графический интерфейс пользователя».

ПО — Программное обеспечение

ВВЕДЕНИЕ

Целью данного курсового проекта является разработка ПО, визуализирующие различные композиции тел вращения. Это включает в себя такие вещи, как

1) Воссоздание сложных композиций тел.
2) Создание тел вращения путем вращения выше обозначенных композиций вокруг установленной оси.
3) Представление полноценной освещенной сцены с композициями объектов.
4) Обзор рабочей сцены из различных ее участков посредством нескольких камер.
5) Интерактивность с различными объектами сцены, такими как:
   - Камеры;
   - Модели;
   - Композиты.

Для достижения поставленных целей необходимо решить следующие задачи:

- описать структуру трехмерной сцены;
- реализовать оптимальные алгоритмы представления, преобразования и визуализации твердотельной модели;
- реализовать аппаратную обработку всех элементов сцены;
- спроектировать процесс моделирования сцены;
- описать использующиеся структуры данных;
- определить средства программной реализации;
- провести экспериментальные замеры временных характеристик разработанного ПО.

В ходе курсовой работы будут затронуты такие темы, как:

- понятия о телах и поверхностях вращения;
- удаление невидимых линий и поверхностей;

– методы закрашивания объемных тел;
– преимущества и особенности языка Rust.

1 Аналитический раздел

В данной части проводится анализ объектов сцены и существующих алгоритмов построения изображений и выбор более подходящих алгоритмов для дальнейшего использования.

1.1 Описание объектов сцены

Сцена состоит из следующих объектов:

- **Камера** – объект, с которого осуществляется наблюдение за сценой. Характеризуется своим пространственным положением, направлением просмотра, углом обзора, ближним и дальним границами обзора.
- **Источник света** – объект, который освещает сцену. Направленный источник света представляет собой вектор направления света и принимает ортогональную проекцию визуализируемой сцены из своего положения с некоторым ограниченным положением. В зависимости от расположения источника и направления распространения лучей света, определяется тень от объекта, расположенных на сцене.
- **Модель** – сущность, которая может быть отображена на сцене. Примитивные тела вращения, расположенные в пространстве сцены. Каждая модель представляет собой набор граней, описываемых точками в пространстве, которые соединены ребрами.
- **Композит** – объект, который может содержать в себе другие объекты.

1.2 Анализ методов создания моделей

Тело вращения — это поверхность в евклидовом пространстве, образованная вращением кривой (образующей) вокруг оси вращения.[2]

На данном этапе стоит рассмотреть существующие схемы представления тел.

1.2.1 Представление тел вращения

Тела вращения характеризуются осью, радиусами оснований и конструктивными точками образующей поверхности тел. Чтобы лучше разобраться в принципах конструктивного построения формы цилиндра и конуса, следует обратить внимание на рис. 1 и на рис. 2, где они изображены в виде прозрачных проволочных моделей. На рисунках ясно выражены конструктивная основа и объемно-пространственная характеристика формы предметов. Задача состоит в том, чтобы реализовать грамотное и правильное изображение тел на сцене.

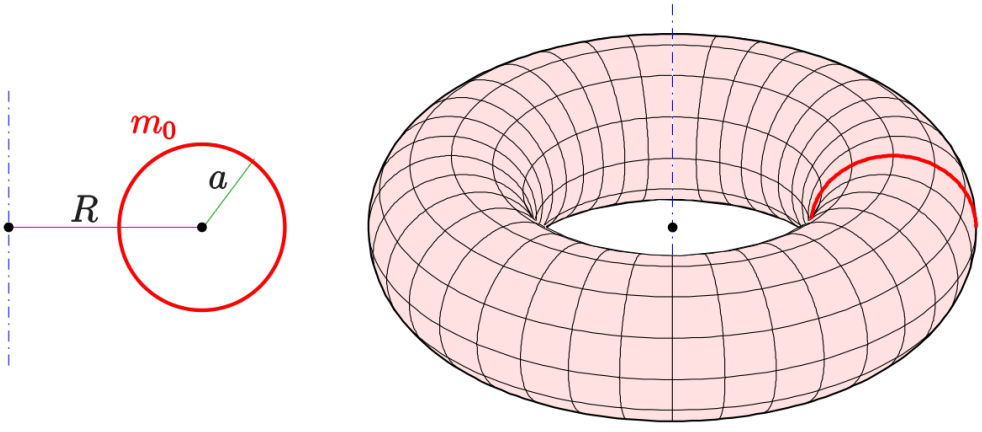

Рис. 1: Проволочная модель тора

Тело вращения образуется вращательной разверткой кривой профиля C вокруг оси. Если поверхность задается с помощью:

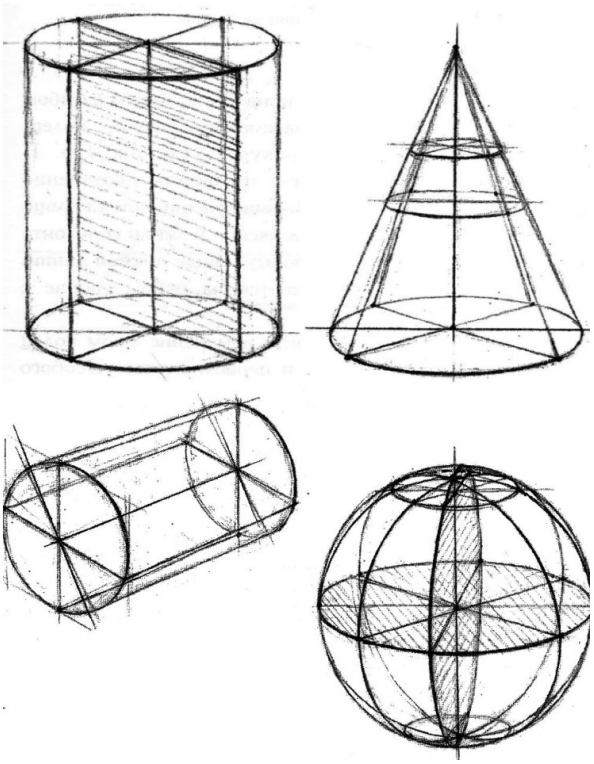

Рис. 2: Различные проволочные модели

$$P(u,v) = (X(v)cos(u), X(v)sin(u), Z(v)) \tag{1}$$

где X и Z - функции, тогда вектор нормали может быть задан через:

$$n(u,v) = X(v)(Z'(v)cos(u), Z'(v)sin(u), -X'(v)) \tag{2}$$

где апостроф обозначает первую производную функции.

1.3 Способы описания трехмерных геометрических моделей на сцене

В компьютерной графики для описание трехмерных геометрических объектов существует три типа моделей: каркасная, поверхностная и твердо-

тельная [3]. Использование моделей позволяет правильно отображать форму и размеры объектов сцены.

- *Каркасная модель* — простейший вид моделей, который содержит минимум информации только о вершинах и ребрах объектов. Это моделирование самого низкого уровня, которое имеет ряд серьезных ограничений, большинство из которых возникает из-за недостатка информации о гранях, которые заключены между ребрами. Невозможно выделить внутреннюю и внешнюю область изображения твердого объемного тела. Однако каркасная модель требует меньше памяти и затрат времени на построение, что достаточно пригодна для решения задач, не требующие информации о поверхности объекта (например, если в объекте есть отверстия). Проблемой этой модели заключается в том, что она не позволяет отличить видимые грани от невидимых. Операции по удалению невидимых линий можно выполнить только вручную с применением команд редактирования каждой отдельной линии, но результат работы нарушает каркасную конструкцию, по причине того что линии невидимы в одном случае и видимы в другом. Кроме того, каркасная модель не несет информации о поверхностях, ограничивающих форму, что обуславливает невозможность обнаружения нежелательных взаимодействий между гранями объекта [4].
- *Поверхностная модель* часто используется в компьютерной графике, кроме содержание информации о вершинах и ребрах, содержит еще информацию о поверхности. При построении поверхностной модели предполагается, что технические объекты ограничены поверхностями, которые отделяют их от окружающей среды. Недостатком поверхностной модели является отсутствие информации о том, с какой стороны поверхности находится материал.
- *Твердотельная модель* отличается от поверхностной тем, что в данной модели к информации о поверхностях добавляется информация о том,

с какой стороны расположен материал. Это достигается путем указания направления внутренней нормали.

Для решения поставленной задачи не подойдет каркасная модель, так как такое представление будет приводить к неправильному восприятию форм объекта. Твердотельная модель также не подойдет, так как по поставленной задачи нет необходимости знать из какого материала будет выполнен тот или иной объект и с какой стороны расположен материал. Поэтому выбор остается лишь поверхностной модели.

1.4 Способы задания поверхностных моделей

Поверхностная модель задается следующими способами [4, 5].

- *Аналитический (параметрический) способ* характеризуется описанием модели объекта, которое доступно в неявной форме,то есть для получения визуальных характеристик необходимо дополнительно вычислять некоторую функцию, которая зависит от параметра.
- *Полигональная сетка* характеризуется совокупностью вершин, ребер и граней, определяющих форму объекта в трехмерном пространстве.

Рассмотрим существующие способы хранения информации о полигональной сетке.

Вершинное представление описывает объект множество вершин, соединенных с другими вершинами (вершины, которые указывают на другие вершины). Информация о ребрах и гранях неявно присутствует в представлении из-за чего для восстановления исходного тела необходимо обойти все вершины и составить списки граней. Кроме того, операции с ребрами и гранями выполнить нелегко [3]. Тем не менее из-за простоты представления дает возможность множество операций над сеткой. Хранение информации о сетке требует не так много памяти по сравнению с другими способами. На ри-

сунке 3 представлен пример вершинного представления, рассмотренный на кубе.

Список вершин

v0	0,0,0	v1 v5 v4 v3 v9
v1	1,0,0	v2 v6 v5 v0 v9
v2	1,1,0	v3 v7 v6 v1 v9
v3	0,1,0	v2 v6 v7 v4 v9
v4	0,0,1	v5 v0 v3 v7 v8
v5	1,0,1	v6 v1 v0 v4 v8
v6	1,1,1	v7 v2 v1 v5 v8
v7	0,1,1	v4 v3 v2 v6 v8
v8	.5,.5,0	v5 v6 v7 v8
v9	.5,.5,1	v0 v1 v2 v3

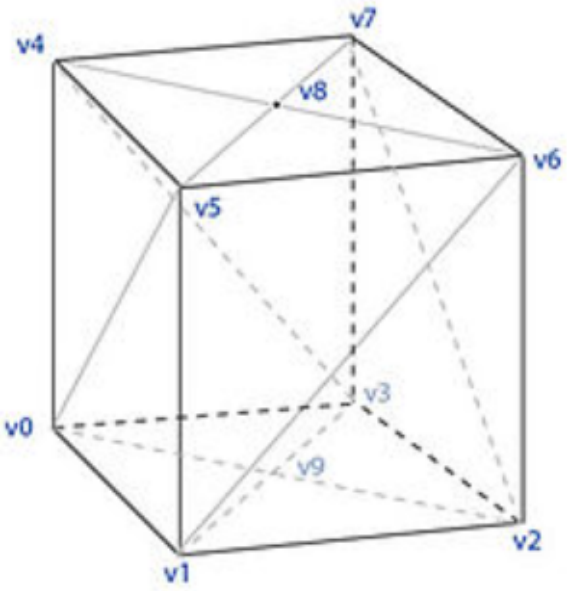

Рис. 3: Вершинное представление

Представление называемый списком граней представляет объект не только множеством вершин, но граней. В отличие от предыдущего способа, вершины и граны определены явно, благодаря чему нахождение соседних вершин и граней довольно проста и поиск соседних граней и вершин занимает постоянное время [3]. Также список вершин содержит список граней, связанных с каждой вершиной. Однако ребра неявны, поэтому поиск все равно необходим, чтобы найти все грани, окружающие данную грань. Другие динамические операции, такие как разделение или слияние граней, также сложны с сетками граней и вершин. Пример представления список граней представлен на рисунке 4.

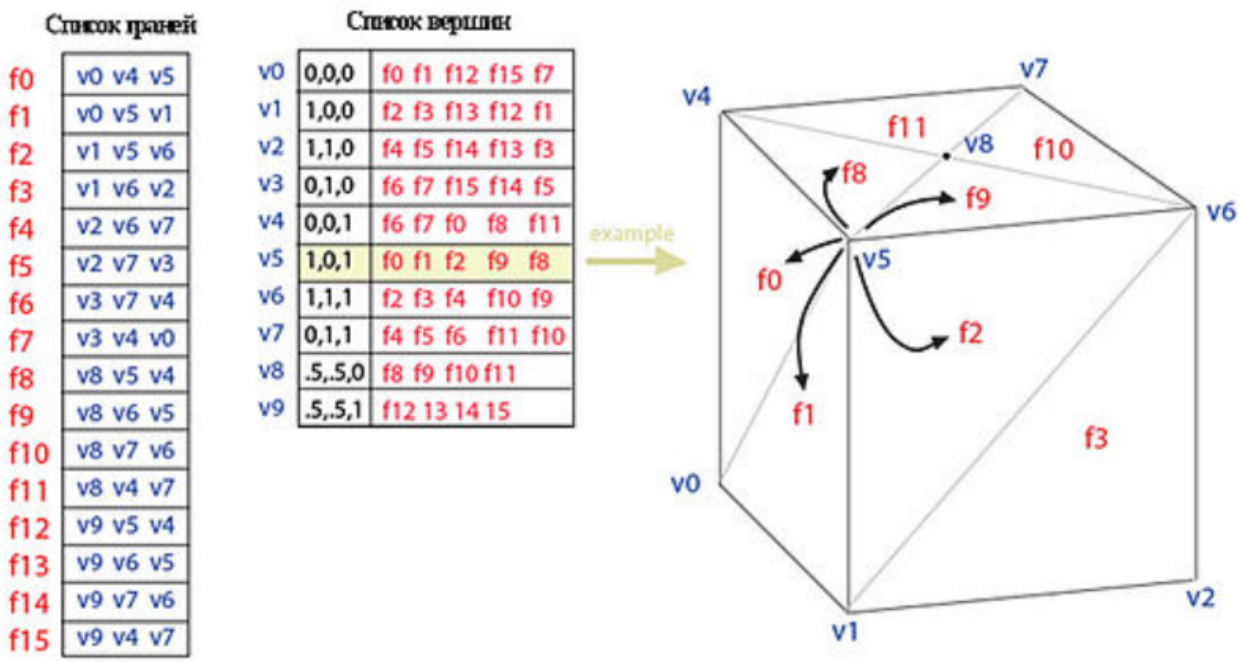

Рис. 4: Список граней

1.4.1 Понятие поверхности вращения

Поверхность вращения - это один из часто встречающихся типов поверхностей. Сферы и цилиндры можно рассматривать как одни из таких аповерхностей. В общем случае, можно получить тело вращения, вращая тело или набор тел вокруг некоторой произвольной оси. Чтобы более подробно проанализировать, что это значит, рассмотрим самый простой объект для вращения, а именно точку. В этом случае получается окружность в плоскости, ортогональной оси, с центром на оси и радиусом, равным расстоянию точки до оси.

Отсюда следует, что можно представить тело вращения как состоящий из объединения окружностей с центром на оси, по одной для каждой точки вращаемого объекта. Это также предполагает, что способ параметризации точки **P** объекта, полученного путем вращения кривой вокруг оси, заключа-

ется в использовании двух параметров. Один параметр - это параметр точки на кривой, которая привела к появлению **P**, а другой - угол, на который она была повернута. Для общих объектов вращения нам понадобилось бы k + 1 параметров, где k - количество параметров, необходимых для параметризации вращаемого объекта.

Наше фактическое определение поверхности вращения, которое будет дано в терминах параметризации, ограничится случаем, когда кривая вращается вокруг оси x. Это упростит определение. Кроме того, из этого можно получить поверхности вращения вокруг произвольной оси, используя жесткие движения.

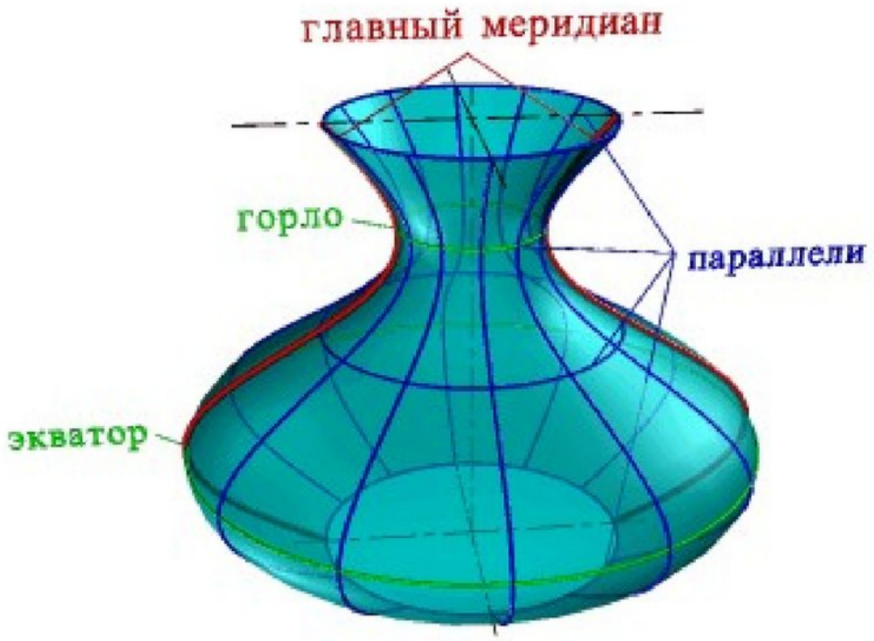

Рис. 5: Поверхность вращения

1.4.2 Формализация тела вращения

Допустим, что

$$g : [a, b] \longrightarrow R^2 \tag{3}$$

будет плоской параметрической кривой и пусть

$$g(t) = (g_1(t), g_2(t)) \tag{4}$$

Определим функцию

$$p : [a, b] \times [c, d] \longrightarrow R^3 \tag{5}$$

как

$$p(t, \theta) = (g_1(t), g_2(t)cos(\theta), g_2(t)sin(\theta)) \tag{6}$$

Подмножество

$$X = p([a, b] \times [c, d]) \subseteq R^3 \tag{7}$$

называется поверхностью вращения вокруг оси x для углов c и d относительно g.

Кривые

$$\gamma(t) = p(t, \theta) \tag{8}$$

для фиксированного θ являются меридианами поверхности вращения и кривые

$$\nu(\theta) = p(t, \theta) \tag{9}$$

для фиксированного t называются кругами широты.

Используя стандартную параметризацию $g(t) = (t,f(t))$ для графика f и замена t на x в уравнении 6, поверхность, полученная путем вращения графика f вокруг оси x, параметризуется с помощью формулы

$$p(x, \theta) = (x, f(x)cos(\theta), f(x)sin(\theta)) \tag{10}$$

Отсюда легко вычислить производные для этой поверхности

$$\frac{\delta p}{\delta x} = (1, f'(x)cos(\theta), f'(x)sin(\theta)) \quad (11)$$

$$\frac{\delta p}{\delta \theta} = (0, -f(x)sin(\theta), f(x)cos(\theta)) \quad (12)$$

Из этого можно сразу узнать касательные плоскости в каждой точке, потому что перекрестное произведение частных производных является нормальным вектором (при условии, что частные производные не обращаются в нуль).

1.5 Анализ алгоритмов удаления невидимых линий и поверхностей

При выборе алгоритма удаления невидимых линий и поверхностей учитывается особенность поставленной задачи - работа программы будет выполняться в реальном режиме при взаимодействии с пользователем. Этот факт предъявляет к алгоритму требование по скорости работы. Для выбора наиболее подходящего алгоритма следует рассмотреть уже имеющиеся алгоритмы удаления невидимых линий и поверхностей.

1.5.1 Алгоритм обратной трассировки лучей

Алгоритм работает в пространстве изображения[6].

Суть алгоритма: для определения цвета пиксела экрана через него из точки наблюдения проводится луч, ищется пересечение первым пересекаемым объектом сцены и определяется освещенность точки пересечения. Эта освещенность складывается из отраженной и преломленной энергий, полученных от источников света, а также отраженной и преломленной энергий, идущих от других объектов сцены. После определения освещенности най-

денной точки учитывается ослабление света при прохождении через прозрачный материал и в результате получается цвет точки экрана.

Преимущества:

- изображение, которое строится с учётом явлений дисперсии лучей, преломления, а также внутреннего отражения;
- возможность использования в параллельных вычислительных системах.

Недостатки:

- трудоёмкие вычисления[7];

1.5.2 Алгоритм, использующий Z-буфер

Алгоритм работает в пространстве изображения[8].

Сущность алгоритма: имеется 2 буфера - буфер кадра, который используется для запоминания цвета каждого пиксела изображения, а также z-буфер - отдельный буфер глубины, используемый для запоминания координаты z (глубины) каждого видимого пиксела изображения. В процессе работы глубина или значение z каждого нового пиксела, который нужно занести в буфер кадра, сравнивается с глубиной того пиксела, который уже занесен в z-буфер. Если это сравнение показывает, что новый пиксел расположен выше пиксела, находящегося в буфере кадра ($z > 0$), то новый пиксел заносится в цвет рассматриваемого пиксела заносится в буфер кадра, а координата z - в z-буфер. По сути, алгоритм является поиском по x и y наибольшего значения функции $z(x, y)$.

Преимущества:

- возможность обработки произвольных поверхностей, аппроксимируемых полигонами;
- отсутствие требования сортировки объектов по глубине.

Недостатки:

- отсутствие возможности работы с прозрачными и просвечивающими объектами (в классической версии).

1.5.3 Алгоритм Робертса

Алгоритм работает в объектном пространстве[9].

Суть алгоритма: алгоритм прежде всего удаляет из каждого тела те ребра или грани, которые экранируются самим телом. Затем каждое из видимых ребер каждого тела сравнивается с каждым из оставшихся тел для определения того, какая его часть или части, если таковые есть, экранируются этими телами.

Преимущества:

- реализации алгоритма, использующие предварительную приоритетную сортировку вдоль оси z и простые габаритные или минимаксные тесты, демонстрируют почти линейную зависимость от числа объектов[9].

Недостатки:

- вычислительная трудоёмкость алгоритма теоретически растет, как квадрат числа объектов[9];
- отсутствие возможности работы с прозрачными и просвечивающими объектами.

Вывод

В таблице 1 представлено сравнение алгоритмов[10] удаления невидимых линий и поверхностей (по каждому параметру составлен рейтинг: 1 - лучший алгоритм, 3 - худший). Так как главным требованием к алгоритму

является скорость работы, алгоритмы были оценены по следующим критериям:

- скорость работы (С);
- масштабируемость с ростом количества моделей (ММ);
- масштабируемость с увеличением размера экрана (МЭ);
- работа с фигурами вращения (ФВ).

Алгоритм	С	ММ	МЭ	ФВ
Z-буфера	1	2	1	1
Трассировка лучей	3	1	3	2
Робертса	2	3	1	3

Таблица 1: Сравнение алгоритмов удаления невидимых линий и поверхностей.

С учётом результатов в таблице 1 был выбран алгоритм **Z-буфера** удаления невидимых линий и поверхностей.

1.6 Анализ методов закрашивания

Методы закрашивания используются для затенения полигонов (или поверхностей, аппроксимированных полигонами) в условиях некоторой сцены, имеющей источники освещения.

1.6.1 Простая закраска

Суть алгоритма: вся грань закрашивается одним уровнем интенсивности, который зависит высчитывается по закону Ламберта[10]. При данной за-

краске все плоскости (в том числе и те, что аппроксимируют фигуры вращения), будут закрашены однотонно, что в случае с фигурами вращения будет давать ложные ребра.

Преимущества:

- используется для работы с многогранниками, обладающими преимущественно диффузным отражением.

Недостатки:

- плохо подходит для фигур вращения: видны ребра.

1.6.2 Закраска по Гуро

Суть алгоритма: билинейная интерполяция в каждой точке интенсивности освещения в вершинах[11].

Нормаль к вершине можно найти несколькими способами:

- интерполировать нормали прилегающих к вершине граней;
- использовать геометрические свойства фигуры (так, например, в случае со сферой ненормированный вектор нормали будет в точности соответствовать вектору от центра сферы до рассматриваемой точки).

После нахождения нормали ко всем вершинам находится интенсивность в каждой вершине по закону Ламберта. Затем алгоритм проходится сканирующими строками по рассматриваемому полигону для всех $y : y \in [y_{min}; y_{max}]$. Каждая сканирующая строка пересекает 2 ребра многоугольника, пусть для определённости это будут ребра через одноименные вершины: MN и KL. В точках пересечения высчитывается интенсивность путём интерполяции интенсивности в вершинах. Так, для точки пересечения с ребром MN интенсивность будет рассчитана как (13):

$$I_{MN} = \frac{l_1}{l_0} \cdot I_M + \frac{l_2}{l_0} \cdot I_N \tag{13}$$

где l_1 - расстояние от точки пересечения до вершины N, l_2 - расстояние от точки пересечения до вершины M, l_0 - длина ребра MN. Для точки пересечения сканирующей строки с ребром KL интенсивность высчитывается аналогично.

Далее, после нахождения точек пересечения, алгоритм двигается по Ox от левой точки пересечения X_{left} до правой точки пересечения X_{right} и в каждой точке $\mathcal{X}$ интенсивность рассчитывается как (14):

$$I_{\mathcal{X}} = \frac{\mathcal{X} - X_{left}}{X_{right} - X_{left}} \cdot I_{X_{right}} + \frac{X_{right} - \mathcal{X}}{X_{right} - X_{left}} \cdot I_{X_{left}} \tag{14}$$

Преимущества:

- преимущественно используется с фигурами вращения с диффузным отражением, аппроксимированными полигонами.

Недостатки:

- при закраске многогранников ребра могут стать незаметными.

1.6.3 Закраска по Фонгу

Суть алгоритма: данный алгоритм работает похожим на алгоритм Гуро образом, однако ключевым отличием является то, что интерполируются не интенсивности в вершинах, а нормали[11]. Таким образом, закон Ламберта в данном алгоритме применяется в каждой точке, а не только в вершинах, что делает этот алгоритм гораздо более трудоёмким, однако с его помощью можно гораздо лучше изображаются блики.

Преимущества:

- преимущественно используется с фигурами вращения с зеркальным отражением, аппроксимированными полигонами.

Недостатки:

- самый трудоёмкий алгоритм из рассмотренных[10].

Вывод

В таблице 2 представлено сравнение алгоритмов[10] закраски (по каждому параметру составлен рейтинг: 1 - лучший алгоритм, 3 - худший). Так как требованиями к алгоритму являются высокая скорость работы, а также возможность закраски фигур вращения с диффузными свойствами отражения, алгоритмы были оценены по следующим критериям:

- скорость работы (С);
- работа с фигурами вращения (ФВ);
- работа с фигурами со свойствами диффузного отражения (ДО).

Алгоритм	С	ФВ	ДО
Простой	1	3	1
Гуро	2	1	1
Фонга	3	1	3

Таблица 2: Сравнение алгоритмов закраски.

С учётом результатов в таблице 2 был выбран алгоритм закраски **Гуро**.

Вывод

В данном разделе были формально описаны тела и поверхности вращения, их структурные характеристики, по которым эти модели строятся, были рассмотрены алгоритмы удаления невидимых линий и поверхностей, методы закрашивания поверхностей. В качестве алгоритма удаления невидимых линий и поверхностей был выбран алгоритм Z-буфера, в качестве метода закрашивания был выбран алгоритм закраски Гуро.

2 Конструкторский раздел

2.1 Требования к программному обеспечению

Программа должна предоставлять доступ к функционалу:

- Добавление, удаление объектов/композитов;
- Вращение, масштабирование, перемещение объектов/композитов;
- Редактирование параметров (ширина, высота) объектов;
- Изменение положения источника света;
- Редактирование объектов (закраска нужным цветом);
- Редактирование композитов;
- Передвижение по сцене (перемещение и вращение камеры);

К программе предъявляются следующие требования:

- время отклика программы должно быть менее 1 секунды для корректной работы в интерактивном режиме;
- программа должна корректно реагировать на любые действия пользователя.

2.2 Разработка алгоритмов

2.2.1 Алгоритм Z-буфера

1) Всем элементам буфера кадра присвоить фоновое значение
2) Инициализировать Z буфер минимальными значениями глубины
3) Выполнить растровую развертку каждого многоугольника сцены:
   – Для каждого пикселя, связанного с многоугольником вычислить его глубину z(x, y)

– Сравнить глубину пикселя со значением, хранимым в Z буфере.

  Если $z(x, y) > z\_buf(x, y)$, тогда

  $z\_buf(x,y) = z(x,y), color(x, y) = colorOfPixel.$

4) Отобразить результат.

На рис. 6 изображена схема алгоритма Z-буфера

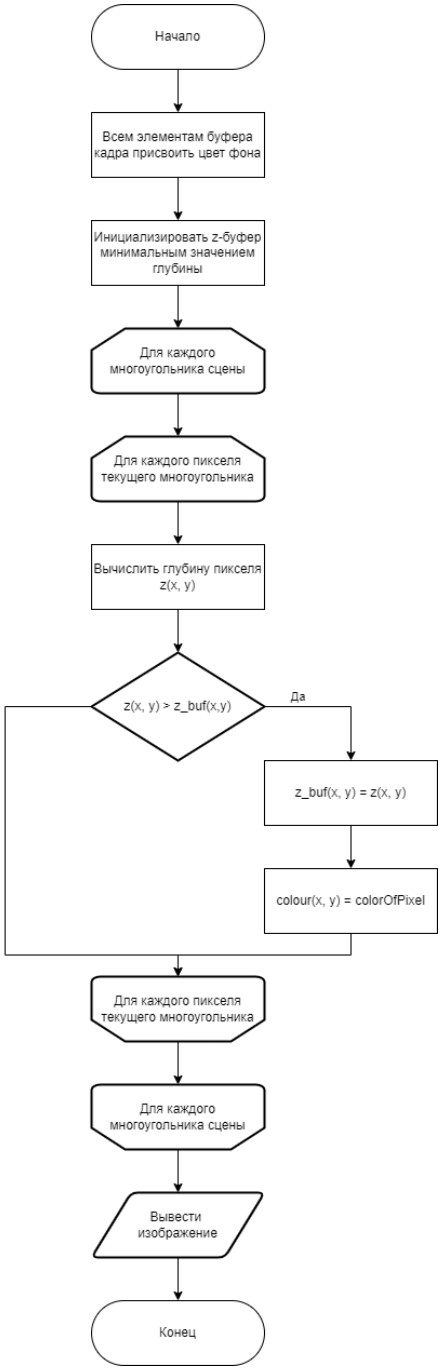

Рис. 6: Алгоритм Z-буфера

2.2.2 Модифицированный алгоритм, использующий z-буфер

1) Для каждого направленного источника света:
   - Инициализировать теневой z-буфер минимальным значением глубины;
   - Определить теневой z-буфер для источника.
2) Выполнить алгоритм z-буфера для точки наблюдения. При этом, если некоторая поверхность оказалась видимой относительно текущей точки наблюдения, то проверить, видима ли данная точка со стороны источников света
3) Для каждого источника света:
   - координаты рассматриваемой точки (x, y, z) линейно преобразовать из вида наблюдателя в координаты $(x0, y0, z0)$ на виде из рассматриваемого источника света;
   - сравнить значение $z\_shadowBuf(x0, y0)$ со значением $z0(x0, y0)$. Если $z0(x0, y0) < zshadowBuf(x0, y0)$, то пиксел высвечивается с учетом его затемнения, иначе точка высвечивается без затемнения
4) Отобразить результат.

2.3 Выбор используемых типов и структур данных

Для разрабатываемого ПО нужно будет реализовать следующие типы и структуры данных.

1) **Источник света** – направленностью света.
2) **Сцена** – задается объектами сцены.
3) **Объекты сцены** – задаются вершинами и гранями.

4) **Математические абстракции**.
   - **Точка** – хранит координаты x, y, z.
   - **Вектор** – хранит направление по x, y, z.
   - **Фигура** – хранит вершины, нормаль, цвет.
5) **Интерфейс** – используются библиотечные классы для предоставления доступа к интерфейсу.
6) **Графический обработчик** - абстрактная структура, выполянющая реализацию алгоритмов.
7) **Фабрики** для сцены, интерфейса, обработчика - для возможной подмены в ходе разработки или дополнения в дальнейшем.
8) **Композит** - объект, который будет содержать в себе другие объекты

3 Технологический раздел

3.1 Средства реализации

Основным языком программирования является мультипарадигменный язык Rust[12].

- Одно из главных достоинств данного языка это гарантия безопасной работы с памятью при помощи системы статической проверки ссылок, так называемый Borrow Checker[13].
- Отсутствие сброщика мусора, как следствие, более экономная работа с ресурсами.
- Встроенный компилятор, постовляемый совместно с пакетным менеджером Cargo.
- Кросс-платформенность, от UNIX и MacOS до Web.
- Крайне побдробные коды ошибок и документация от разработчиков языка.
- Важно отметить, что язык программирования Rust сопоставим по скорости с такими языками как C и C++, предоставляя в то же время более широкий функционал для тестирования кода и контроля памяти.

  Также были выбраны следующие библиотеки:
- В качестве графического интерфейса была выбрана библиотека egui[14] (или иначе crate в контексте языка Rust)
- Для рендера изображения была выбрана библиотека tiny-skia[15], предоставляющий быстрый CPU-рендеринг
- Помимо этого egui дает инструментарий для запуска приложения в браузере при непосредственном участии WebAssembly при практически нулевых затратах со стороны программиста.
- Для тестирования ПО использовались инструменты Cargo[16] — па-

кетного менеджера языка Rust, поставляемого вместе с компилятором из официального источника.

Среда разработки:

В процессе разработки был использован инструмент LSP[17] (англ. *Language Server Protocol*), а в частности его реализацию в виде Rust Analyzer [18], позволяющий форматировать исходные коды, а также в процессе их написания обнаружить наличие синтаксических ошибок и некоторых логических, таких как, например, нарушение правила владения[19].

В качестве среды разработки был выбран текстовый редактор VIM[20], поддерживающий возможность установки плагинов[21], в том числе для работы с LSP[17].

3.2 Структура классов

На рисунках 7 - **??** представлена структура реализуемых классов.

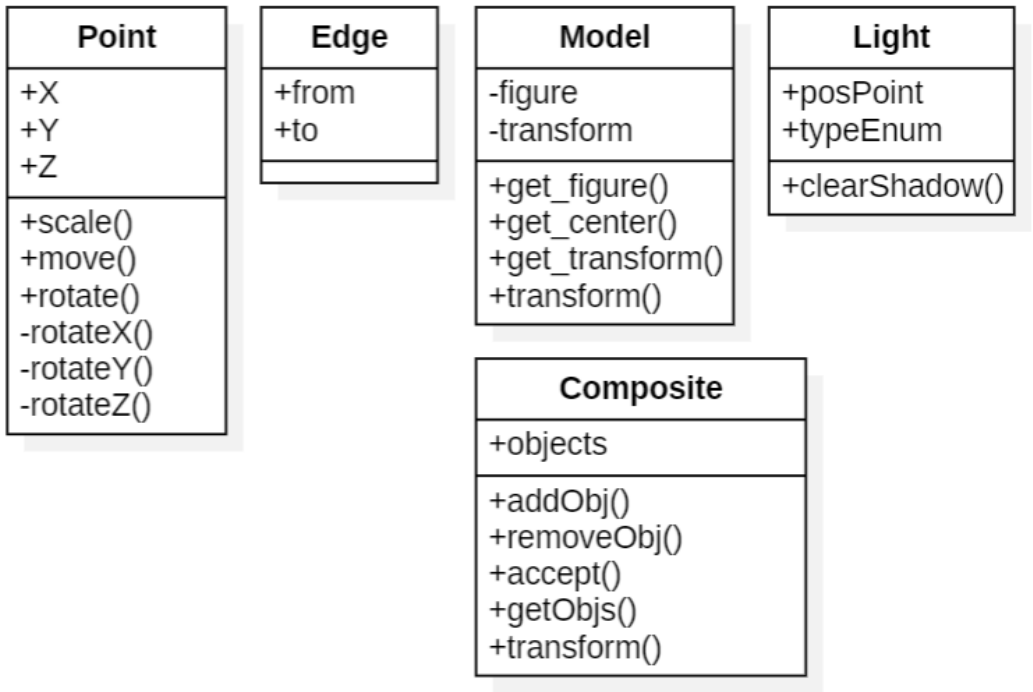

Рис. 7: Структура классов-объектов

- Point – класс точки трехмерного пространства. Хранит координаты в пространстве, владеет методами преобразований точки.
- Edge – класс грани. Хранит номера задействованных в грани вершин.
- Light – класс источника света.
- Model - класс модели. Скрывает конкретную реализацию модели(фигуры) и предоставляет единый интерфейс для работы с ней. Владеет методами преобразования модели, а также методами для получения информации о модели.
- Composite - класс композита. Хранит в себе набор моделей, владеет методами для работы с ними.

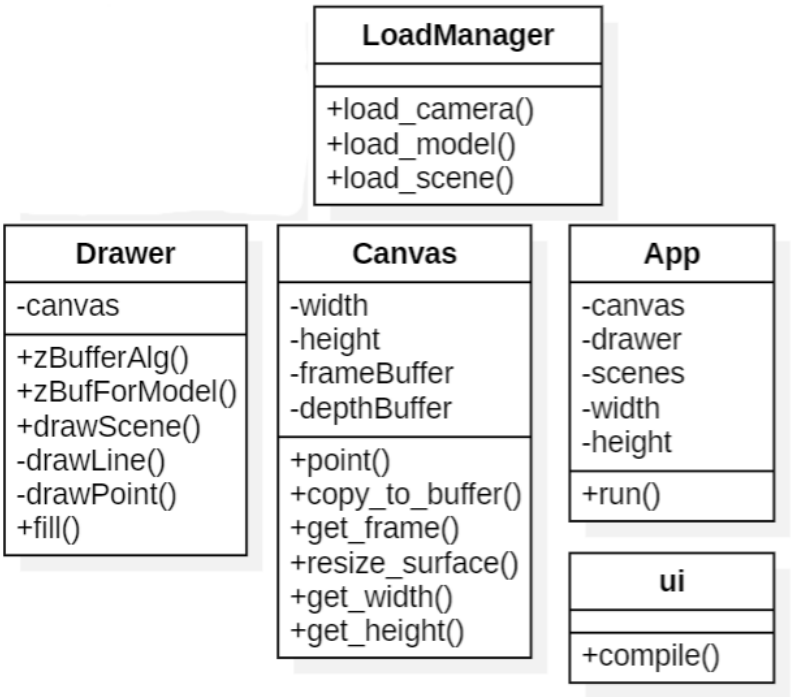

Рис. 8: Структура классов

- Drawer – класс, отвечающий за растеризацию сцены. Хранит полотно для отрисовки. Владеет методами алгоритма теневого z-буфера и формирования объекта для отображения рисунка в главном приложении.
- App – точка входа в программу.
- Ui - класс, отвечающий за отображение графического интерфейса.

- TransformManager – абстракция, содержащия методы трансформмации объектов.
- LoadManager - абстракция, содержащия методы загрузки объектов.
- Canvas - класс, отвечающий за отображение сцены.

3.3 Реализация алгоритмов

В листинге 1 представлена реализация Z-буффера и Гуро на языке Rust. В листинге 2 представлена реализация 3D модели.

Листинг 1: Реализация алгоритмов компьютерной графики.

```
1   fn bresenham(&mut self, mut start: Point3<i32>, end: Point3<i32>, color: [u8; 4]) {
2       if !self.check_pos3_all([start, end].into_iter()) {
3           return;
4       }
5       dbg!(start, end);
6       let mut canvas = self.canvas.as_ref().borrow_mut();
7       canvas.point(
8           start.x, //.checked_div(start.z).unwrap_or(start.x),
9           start.y, //.checked_div(start.z).unwrap_or(start.y),
10          color, 1.0,
11      );
12      let dx = (end.x - start.x).abs();
13      let dy = (end.y - start.y).abs();
14      let dz = (end.z - start.z).abs();
15      let xs = if end.x > start.x { 1 } else { -1 };
16      let ys = if end.y > start.y { 1 } else { -1 };
17      let zs = if end.z > start.z { 1 } else { -1 };
18      // Driving axis is X-axis"
19      if dx >= dy && dx >= dz {
20          let mut p1 = 2 * dy - dx;
21          let mut p2 = 2 * dz - dx;
22          while start.x != end.x {
23              start.x += xs;
24              if p1 >= 0 {
25                  start.y += ys;
```

```
                p1 -= 2 * dx;
            }
            if p2 >= 0 {
                start.z += zs;
                p2 -= 2 * dx;
            }
            p1 += 2 * dy;
            p2 += 2 * dz;
            canvas.point(
                start.x, //.checked_div(start.z).unwrap_or(start.x),
                start.y, //.checked_div(start.z).unwrap_or(start.y),
                color, 1.0,
            );
        }

    // Driving axis is Y-axis"
    } else if dy >= dx && dy >= dz {
        let mut p1 = 2 * dx - dy;
        let mut p2 = 2 * dz - dy;
        while start.y != end.y {
            start.y += ys;
            if p1 >= 0 {
                start.x += xs;
                p1 -= 2 * dy;
            }
            if p2 >= 0 {
                start.z += zs;
                p2 -= 2 * dy;
            }
            p1 += 2 * dx;
            p2 += 2 * dz;
            canvas.point(
                start.x, //.checked_div(start.z).unwrap_or(start.x),
                start.y, //.checked_div(start.z).unwrap_or(start.y),
                color, 1.0,
            );
        }

    // Driving axis is Z-axis"
    } else {
```

```
66              let mut p1 = 2 * dy - dz;
67              let mut p2 = 2 * dx - dz;
68              while start.z != end.z {
69                  start.z += zs;
70                  if p1 >= 0 {
71                      start.y += ys;
72                      p1 -= 2 * dz;
73                  }
74                  if p2 >= 0 {
75                      start.x += xs;
76                      p2 -= 2 * dz;
77                  }
78                  p1 += 2 * dy;
79                  p2 += 2 * dx;
80                  canvas.point(
81                      start.x, //.checked_div(start.z).unwrap_or(start.x),
82                      start.y, //.checked_div(start.z).unwrap_or(start.y),
83                      color, 1.0,
84                  );
85              }
86          }
87      }
88
89      fn reset(&mut self) {
90          self.z_buffer = vec![
91              vec![f64::MAX; self.canvas.borrow().width() as usize];
92              self.canvas.borrow().height() as usize
93          ];
94      }
95
96      pub fn transform_and_add(
97          &mut self,
98          (points, normals, triangles): &(
99              &[&Vec<Point<f64>>],
100             &[&Vec<Point<f64>>],
101             &[&Vec<Triangle>],
102         ),
103         light_source: Point<f64>,
104         color: [u8; 4],
105     ) {
```

```
106         // for every triangle (3 points + 3 normal points):
107         for ((p, n), t) in points.iter().zip(normals.iter()).zip(triangles.iter()) {
108             for (triangle, normal) in t.iter().zip(n.iter()) {
109                 // transform new point
110                 let current_window = [p[triangle.a()], p[triangle.b()], p[triangle.c()]];
111                 // transform_and_normalize(new_point, new_normal, matrix);
112                 // check if any part of triangle visible and triangle isn't rotated to
                        background
113                 if self.check_pos_all(current_window.into_iter())
114                     && self.check_normals_all([*normal].into_iter())
115                 {
116                     // add transformed triangle polygon to buffer
117                     self.add_polygon(current_window, normal, &light_source, color);
118                 }
119             }
120         }
121     }
122
123     fn add_polygon(
124         &mut self,
125         points: [Point<f64>; 3],
126         mut normal: &Point<f64>,
127         light_source: &Point<f64>,
128         color: [u8; 4],
129     ) {
130         // cast Y coordinate to integer (coordinates of the screen are integers)
131         // find brightnesses for all vertexes for furhter processing by Gouraud algorithm
132         let brightnesses = self.find_brightnesses(points, normal, light_source);
133         // divide triangle on 2 pairs of sections, which make up 2 triangles with
134         // parallel to X axis edge
135         // let points: Vec<Point<f64>> = points
136         // .iter()
137         // .map(|point| self.pov.as_ref().unwrap().borrow().project2(&point))
138         // .collect();
139         let mut int_points: Vec<Point3<i64>> = points
140             .iter()
141             .map(|point| Point3::new(point.x() as i64, point.y() as i64, point.z() as i64))
142             .collect();
143         // sort points by Y coordinate
144         self.sort_by_y(&mut int_points, normal);
```

```
145
146         let sections = self.divide_on_sections(&int_points, brightnesses);
147         self.process_sections(sections, color);
148     }
149
150     // application of Gouraud and Z-buffer algorithms for 2 processed triangles
151     fn process_sections(&mut self, mut sections: [Section; 4], color: [u8; 4]) {
152         let mut canvas_ref = self.canvas.borrow_mut();
153         for pair in sections.chunks_mut(2) {
154             if pair[0].x_start > pair[1].x_start {
155                 continue;
156             }
157
158             if pair[0].y_start < 0 {
159                 let diff = (-pair[0].y_start) as f64;
160                 for sec in pair.iter_mut() {
161                     sec.x_start += diff * sec.x_step;
162                     sec.br_start += diff * sec.br_step;
163                     sec.z_start += diff * sec.z_step;
164                 }
165                 pair[0].y_start = 0;
166             }
167             let height = canvas_ref.height();
168             let width = canvas_ref.width();
169             for y in (pair[0].y_start..=pair[0].y_end)
170                 .filter(|&elem| elem < height as i64)
171                 .map(|y| y as usize)
172             {
173                 let x_from = f64::round(pair[0].x_start) as usize;
174                 let x_to = f64::round(pair[1].x_start) as usize;
175                 let diff_x = (x_to - x_from) as f64;
176
177                 let mut br = pair[0].br_start;
178                 let br_diff = (pair[1].br_start - br) / diff_x;
179                 let mut z = pair[0].z_start;
180                 let z_diff = (pair[1].z_start - z) / diff_x;
181
182                 for x in (x_from..=x_to).filter(|&x| x < width as usize) {
183                     if z < self.z_buffer[y][x] {
184                         self.z_buffer[y][x] = z;
```

```
                    canvas_ref.point(x as i32, y as i32, color, br);
                }

                br += br_diff;
                z += z_diff;
            }

            for sec in pair.iter_mut() {
                sec.x_start += sec.x_step;
                sec.br_start += sec.br_step;
                sec.z_start += sec.z_step;
            }
        }
    }
}

fn sort_by_y(&mut self, int_points: &mut [Point3<i64>], normals: &Point<f64>) {
    for (&i, &j) in [0, 0, 1].iter().zip([2, 1, 2].iter()) {
        let condition = {
            let (a, b) = (&int_points[i], &int_points[j]);
            a.y > b.y || a.y == b.y && a.x > b.x
        };
        if condition {
            int_points.swap(i, j);
        }
    }
}

fn find_brightnesses(
    &mut self,
    points: [Point<f64>; 3],
    normal: &Point<f64>,
    light_source: &Point<f64>,
) -> [f64; 3] {
    let mut lsvs = Vec::with_capacity(3);
    for i in 0..3 {
        let mut lsv = points[i] - *light_source;
        lsv.normalize();
        lsvs.push(lsv);
    }
```

```
        [
            ZERO_BRIGHTNESS + BRIGHTNESS_RANGE * (normal.scalar_mul(&lsvs[0])),
            ZERO_BRIGHTNESS + BRIGHTNESS_RANGE * (normal.scalar_mul(&lsvs[1])),
            ZERO_BRIGHTNESS + BRIGHTNESS_RANGE * (normal.scalar_mul(&lsvs[2])),
        ]
    }
```

Листинг 2: Реализация 3D модели.

```
#[derive(Clone, Debug)]
pub struct FrameFigure {
    points: Vec<Point<f64>>,
    edges: Vec<Edge>,
    cached_points: Vec<Point<f64>>,
    triangles: Vec<Triangle>,
    normals: Vec<Point<f64>>,
    state: State,
    color: [u8; 4],
}

#[derive(Clone, Debug)]
pub struct FrameModel {
    name: String,
    figures: Vec<Rc<RefCell<FrameFigure>>>,
    transform: Matrix4<f64>,
    is_cached: bool,
}

impl Default for FrameFigure {
    fn default() -> Self {
        Self::new()
    }
}

impl FrameFigure {
    #[must_use]
    pub const fn new() -> Self {
        Self {
            points: Vec::new(),
            edges: Vec::new(),
            cached_points: vec![],
```

```
            triangles: vec![],
            normals: vec![],
            state: State::Union,
            color: [255; 4],
        }
    }

    #[must_use]
    pub fn new_with_points(points: Vec<Point<f64>>) -> Self {
        Self {
            points,
            edges: Vec::new(),
            cached_points: vec![],
            triangles: vec![],
            normals: vec![],
            state: State::Union,
            color: [255; 4],
        }
    }

    #[must_use]
    pub fn new_with_edges(edges: Vec<Edge>) -> Self {
        Self {
            points: Vec::new(),
            edges,
            cached_points: vec![],
            triangles: vec![],
            normals: vec![],
            state: State::Union,
            color: [255; 4],
        }
    }

    #[must_use]
    pub fn new_with_points_and_edges(points: Vec<Point<f64>>, edges: Vec<Edge>) -> Self {
        Self {
            points,
            edges,
            cached_points: vec![],
            triangles: vec![],
```

```
            normals: vec![],
            state: State::Union,
            color: [255; 4],
        }
    }

    getter!(color: [u8; 4]);

    setter!(
        points: Vec<Point<f64>>,
        edges: Vec<Edge>,
        cached_points: Vec<Point<f64>>,
        triangles: Vec<Triangle>,
        normals: Vec<Point<f64>>,
        color: [u8; 4]
    );

    getter_ref!(
        points: Vec<Point<f64>>,
        edges: Vec<Edge>,
        cached_points: Vec<Point<f64>>,
        triangles: Vec<Triangle>,
        normals: Vec<Point<f64>>
    );

    #[must_use]
    pub const fn name(&self) -> &str {
        "FrameFigure"
    }

    #[must_use]
    pub fn state(&self) -> State {
        self.state.clone()
    }

    pub fn triangles_mut(&mut self) -> &mut Vec<Triangle> {
        &mut self.triangles
    }

    pub fn points_mut(&mut self) -> &mut Vec<Point<f64>> {
```

```
113         &mut self.points
114     }
115
116     pub fn edges_mut(&mut self) -> &mut Vec<Edge> {
117         &mut self.edges
118     }
119
120     pub fn add_point(&mut self, point: Point<f64>) {
121         self.points.push(point);
122     }
123
124     pub fn add_edge(&mut self, edge: Edge) {
125         self.edges.push(edge);
126     }
127
128     pub fn remove_point(&mut self, index: usize) {
129         self.points.remove(index);
130     }
131
132     pub fn remove_edge(&mut self, index: usize) {
133         self.edges.remove(index);
134     }
135
136     #[must_use]
137     pub fn point(&self, index: usize) -> &Point<f64> {
138         &self.points[index]
139     }
140
141     #[must_use]
142     pub fn edge(&self, index: usize) -> &Edge {
143         &self.edges[index]
144     }
145
146     pub fn point_mut(&mut self, index: usize) -> &mut Point<f64> {
147         &mut self.points[index]
148     }
149
150     pub fn edge_mut(&mut self, index: usize) -> &mut Edge {
151         &mut self.edges[index]
152     }
```

```
153
154     pub fn compute_normals(&mut self) -> &Vec<Point<f64>> {
155         for triangle in &self.triangles {
156             let a = self.points[triangle.a()];
157             let b = self.points[triangle.b()];
158             let c = self.points[triangle.c()];
159
160             let ab = b - a;
161             let ac = c - a;
162             let mut normal = Point::new(
163                 ab.y().mul_add(ac.z(), -ab.z() * ac.y()),
164                 ab.z().mul_add(ac.x(), -ab.x() * ac.z()),
165                 ab.x().mul_add(ac.y(), -ab.y() * ac.x()),
166             );
167             normal.normalize();
168             self.normals.push(normal);
169         }
170
171         &self.normals
172     }
173
174     #[must_use]
175     pub fn center(&self) -> Point<f64> {
176         let mut max = self.points[0];
177         let mut min = self.points[0];
178
179         for point in &self.points {
180             max = Point::new(
181                 max.x().max(point.x()),
182                 max.y().max(point.y()),
183                 max.z().max(point.z()),
184             );
185             min = Point::new(
186                 min.x().min(point.x()),
187                 min.y().min(point.y()),
188                 min.z().min(point.z()),
189             );
190         }
191
192         (max + min) / Point::new(2.0, 2.0, 2.0)
```

```
193     }
194
195     fn update_cached(&mut self, transform: &Matrix4<f64>) {
196         self.cached_points = self
197             .points
198             .clone()
199             .into_iter()
200             .map(|pnt| pnt.transform(transform))
201             .collect();
202         self.compute_normals();
203     }
204 }
205
206 impl FrameModel {
207     pub(crate) fn new(figure: Rc<RefCell<FrameFigure>>, name: String) -> Self {
208         Self {
209             figures: vec![figure],
210             transform: Matrix4::identity(),
211             name,
212             is_cached: false,
213         }
214     }
215 }
216
217 impl Model for FrameModel {
218     type Output = FrameFigure;
219     fn figures(&self) -> Vec<Rc<RefCell<Self::Output>>> {
220         self.figures.clone()
221     }
222
223     fn add_figure(&mut self, model: Rc<RefCell<Self::Output>>) {
224         let new_points = model.borrow().cached_points.clone();
225         model.borrow_mut().points = new_points;
226         for figure in &self.figures {
227             let new_points = figure.borrow().cached_points.clone();
228             figure.borrow_mut().points = new_points;
229         }
230         self.transform = Matrix4::identity();
231         self.figures.push(model);
232     }
```

```
233
234     fn center(&self) -> Point<f64> {
235         self.true_center().transform(&self.transform)
236     }
237
238     fn true_center(&self) -> Point<f64> {
239         let mut center = Point::default();
240         for figure in &self.figures {
241             center += figure.borrow().center();
242         }
243
244         center
245             / Point::new(
246                 self.figures.len() as f64,
247                 self.figures.len() as f64,
248                 self.figures.len() as f64,
249             )
250     }
251
252     fn name(&self) -> String {
253         self.name.clone()
254     }
255
256     fn transform(&self) -> Matrix4<f64> {
257         self.transform
258     }
259
260     fn transform_self(&mut self, transform: Matrix4<f64>) {
261         self.is_cached = false;
262         self.transform = self.transform * transform;
263     }
264     fn transform_first(&mut self, transform: Matrix4<f64>) {
265         self.is_cached = false;
266         self.transform = transform * self.transform;
267     }
```

3.4 Интерфейс программного обеспечения

При запуске программы перед пользователем предстает пустая сцена. Для операций над объектами, их модифицирования, для управления камерой или освещением в левой части интерфейса определены соответствующие разделы (рисунок 9).

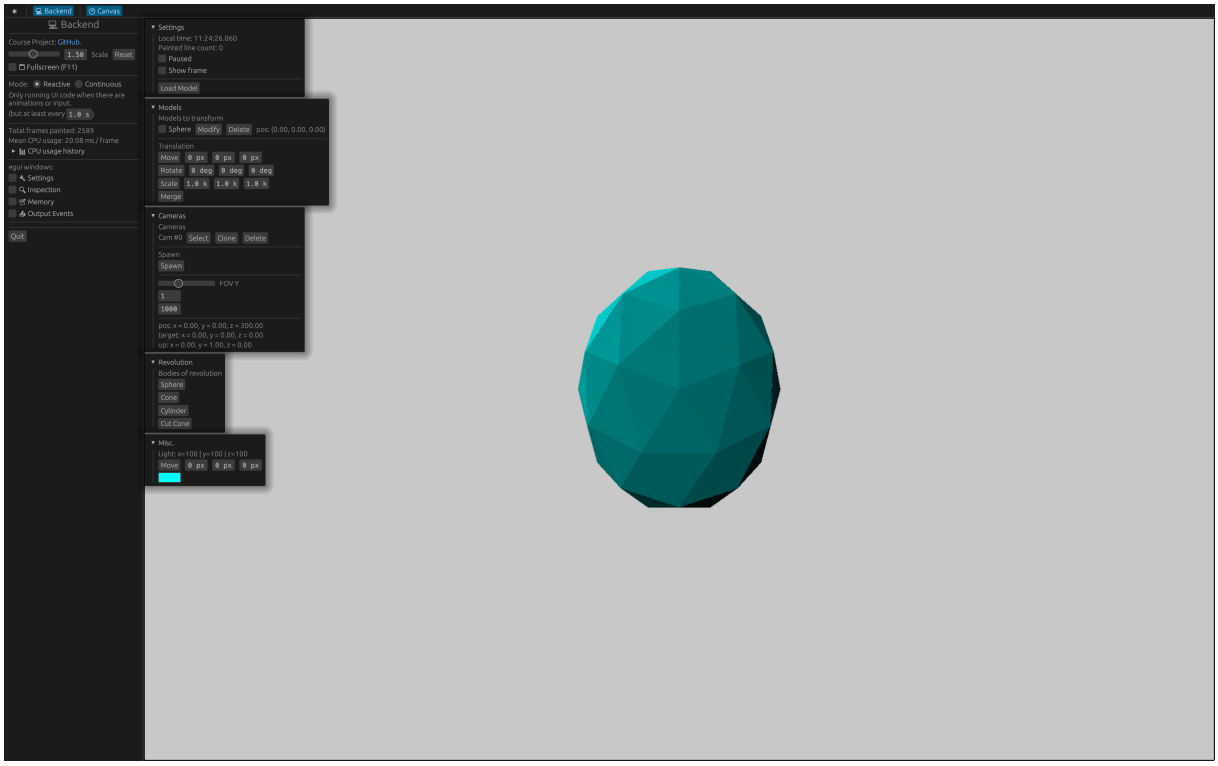

Рис. 9: GUI

Для создания модели пользователю необходимо нажать на соответствующий раздел и выбрать одно из предложенных тел. Далее в разделе моделей пользоватеь может каких-либо образом изменить необходимую модель. (рисунки 10-11).

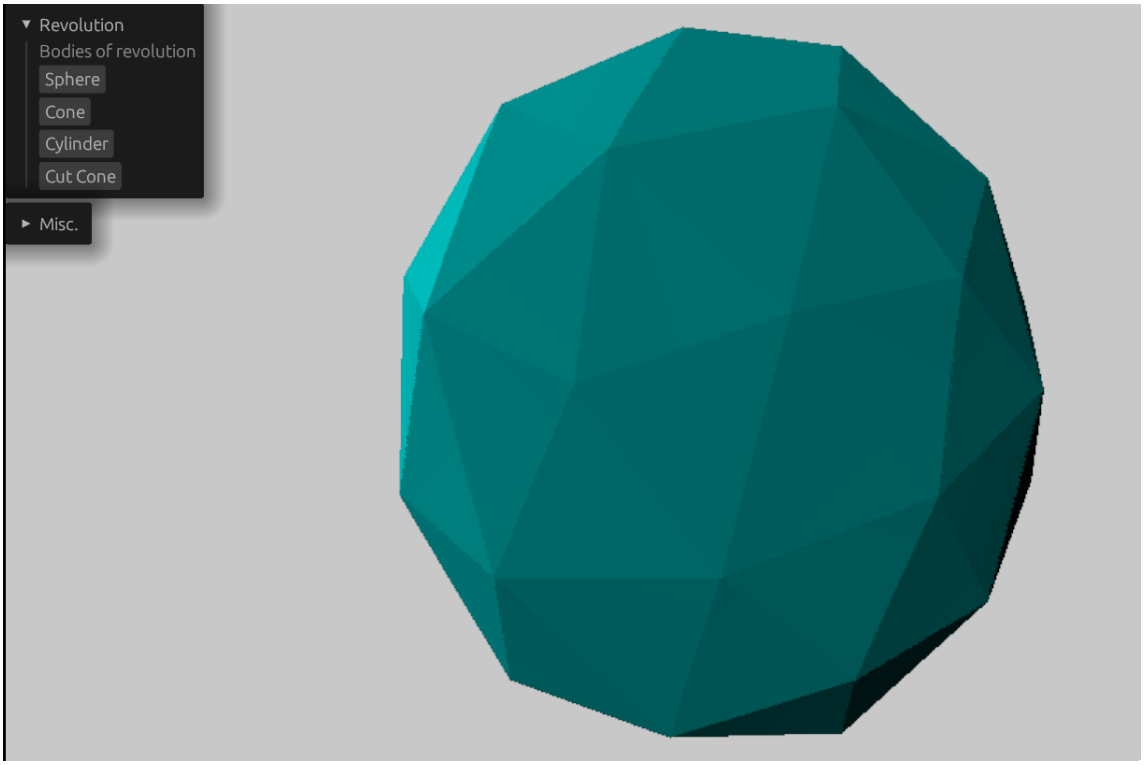

Рис. 10: Раздел создание модели

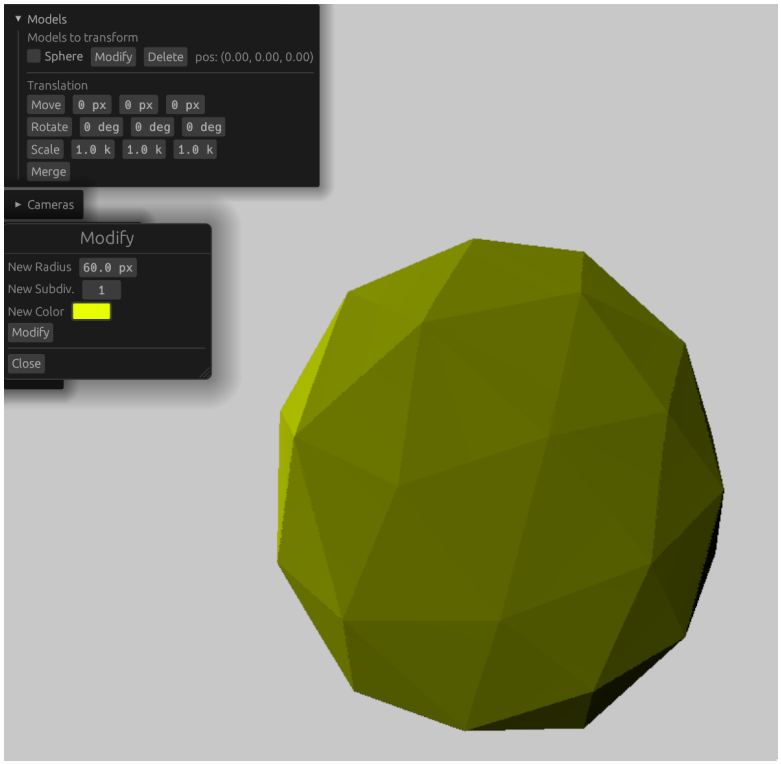

Рис. 11: Раздел модифицирования модели

Для перемещения по сцене используются клавиши: W – вперед, S – назад, A – влево, D – вправо, Стрелка вверх - вверх, Стрелка вниз - вниз, R и

T - вращение камеры, Z и C - поворот камеры.

В разделе камеры находятся следующие параметры: расстояния до ближней и дальней плоскостей пирамиды видимости, угол обзора, позиция камеры в пространстве, углы поворота, скорость перемещения (рисунок 12).

Рис. 12: Раздел камеры

На рисунке 13 приведен пример работы программы.

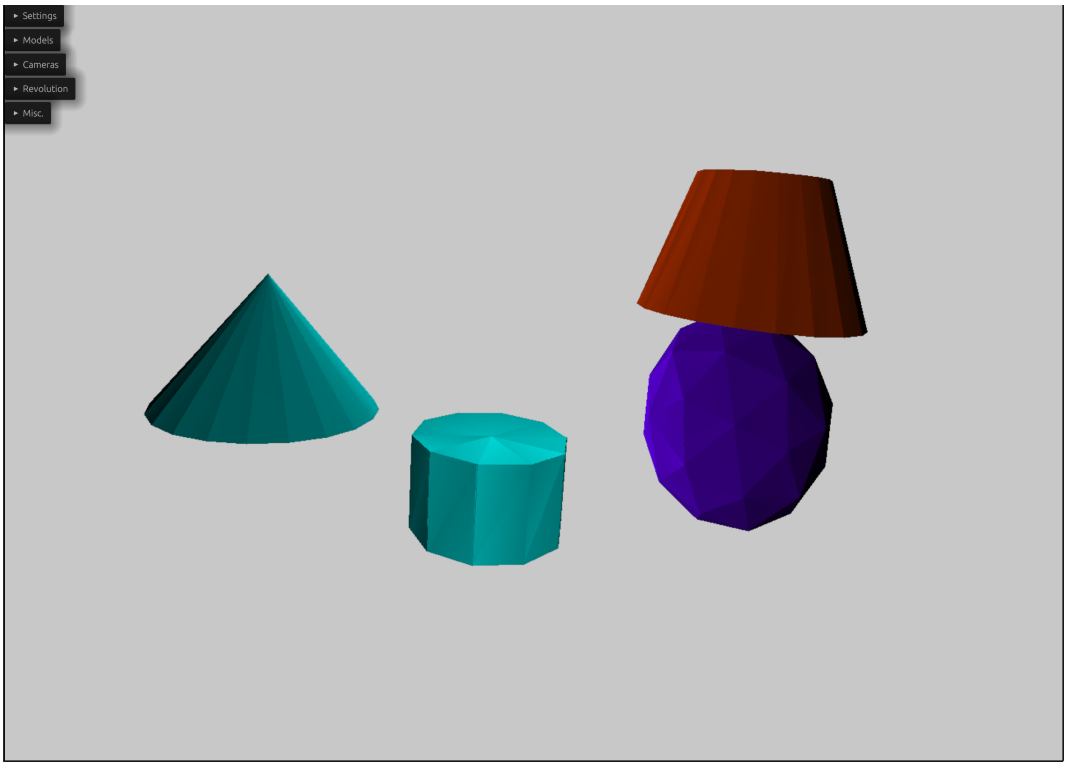

Рис. 13: Тестовая сцена

Вывод

В данном разделе были рассмотрены средства, с помощью которых было реализовано ПО, а также представлены структуры классов и листинги кода с реализацией алгоритмов компьютерной графики.

4 Исследовательский раздел

В данном разделе приведены технические характеристики устройства, на котором проводилось измерение времени работы программного обеспечения, а также результаты замеров времени.

4.1 Постановка эксперимента

Целью эксперимента является провести анализ скорости работы алгоритма генерации изображения с использованием алгоритма Z-буфером.

4.1.1 Технические характеристики

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система: Arch Linux [22] 64-bit.
- Количество ядре: 4 физических и 8 логических ядер.
- Оперативная память: 16 Гб.
- Процессор: 11th Gen Intel® Core™ i5-11320H @ 3.20 ГГц[23].

Во время тестирования устройство было нагружено только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.1.2 Результаты эксперимента

Для исследования зависимости времени обработки изображения от числа объектов на сцене, использовались объекты с фиксированным количеством

граней, каждый объект имел освещенную и затененную части. Количество объектов менялось на сцене от 100 до 1000 с шагом 100. Результаты проведенного исследования представлены.

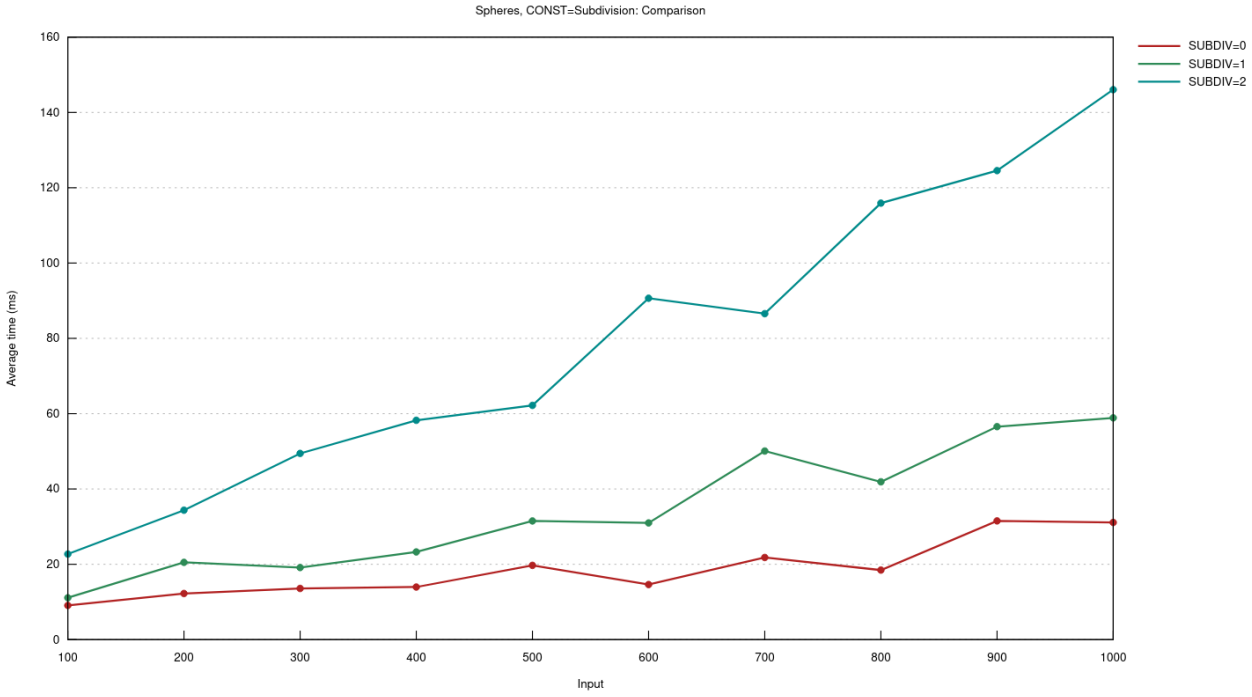

Рис. 14: График зависимости времени отрисовки от числа объектов

Как видно из графика, время визуализации сцены зависит от количества объектов линейно, независимо от числа граней.

Следующим этапом исследования разработанной программы является исследование зависимости времени построения сцены от количества граней при фиксированном количестве объектов. В ходе эксперимента количество граней менялось по закону $X * 2^n$, где X — количество граней базовой модели, для наглядности были использованы различные модели.

Из проведенного эксперимента можно сделать вывод, что время визуализации сцены линейно зависит от числа граней объектов, поскольку сам график соответствует графику увеличения граней.

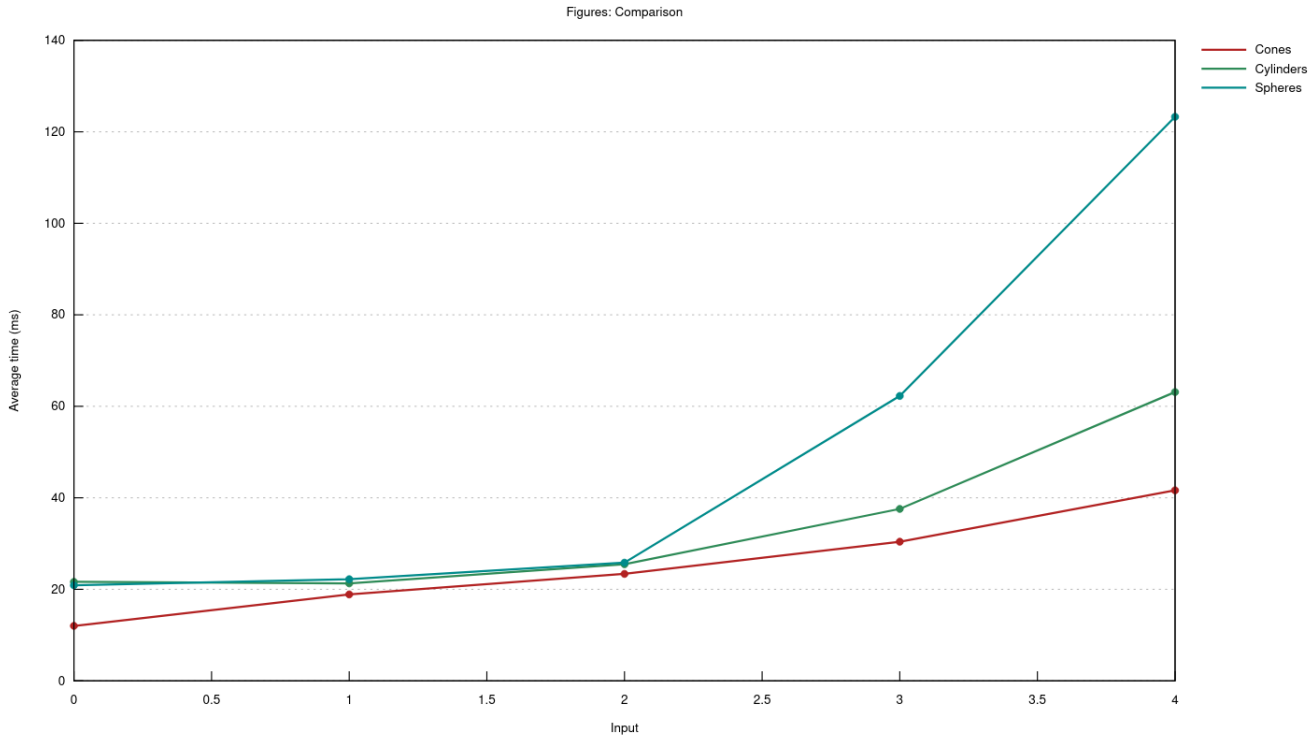

Рис. 15: График зависимости времени отрисовки от числа граней

Вывод

В данном разделе приведены результаты работы программного обеспечения и проведен эксперимент с использованием библиотеки Criterion, функции которой использовались для определения эффективности работы программы по времени.

Результаты эксперимента совпали с ожидаемыми, так как в ходе эксперимента было установлено, что время работы увеличивается линейно с увеличением количества объектов на сцене и количеством ребер у объекта.

ЗАКЛЮЧЕНИЕ

Целью данного курсового проекта была достигнута, то есть был разработан программный продукт, позволяющий создавать и редактировать композиции из трехмерных графических тел вращения. Также ПО предоставляет возможности настройки геометрических характеристик объектов, положения камеры и положения источника освещения.

Для достижение цели были выполнены следующие задачи:

- проведен анализ существующих алгоритмов компьютерной графики, использующие для создание реалистичной модели и трехмерной сцены;
- выбраны наиболее подходящие алгоритмы (алгоритмы удаления невидимых линий, методы закраски, модели освещения) для решения поставленной задачи;
- спроектированы архитектура и графический интерфейс программы;
- выбраны средства реализации программного обеспечения;
- разработано ПО и реализация выбранных алгоритмов и структур данных;
- проведены замеры временных характеристик разработанного программного обеспечения.

В процессе исследовательской работы было выяснено, что полученная модель имеет линейную зависимость времени отрисовки от количества граней и количества объектов. Данное свойство говорит о том, что программа будет хорошо работать со средним количеством объектов, однако для большего их количества понадобятся дополнительные оптимизации вычислений. В частности, быстродействие низкоуровневых частей программы, отвечающих за растеризацию полигонов, может быть улучшено за счет замены их программной реализации на поддерживаемую аппаратно из интерфейсов графических движков, например, OpenGL или DirectX.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Т.О. Перемитина. Компьютерная графика : учебное пособие. — Томск: Эль Контент, 2012. с. 114.

[2] Middlemiss; Marks; Smart. ’15-4. Surfaces of Revolution’. Analytic Geometry (3rd ed.). p. 378. https://lccn.loc.gov/68015472. 1968.

[3] Юрьевич Аксенов Алексей. Модели и методы обработки и представления сложных пространственных объектов. — СПИИРАН, 2015. с. 110.

[4] Д. Роджерс. Алгоритмические основы машинной графики: Пер. с англ. — СПб: БХВ-Петербург, 1989. с. 512.

[5] Н. Порев В. Компьютерная графика. — СПб.: БХВ-Петербург, 2002. с. 429.

[6] Трассировка лучей в реальном времени. Режим доступа: https://www.ixbt.com/3dv/directx-raytracing.html (дата обращения: 13.08.2022).

[7] Cost Analysis of a Ray Tracing Algorithm. Режим доступа: https://www.graphics.cornell.edu/ bjw/mca.pdf (дата обращения: 05.09.2022).

[8] Алгоритм Z-буфера. Режим доступа: http://compgraph.tpu.ru/zbuffer.htm (дата обращения: 21.08.2022).

[9] Алгоритм Робертса. Режим доступа: http://compgraph.tpu.ru/roberts.htm (дата обращения: 14.05.2022).

[10] Д. Роджерс. Алгоритмические основы машинной графики. 1989.

[11] Модели затенения. Плоская модель. Затенение по Гуро и Фонгу. Режим доступа: https://compgraphics.info/3D/lighting/shading_model.php (дата обращения: 18.06.2022).

[12] Rust [Электронный ресурс]. Режим доступа: https://www.rust-lang.org/. Дата обращения: 19.10.2022.

[13] Rust Borrow Checker [Электронный ресурс]. https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

[14] egui: an easy-to-use immediate mode GUI in Rust [Электронный ресурс]. https://github.com/emilk/egui.

[15] A tiny Skia subset ported to Rust [Электронный ресурс]. https://github.com/RazrFalcon/tiny-skia.

[16] The Cargo Book. Режим доступа: https://doc.rust-lang.org/cargo/ (дата обращения: 21.07.2022).

[17] Language Server Protocol (LSP). Режим доступа: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/ (дата обращения: 21.12.2022).

[18] Rust Analuzer. Режим доступа: https://rust-analyzer.github.io/ (дата обращения: 21.12.2022).

[19] Rustbook. What is Ownership? Режим доступа: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (дата обращения: 20.11.2022).

[20] VIM the editor. Режим доступа: https://www.vim.org/ (дата обращения: 24.11.2022).

[21] VimAwesome. Режим доступа: https://vimawesome.com/ (дата обращения: 24.11.2022).

[22] Arch Linux [Электронный ресурс]. Режим доступа: https://archlinux.org/. Дата обращения: 19.10.2022.

[23] Процессор Intel® Core™ i5-11320H [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu.html. Дата обращения: 19.10.2022.

ПРИЛОЖЕНИЕ А

Презентация к курсовой работе

Презентация содержит 13 слайдов.