|  |  |
| --- | --- |
| *Gerb-BMSTU_01* | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ  *Информатика и системы управления*

КАФЕДРА *Программное обеспечение ЭВМ и информационные технологии*

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

Программа для создания сцены из файлов моделей с применением текстур и учитыванием освещения

Студент ИУ7-55Б  К. К. Руднев

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы  З.Н. Русакова

(Подпись, дата) (И.О.Фамилия)

Консультант  \_\_\_\_\_\_\_\_\_\_\_\_

(Подпись, дата) (И.О.Фамилия)

Москва, 2019 г.

**Содержание**

[Введение 4](#_Toc26974075)

[1. Аналитический раздел 5](#_Toc26974076)

[1.1 Алгоритмы растеризации отрезков 5](#_Toc26974077)

[1.1.1 Алгоритм ЦДА 5](#_Toc26974078)

[1.1.2 Алгоритм Брезенхема 5](#_Toc26974079)

[1.1.3 Алгоритм Ву со сглаживанием 8](#_Toc26974080)

[1.1.4 Вывод по выбор алгоритма растеризации отрезков 9](#_Toc26974081)

[1.2 Алгоритмы удаления невидимых линий и поверхностей 9](#_Toc26974082)

[1.2.1 Алгоритм Робертса 9](#_Toc26974083)

[1.2.2 Алгоритм Варнока 11](#_Toc26974084)

[1.2.3 Алгоритм художника 13](#_Toc26974085)

[1.2.4 Алгоритм обратной трассировки лучей 14](#_Toc26974086)

[1.2.5 Алгоритм удаления невидимых граней с использованием Z-буфера 16](#_Toc26974087)

[1.2.6 Вывод по выбору алгоритма удаления невидимых линий и поверхностей 17](#_Toc26974088)

[1.3 Алгоритмы закраски 17](#_Toc26974089)

[1.3.1 Простой алгоритм закраски 17](#_Toc26974090)

[1.3.2 Закраска методом Гуро 18](#_Toc26974091)

[1.3.3 Закраска методом Фонга 19](#_Toc26974092)

[1.3.4 Вывод по выбору алгоритма закраски 20](#_Toc26974093)

[1.4 Алгоритмы наложения текстур 20](#_Toc26974094)

[1.4.1 Аффинное текстурирование 20](#_Toc26974095)

[1.4.2 Перспективно-корректное текстурирование 20](#_Toc26974096)

[1.4.3 Параболическое текстурирование 21](#_Toc26974097)

[1.4.4 Вывод по выбору алгоритма наложения текстур 21](#_Toc26974098)

[1.5 Выбор модели освещения 22](#_Toc26974099)

[1.5.1 Модель освещения Ламберта 22](#_Toc26974100)

[1.5.2 Модель освещения Фонга 23](#_Toc26974101)

[1.5.3 Модель освещения Уиттеда 24](#_Toc26974102)

[1.5.4 Вывод по выбору модели освещения 25](#_Toc26974103)

[1.6 Описание трёхмерных преобразований 26](#_Toc26974104)

[1.6.1 Способы хранения и обработки декартовых координат 26](#_Toc26974105)

[1.6.2 Матрицы аффинных преобразований декартовых координат 26](#_Toc26974106)

[1.6.3 Преобразование трехмерной сцены в пространство камеры 27](#_Toc26974107)

[1.6.4 Преобразование трёхмерной сцены в пространство области изображения 28](#_Toc26974108)

[2. Конструкторский раздел 29](#_Toc26974109)

[2.1 Алгоритм Брезенхема растеризации отрезка 29](#_Toc26974110)

[2.2 Алгоритм удаления невидимых граней с использованием z-буфера 30](#_Toc26974111)

[2.3 Алгоритм текстурирования 30](#_Toc26974112)

[2.3.1 Уточнение принадлежности точки полигону с помощью барицентрических координат 30](#_Toc26974113)

[2.3.2 Наложение текстур 32](#_Toc26974114)

[2.4 Алгоритмы закраски 33](#_Toc26974115)

[2.4.1 Простая закраска полигона 33](#_Toc26974116)

[2.5 Модель освещения 34](#_Toc26974117)

[2.6 Описание входных данных 35](#_Toc26974118)

[2.6.1 Файл 3-D модели 35](#_Toc26974119)

[2.6.2 Файл текстуры 35](#_Toc26974120)

[3. Технологический раздел 37](#_Toc26974121)

[3.1 Обоснование выбора языка, среды и платформы программирования 37](#_Toc26974122)

[3.2 Описание структуры программы 37](#_Toc26974123)

[3.2.1 Описание основных модулей программы 37](#_Toc26974124)

[3.2.1.1 Описание класса для работы с моделью 37](#_Toc26974125)

[3.2.1.2 Описание класса для работы с файлами .tga изображений 39](#_Toc26974126)

[3.2.1.3 Описание класса для работы с цветом 40](#_Toc26974127)

[3.2.1.4 Описание классов для работы с геометрией 40](#_Toc26974128)

[3.2.1.5 Описание классов шейдеров 41](#_Toc26974129)

[3.2.2 Описание интерфейса приложения 42](#_Toc26974130)

[3.2.2.1 Меню загрузки моделей 43](#_Toc26974131)

[3.2.2.2 Поле выбора режима отображения сцены 44](#_Toc26974132)

[3.2.2.3 Меню сохранения состояния сцены 44](#_Toc26974133)

[3.2.2.4 Меню выбора цвета каркаса/тонировки 45](#_Toc26974134)

[3.2.2.5 Меню настройки освещения 45](#_Toc26974135)

[3.2.2.6 Схема управления клавиатурой/мышью 46](#_Toc26974136)

[Заключение 47](#_Toc26974137)

[Список используемой литературы 48](#_Toc26974138)

# Введение

В наши дни в игровой и киноиндустрии повсеместно применяются визуальные эффекты. Специалисты стремятся максимально реалистично изобразить в своих продуктах построенные на компьютере изображения, поэтому сегодня всем тем, кто хочет понять принципы их создания, важно иметь инструменты для разработки собственных реалистичных трехмерных сцен, инструменты, которые позволили бы на первых этапах познакомиться с особенностями компьютерной графики.

Целью проекта является разработка программы для создания трехмерных графических сцен с использованием трехмерных готовых моделей и/или подготовленных для работы с данным программным обеспечением и их визуализация с учетом предоставленной текстуры или цвета, а также оптических эффектов, таких как наложение теней, правильное отображение присутствующего света.

Для достижения поставленной цели необходимо решить следующие задачи:

* провести анализ существующих алгоритмов удаления невидимых линий и поверхностей, закраски, текстурирования, а также моделей освещения и выбрать из них подходящие для выполнения проекта;
* произвести основные математические расчеты для реализации выбранных алгоритмов;
* выбрать подходящий язык программирования для реализации поставленной задачи;
* реализовать парсер .obj файлов для считывания моделей, .tga файлов для считывания текстур;
* реализовать интерфейс программного модуля.

# 1. Аналитический раздел

В данном разделе проводится анализ существующих алгоритмов, которые возможно использовать для построения трехмерных изображений, и выбираются наиболее подходящие алгоритмы для выполнения поставленных задач.

## 1.1 Алгоритмы растеризации отрезков

### 1.1.1 Алгоритм ЦДА

Алгоритм ЦДА (цифровой дифференциальный анализатор) растеризует отрезок прямой между двумя заданными точками, используя вычисления в числах с плавающей запятой или целых числах [1].

Пусть отрезок задан вещественными координатами концов {\displaystyle (x\_{1},y\_{1})}(x1,y1); (x2,y2)()() . Растровыми координатами концевых точек становятся округлённые значения исходных координат: {\displaystyle x\_{\mathrm {start} }=\operatorname {round} (x\_{1})}xstart = round(x1), {\displaystyle y\_{\mathrm {start} }=\operatorname {round} (y\_{1})} ystart = round(y1); {\displaystyle x\_{\mathrm {end} }=\operatorname {round} (x\_{2})} xend = round(x2), {\displaystyle y\_{\mathrm {start} }=\operatorname {round} (y\_{1})} yend = round(y2).

Большее по абсолютной величине число, {\displaystyle (x\_{\mathrm {end} }-x\_{\mathrm {start} })}(xend - xstart) или {\displaystyle (y\_{\mathrm {end} }-y\_{\mathrm {start} })}(yend - ystart), увеличенное на 1 принимается за количество шагов {\displaystyle L}L цикла растеризации.

В начале цикла вспомогательным вещественным переменным {\displaystyle x}x и {\displaystyle y}y присваиваются исходные координаты начала отрезка: {\displaystyle x=x\_{1}}x = x1; {\displaystyle y=y\_{1}}y = y1. На каждом шаге цикла эти вещественные переменные получают приращения {\displaystyle (x\_{\mathrm {end} }-x\_{\mathrm {start} })/L}( xend - xstart)/L; {\displaystyle (y\_{\mathrm {end} }-y\_{\mathrm {start} })/L}( yend - ystart)/L. Растровые же координаты, получаемые на каждом шаге, являются результатом округления соответствующих вещественных значений {\displaystyle x}x и {\displaystyle y}y.

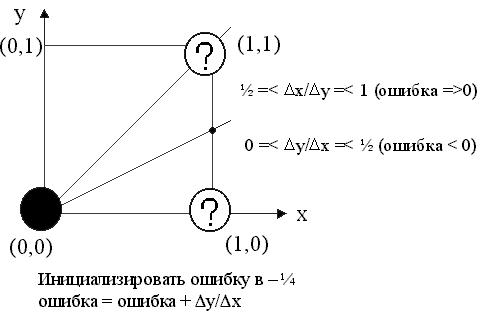
Применение вычислений с вещественными числами и лишь однократное использование округления для окончательного получения значения растровой координаты обусловливают высокую точность и низкое быстродействие алгоритма.

### 1.1.2 Алгоритм Брезенхема

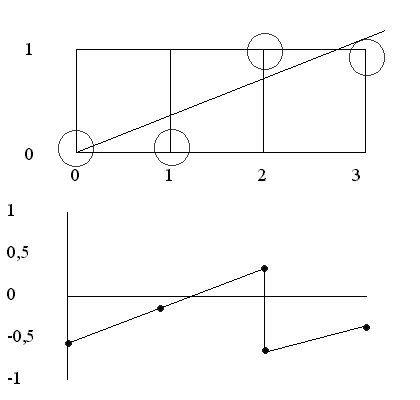
Алгоритм выбирает оптимальные растровые координаты для представления отрезка. В процессе работы одна из координат - либо x, либо y (в зависимости от углового коэффициента) - изменяется на единицу. Изменение другой координаты (на 0 или 1) зависит от расстояния между действительным положением отрезка и ближайшими координатами сетки. Такое расстояние назовем ошибкой.

Алгоритм построен так, что требуется проверить лишь знак этой ошибки. На Рисунке 3.1 это иллюстрируется для отрезка в первом октанте, т.е. для отрезка с угловым коэффициентом, лежащим в диапазоне от 0 до 1. Из рисунка можно заметить, что если угловой коэффициент отрезка из точки (0,0) больше, чем 1/2, то пересечение с прямой x = 1 будет расположено ближе к прямой y = 1, чем к прямой y = 0. Следовательно, точка растра (1,1) лучше аппроксимирует ход отрезка, чем точка (1,0). Если угловой коэффициент меньше 1/2, то верно обратное. Для углового коэффициента, равного 1/2, нет какого-либо предпочтительного выбора. В данном случае алгоритм выбирает точку (1,1).

Не все отрезки проходят через точки растра. Подобная ситуация иллюстрируется Рисунке 1.2, где отрезок с тангенсом угла наклона 3/8 сначала походит через точку растра (0,0) и последовательно пересекает три пиксела. Так же иллюстрируется вычисление ошибки при представлении отрезка дискретными пикселами.



*Рисунок 1.1. Основная идея работы алгоритма Брезенхема*



*Рисунок 1.2. График ошибки в алгоритме Брезенхема*

Так как желательно проверять только знак ошибки, то она первоначально устанавливается равной -1/2. Таким образом, если угловой коэффициент отрезка больше или равен 1/2, то величина ошибки в следующей точке растра с координатами (1,0) может быть вычислена как

(1.1)

где m - угловой коэффициент. В нашем случае при начальном значении ошибки -1/2

(1.2)

Так как е отрицательно, отрезок пройдет ниже середины пиксела. Следовательно, пиксел на том же самом горизонтальном уровне лучше аппроксимирует положение отрезка, поэтому у не увеличивается. Аналогично вычисляем ошибку

(1.3)

в следующей точке растра (2,0). Теперь е положительно, значит отрезок пройдет выше средней точки. Растровый элемент (2,1) со следующей по величине координатой у лучше аппроксимирует положение отрезка. Следовательно, у увеличивается на 1. Прежде чем рассматривать следующий пиксел, необходимо откорректировать ошибку вычитанием из нее 1. Имеем

(1.4)

Заметим, что пересечение вертикальной прямой x = 2 с заданным отрезком лежит на 1/4 ниже прямой у = 1. Если же перенести отрезок 1/2 вниз, мы получим как раз величину -3/4. Продолжение вычислений для следующего пикселя дает

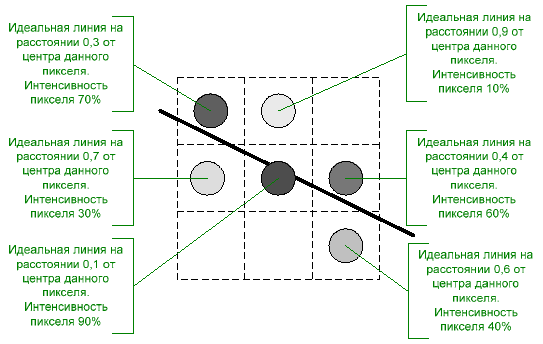
(1.5)

Так как е отрицательно, то у не увеличивается. Из всего сказанного следует, что ошибка — это интервал, отсекаемый по оси у рассматриваемым отрезком в каждом растровом элементе (относительно -1/2).

### 1.1.3 Алгоритм Ву со сглаживанием

Алгоритм был предложен Ву Сяолинем в 1991 году. Алгоритм сочетает высококачественное устранение ступенчатости и скорость работы, близкую к скорости алгоритма Брезенхема без сглаживания.

Горизонтальные и вертикальные линии не требуют никакого сглаживания, поэтому их рисование выполняется отдельно. Для остальных линий алгоритм Ву проходит их вдоль основной оси, подбирая координаты по неосновной оси аналогично алгоритму Брезенхема. Отличие состоит в том, что в алгоритме Ву на каждом шаге устанавливается не одна, а две точки. Например, если основной осью является *Х*, то рассматриваются точки с координатами (х, у) и (х, у+1). В зависимости от величины ошибки, которая показывает, как далеко ушли пиксели от идеальной линии по неосновной оси, распределяется интенсивность между этими двумя точками. Чем больше удалена точка от идеальной линии, тем меньше её интенсивность. Значения интенсивности двух пикселей всегда дают в сумме единицу, то есть это интенсивность одного пикселя, в точности попавшего на идеальную линию. Такое распределение придаст линии одинаковую интенсивность на всём её протяжении, создавая при этом иллюзию, что точки расположены вдоль линии не по две, а по одной. Пример такого распределения представлен на Рисунке 1.3.



*Рисунок 1.3. Распределение интенсивности пикселя в зависимости от расстояния до идеальной линии*

### 1.1.4 Вывод по выбор алгоритма растеризации отрезков

В рамках проекта алгоритм растеризации отрезка используется для построения каркасной модели. В данном режиме отображения сцены важно не качество отображения, а точность передачи положения модели в пространстве. Исходя из этого факта, приходит вывод о том, что алгоритмы со сглаживанием не нужны, они попросту будут использовать больше вычислительных ресурсов, поэтому было принято решение остановиться на простом алгоритме Брезенхема. Этот алгоритм удовлетворяем основным требованиям: он быстрый и максимально точно растеризует отрезок.

## 1.2 Алгоритмы удаления невидимых линий и поверхностей

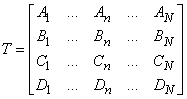
### 1.2.1 Алгоритм Робертса

Алгоритм Робертса - алгоритм получения изображений одиночных выпуклых объектов, составленных из плоских граней, работающий в объектном пространстве. Алгоритм Робертса может быть применен для изображения множества выпуклых многогранников на одной сцене в виде проволочной модели с удаленными невидимыми линиями [2]. Метод не пригоден непосредственно для передачи падающих теней и других сложных визуальных эффектов.

Многогранник, состоящий из N граней, описывается матрицей T размера 4×N, каждый n-й столбец которой содержит коэффициенты уравнения n-й плоскости:

в объектной системе координат XYZ:

.



Минимальное число граней N = 4 наблюдается у тетраэдра.

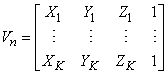
Важное требование к модели объекта в алгоритме Робертса заключается в достижении такой функции каждой плоскости:

при которой справедливо   для любой точки, заведомо принадлежащей телу многогранника. Достижение этого условия осуществляется путем опытной проверки знака функции относительно внутренней точки, в качестве которой может выступать точка со средним геометрическим положением относительно всех вершин многогранника. При достижении положительного знака функций всех граней автоматически достигается ориентация нормали  к любой из граней внутрь фигуры.

Кроме параметров уравнений граней необходимо знать координаты вершин каждой грани. Вершины могут быть заданы или вычислены из матрицы   путем определения общих решений каждой плоскости со всеми остальными.

Для каждой n-й грани должна быть составлена матрица K, содержащая координаты всех вершин, принадлежащих этой грани:

.



He все грани, входящие в состав объекта, будут видны наблюдателю. Невидимой будет та, для которой выполняется условие - , где    - нормаль к грани,   – вектор, совпадающий с направлением взгляда.

Эти грани исключаются из дальнейшего анализа. Все оставшиеся являются видимыми.

После этапа удаления не лицевых отрезков необходимо выяснить, существуют ли такие отрезки, которые экранируются другими телами в картинке или в сцене. Для этого каждый оставшийся отрезок или ребро нужно сравнить с другими телами сцены или картинки.

Возможны следующие случаи:

1. Грань ребра не закрывает. Ребро остается в списке ребер.
2. Грань полностью закрывает ребро. Ребро удаляется из списка рассматриваемых ребер.
3. Грань частично закрывает ребро. В этом случае ребро разбивается на несколько частей, видимыми из которых являются не более двух. Само ребро удаляется из списка рассматриваемых ребер, но в список проверяемых ребер добавляются те его части, которые данной гранью не закрываются.

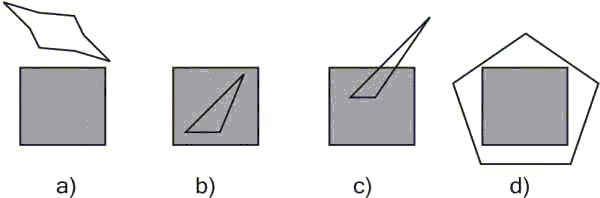
Достоинствами алгоритма являются скорость работы и простота реализации. Основным недостатком метода, определившим ограниченность его распространения, являются неспособность без привлечения других подходов реализовать падающие тени, невозможность передачи зеркальных эффектов и преломления света и, наконец, строгая ориентация метода только на выпуклые многогранники.

### 1.2.2 Алгоритм Варнока

В алгоритме Варнока и его вариантах делается попытка воспользоваться тем, что большие области изображения когерентны [6].

В пространстве изображения рассматривается окно и решается вопрос о том, пусто ли оно, или его содержимое достаточно просто для визуализации. Если это не так, то окно разбивается на фрагменты до тех пор, пока содержимое фрагмента не станет достаточно простым для визуализации или его размер не достигнет требуемого предела разрешения. В последнем случае информация, содержащаяся в окне, усредняется, и результат изображается с одинаковой интенсивностью или цветом. В оригинальной версии алгоритма каждое окно разбивалось на четыре одинаковых подокна. Многоугольник, входящий в изображаемую сцену, называют:

1. Внешним, если он целиком находится вне окна – Рисунок 1.4a;
2. Внутренним, если он целиком расположен внутри окна – Рисунок 1.4b;
3. Пересекающим, если он пересекает границу окна – Рисунок 1.4c;
4. Охватывающим, если окно целиком расположено внутри него – Рисунок 1.4d.



*Рисунок 1.4. Варианты расположения многоугольника по отношению к окну*

Для каждого окна:

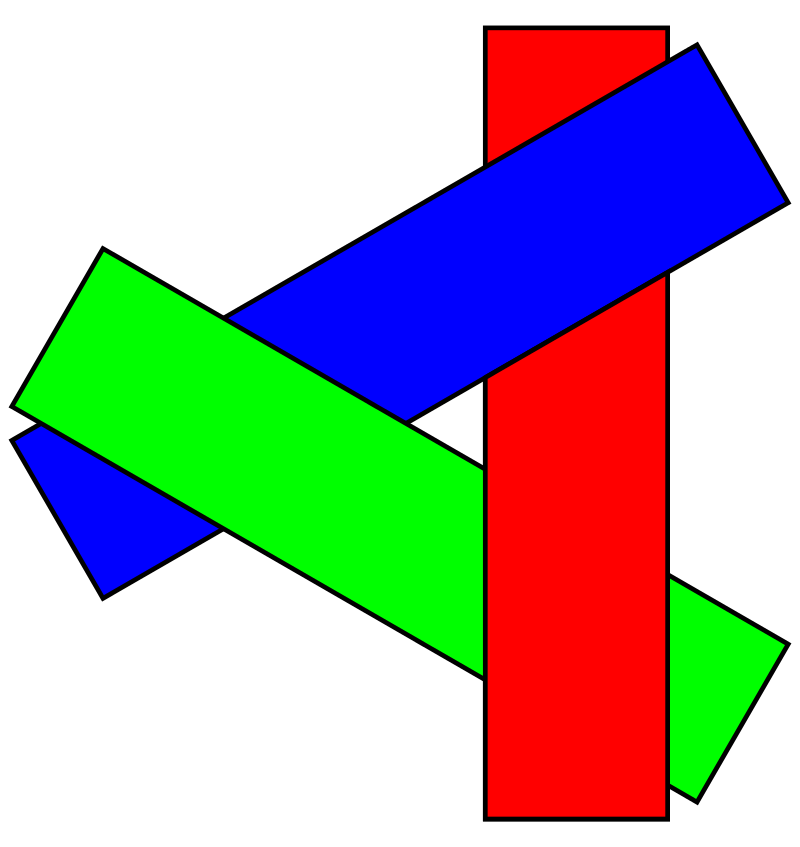
1. Если все многоугольники сцены являются внешними по отношению к окну, то оно пусто; в этом случае окно закрашивается фоновым цветом и дальнейшему разбиению не подлежит.
2. Если только один многоугольник сцены имеет общие точки с окном и является по отношению к нему внутренним, то окно заполняется фоновым цветом, а сам многоугольник заполняется своим цветом.
3. Если только один многоугольник сцены имеет общие точки с окном и является по отношению к нему пересекающим, то окно заполняется фоновым цветом, а часть многоугольника, принадлежащая окну, заполняется цветом многоугольника.
4. Если только один многоугольник охватывает окно и нет других многоугольников, имеющих общие точки с окном, то окно заполняется цветом этого многоугольника.
5. Если существует хотя бы один многоугольник, охватывающий окно, то среди всех таких многоугольников выбирается тот, который расположен ближе всех многоугольников к точке наблюдения, и окно заполняется цветом этого многоугольника.
6. В противном случае производится новое разбиение окна.

Существуют различные реализации алгоритма Варнока. Были предложены варианты оптимизации, использующие предварительную сортировку многоугольников по глубине, т. е. по расстоянию от точки наблюдения, и другие. Достоинством алгоритма Варнока является учет когерентности, из-за чего скорость его работы повышается при увеличении размеров однородных областей изображения. К недостаткам алгоритма можно отнести невозможность передачи зеркальных эффектов и преломления света, а также несовершенство способа разбиения изображения – в сложных сценах число разбиений может стать очень большим, что приведет к потере скорости.

### 1.2.3 Алгоритм художника

Алгоритм художника является простейшим программным решением проблемы «видимости». Данный алгоритм основывается на работе художника из настоящей жизни. Равно как настоящие художники наносят на картину сначала наиболее отдаленные детали сцены, алгоритм заполняет сцену по тому же принципу: все полигоны сортируются по удаленности от камеры, а затем выводятся по порядку от самых дальних до самых ближних.

Недостатком алгоритма является неправильный рендер изображения в неоднозначных ситуациях при перекрывающихся полигонах. Пример похожей ситуации приведен на Рисунке 1.5.

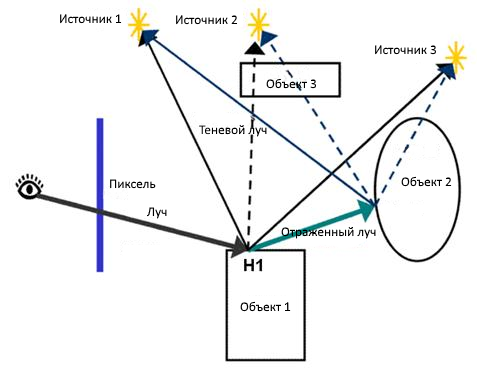


*Рисунок 1.5. Пример ситуации, в которой будет получен неверный рендер сцены, используя алгоритм художника*

У данного недостатка есть решение – например, разделение полигонов на части до перекрытия и после, но с увеличением количества полигонов и самого изображения это приведет к большим ресурсным издержкам в виде длительного времени и дополнительных затрат памяти на хранение информации о пересечениях полигонов.

### 1.2.4 Алгоритм обратной трассировки лучей

Алгоритм обратной трассировки лучей выглядит следующим образом: из камеры через каждый пиксел изображения испускается луч и находится точка его пересечения с поверхностью сцены. Лучи, выпущенные из камеры, называют первичными. Пусть, первичный луч пересекает некий объект 1 в точке H1, как показано на Рисунке 1.6.



*Рисунок 1.6. Алгоритм обратной трассировки лучей*

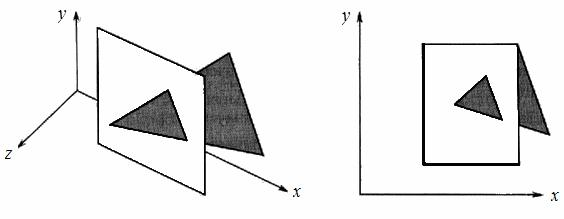
Далее необходимо определить для каждого источника освещения, видна ли из него эта точка. Тогда в направлении каждого точечного источника света испускается теневой луч из точки H1. Это позволяет определить, освещается ли данная точка конкретным источником. Если теневой луч находит пересечение с другими объектами, расположенными ближе к точке H1, чем источник света, значит, точка H1 находится в тени от этого источника и освещать ее не надо. Иначе считаем освещение по некоторой локальной модели. Освещение со всех видимых (из точки H1) источников света складывается. Далее, если материал объекта 1 имеет отражающие свойства, из точки H1 испускается отраженный луч и для него вся процедура трассировки рекурсивно повторяется. Аналогичные действия должны быть выполнены, если материал имеет преломляющие свойства.

Техника рендеринга с трассировкой лучей отличается высоким реализмом получаемого изображения. Она позволяет изобразить гладкие объекты без аппроксимации их полигональными поверхностями, а также воссоздать тени, отражения и преломления света. Алгоритм трассировки позволяет параллельно и независимо трассировать два и более лучей, разделять участки для трассирования на разных узлах кластера и т.д. Также присутствует отсечение невидимых поверхностей, перспектива.

Недостатком метода обратного трассирования является производительность. Трассировка лучей каждый раз начинает процесс определения цвета пикселя заново, рассматривая каждый луч наблюдения в отдельности [8].

### 1.2.5 Алгоритм удаления невидимых граней с использованием Z-буфера

Это алгоритм, работающий в пространстве изображения, как показано на Рисунке 1.7. Буфер кадра используется для запоминания интенсивности каждого пиксела в пространстве изображения, z-буфер — это отдельный буфер глубины, используемый для запоминания координаты z или глубины каждого видимого пиксела в пространстве изображения.



*Рисунок 1.7. Пример алгоритма с использованием Z-буфера*

Уравнение плоскости имеет вид:

(1.6) (1.7)

При плоскость многоугольника параллельна оси Z.

У сканирующей строки и глубина пиксела на этой строке, у которого , равна:

(1.8)

Так как , то

В процессе работы значение z каждого пиксела, который нужно занести в буфер кадра, сравнивается с глубиной уже занесенного в z-буфер пиксела. Если новый пиксел расположен впереди пиксела, находящегося в буфере кадра, то новый пиксел заносится в этот буфер и производится корректировка z-буфера новым значением z.

К достоинствам алгоритма, использующего z-буфер, можно отнести простоту реализации, а также отсутствие предварительной сортировки элементов сцены. Недостатками алгоритма являются трудоёмкость реализации эффектов прозрачности, а также перерасход по памяти - алгоритм предполагает хранение двух двумерных массивов, размер которых увеличивается с увеличением размеров изображения.

### 1.2.6 Вывод по выбору алгоритма удаления невидимых линий и поверхностей

В рамках выполнения проекта будет использоваться алгоритм с использованием z-буфера. Это обусловлено некоторыми факторами:

* Большинство используемых моделей предполагают быть высоко полигональными (от 2500 до 100000 полигонов)
* В рамках проекта рабочая область будет ограничена областью 600\*680 пикселей
* Алгоритм не требует сортировки или особого упорядочивания объектов сцены перед отрисовкой

## 1.3 Алгоритмы закраски

### 1.3.1 Простой алгоритм закраски

В данном методе закраски цвет всей поверхности рассчитывается согласно закону Ламберта [10]. Он формулируется так: плоская поверхность, имеющая одинаковую яркость по всем направлениям, отражает свет, интенсивность которого изменяется по закону косинуса:

(1.9)

где I0 – интенсивность, отражается в направлении нормали к поверхности,

θ – угол между направлением на наблюдателя и нормалью к поверхности.

Метод гранения позволяет получать изображения, сравнимые по качеству с реальными объектами, лишь при выполнении следующих условий:

1. источник света находится на большом расстоянии от объекта;
2. наблюдатель находится на большом расстоянии от объекта;
3. каждая грань тела является гранью многогранника, а не аппроксимирующей поверхностью;
4. поверхность аппроксимирована большим числом небольших плоских граней.

Преимуществами данного метода закраски является простая реализация, а также небольшие требования к ресурсам. В то же время данный метод имеет ряд существенных недостатков. Так, например, он плохо подходит для гладких объектов и плохо учитывает отраженный свет, ярко выражает края фигур.

### 1.3.2 Закраска методом Гуро

Метод закраски Гуро основан на интерполяции интенсивности. Он позволяет устранить дискретность изменения интенсивности и создать иллюзию гладкой криволинейной поверхности.

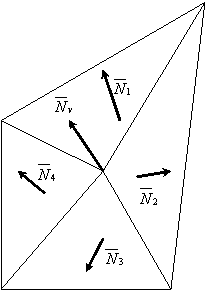
Процесс закраски по методу Гуро осуществляется в четыре этапа:

1) Вычисляются нормали ко всем полигонам.

2) Определяются нормали в вершинах путем усреднения нормалей по всем полигональным граням, которым принадлежит рассматриваемая вершина, как показано на Рисунке 1.8.

3) Используя нормали в вершинах и, применяя определенную модель освещения, вычисляют значения интенсивностей в вершинах многоугольника.

4) Каждый многоугольник закрашивается путем линейной интерполяции значений интенсивностей в вершинах сначала вдоль каждого ребра, а затем и между ребрами вдоль каждой сканирующей строки



*Рисунок 1.8. Определение нормалей*

Метод Гуро применим только для небольших граней, расположенных на значительном расстоянии от источника света. Если же размер грани достаточно велик, то расстояние от источника света до ее центра будет значительно меньше, чем до ее вершин, и, согласно закону освещенности, центр грани должен быть освещен сильнее ребер. Однако модель изменения освещенности, принятая в методе Гуро, предполагает линейное изменение яркости в пределах грани и не позволяет сделать середину грани ярче, чем ее края. В итоге на изображении появляются участки с неестественной освещенностью.

### 1.3.3 Закраска методом Фонга

Метод закраски Фонга основан на интерполяции вектора нормали, который затем используется в модели освещения для вычисления интенсивности пиксела.

Процесс закраски по методу Фонга осуществляется в четыре этапа:

1. Определяются нормали к граням.
2. По нормалям к граням определяются нормали в вершинах.
3. В каждой точке закрашиваемой грани определяется интерполированный вектор нормали.
4. По направлению векторов нормали определяется цвет точек грани.

Закраска Фонга требует больших вычислительных затрат, однако при этом достигается лучшая локальная аппроксимация кривизны поверхности, получается более реалистичное изображение, правдоподобнее выглядят зеркальные блики [11].

### 1.3.4 Вывод по выбору алгоритма закраски

Для закраски тонировки модели был выбран простой алгоритм закраски. Это обусловлено несколькими факторами, такими как: он прост в реализации, не требователен к ресурсам, позволяет достигнуть правдоподобного результата при использовании высоко полигональных моделей. Учитывая, что этот режим отображения не является основным, а служит лишь для предварительного просмотра загруженной модели, этот алгоритм подходит идеально.

## 1.4 Алгоритмы наложения текстур

### 1.4.1 Аффинное текстурирование

Этот метод текстурирования основан на приближении u, v линейными функциями, где u, v – координаты текстуры [9]. Пусть u - линейная функция, u = k1\*sx+k2\*sy+k3, где sx, sy - координаты принадлежащие проекции текстурируемого треугольника. Можно посчитать k1, k2, k3 исходя из того, что в вершинах грани u, v известны — это даст три уравнения, из которых находятся эти коэффициенты. Однако в таком случае вычисление цвета пикселя получается медленным. Оптимизацией подхода является расчет начальных значений координат текстур для каждого полигона, а затем билинейная интерполяция значений x и y текстуры. Основным недостатком данного метода является игнорирование координаты z, вследствие чего данные искажаются, и текстура накладывается нереалистично, что продемонстрировано на Рисунке 1.9.



*Рисунок 1.9. Пример неверного накладывания текстур методом аффинного текстурирования*

### 1.4.2 Перспективно-корректное текстурирование

Этот метод основан на приближении u, v кусочно-линейными функциями. При отрисовке каждая сканирующая строка разбивается на части, в начале и конце каждого куска считаются точные значения u, v, а в каждой части они интерполируется линейно.

Точные значения u и v можно считать по формулам точного текстурирования, но обычно используют более простой путь. Он основан на том факте, что значении 1/Z, u/Z и v/Z зависят от sx, sy линейно. Таким образом, достаточно для каждой вершины посчитать 1/Z, u/Z, v/Z и линейно их интерполировать - точно так же, как интерполируются u и v в аффинном текстурировании. Причем, поскольку эти значения зависят от sx, sy строго линейно, то интерполяция дает не приближенные результаты, а точные.

Сами же точные значения u, v считаются, как

u = (u/Z) / (1/Z),  
v = (v/Z) / (1/Z).

Пример результата работы такого метода изображен на Рисунке 1.10.

­

*Рисунок 1.10. Пример накладывания текстур методом перспективно-корректного текстурирования*

### 1.4.3 Параболическое текстурирование

Метод параболического текстурирования основан на приближении u, v квадратичными функциями - параболами. Для каждой сканирующей строки строятся приближающие u, v квадратичные функции. С помощью этих функций координаты текстур интерполируются по сканирующей строке. Для этого необходимы точные значения u, v в трех точках - начале, середине и конце строки. Они считаются точно так же, как в предыдущем методе.

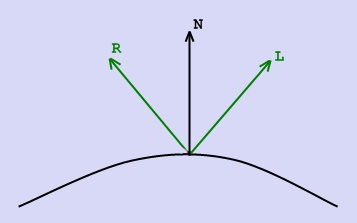
### 1.4.4 Вывод по выбору алгоритма наложения текстур

Наиболее подходящим методом текстурирования является перспективно-корректное текстурирование. Оно позволяет наложить текстуры максимально реалистично, получить результат, более приближенный к правдоподобному, а также позволяет избежать ошибки с неверным наложением текстур, связанных с перспективным проецированием, если мы сравниваем этот метод с методом аффинного текстурирования.

## 1.5 Выбор модели освещения

### 1.5.1 Модель освещения Ламберта

Модель освещения Ламберта является одной из самых простых моделей. Данная модель освещения моделирует идеальное диффузное освещение. Считается, что свет при попадании на поверхность рассеивается равномерно во все стороны. При расчёте такого освещения учитывается только ориентация поверхности (нормаль N) и направление на источник света (вектор L). Схема представлена на Рисунке 1.11. Рассеянная составляющая рассчитывается по закону Ламберта: интенсивность отражения пропорциональна косинусу угла между внешней нормалью к поверхности и направлением к источнику света.



*Рисунок 1.11. Отражение света в модели Ламберта*

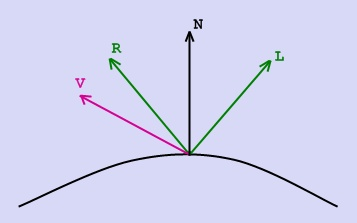
Если векторы оказываются единичными (а для удобства их принято считать единичными), косинус угла между ними оказывается равным их скалярному произведению.

(1.10)

где  
*Id* – рассеянная составляющая освещенности в точке;  
*kd* – свойство материала воспринимать рассеянное освещение;  
*id* – мощность рассеянного освещения;  
 – направление из точки на источник;  
 – вектор нормали в точке.

### 1.5.2 Модель освещения Фонга

Модель Фонга – модель освещения, представляющая собой комбинацию диффузной составляющей (модели Ламберта) и зеркальной составляющей, и работает таким образом, что кроме равномерного освещения на материале может еще появляться блик [12]. Падающий и отраженный лучи лежат в одной плоскости с нормалью к отражающей поверхности в точке падения, и эта нормаль делит угол между лучами на две равные части, как показано на Рисунке 1.6, где – направление на наблюдателя, – направление на источник, – отраженный луч. Отраженная составляющая освещенности в точке зависит от того, насколько близки направления вектора, направленного на наблюдателя, и отраженного луча.



*Рисунок 1.12. Модель Фонга*

В модели учитываются интенсивности фоновой, рассеянной компонент освещения, а также глянцевые блики.

, (1.11)

где

 — вектор нормали к поверхности в точке;

— направление проецирования (направление на источник света);

 — направление на наблюдателя;

*ka*— коэффициент фонового освещения;

*ks —* коэффициент зеркального освещения;

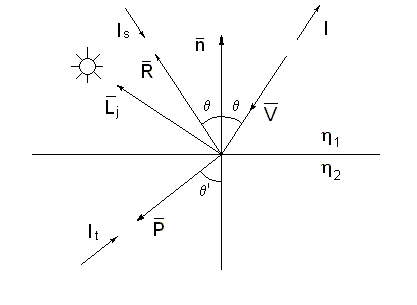
*kd* — коэффициент диффузного освещения;

*p* — степень блеска.

Модель Фонга учитывает только свойства заданной точки и источников освещения, игнорируя эффекты рассеивания, линзирования, отражения от соседних тел.

### 1.5.3 Модель освещения Уиттеда

Модель освещения предназначена для того, чтобы рассчитать интенсивность отраженного к наблюдателю света в каждом пикселе изображения. Она может быть локальной или глобальной [12]. В первом случае во внимание принимается только свет, падающий от источника (источников), и ориентация поверхности. Во втором учитывается также свет, отраженный от других объектов сцены или пропущенный сквозь них, как показано на Рисунке 1.7, где – направление взгляда, – отраженный луч направления взгляда, – преломленный луч направления взгляда, L – направление на источник освещения, n1, n2 – показатели преломления сред.



*Рисунок 1.13. Зеркальное отражение и преломление*

*в глобальной модели освещения Уиттеда.*

Согласно модели Уиттеда наблюдаемая интенсивность определяется суммарной интенсивностью:

, (1.12)

где

- коэффициент рассеянного отражения;

- коэффициент диффузного отражения;

- коэффициент зеркальности;

- коэффициент отражения;

- коэффициент преломления;

- интенсивность фонового освещения;

*-* интенсивность, учитываемая для диффузного рассеивания;

- интенсивность, учитываемая для зеркальности;

- интенсивность излучения, приходящего по отраженному лучу;

- интенсивность излучения, приходящего по преломленному лучу;

*С* - цвет поверхности.

Модель Уиттеда учитывает эффекты преломления и отражения, зеркальности.

### 1.5.4 Вывод по выбору модели освещения

Используя z-буфер для отрисовки тонированной модели, нет смысла использовать сложную модель освещения. По этой причине для данной цели мой выбор остановился на локальной модели Ламберта.

Для получения правдоподобного результата при наложении текстур необходимо использовать модель освещения Уиттеда, однако, поскольку обозреватель моделей считает сцену одним объектом с одним источником освещения, причем модель находится в искусственном пространстве, мы можем отказаться от преломленных и отраженных лучей без существенной потери правдоподобности изображения. Тем более, использование z-буфера не предполагает использование сложных моделей освещения, как это было указано ранее.

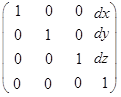
## 1.6 Описание трёхмерных преобразований

### 1.6.1 Способы хранения и обработки декартовых координат

Для удобства хранения и взаимодействия координат предусмотрено использовать вектор столбец размерности 3 или 4 вида [x y z=1 w=1] (z = 1 при условии, что мы работаем с 2-D координатами на плоскости). Это обусловлено тем, что матрицы основных преобразований являются матрицами размерностей 3 (на плоскости) и 4 (в пространстве).

### 1.6.2 Матрицы аффинных преобразований декартовых координат

1. сдвиг точки на dx, dy, dz по координатным осям:

 ;

1. масштабирование относительно начала координат с коэффициентами kx, ky, kz:

 ;

1. поворот относительно осей x, y, z на угол :
   * ось x:

 ;

* ось y:

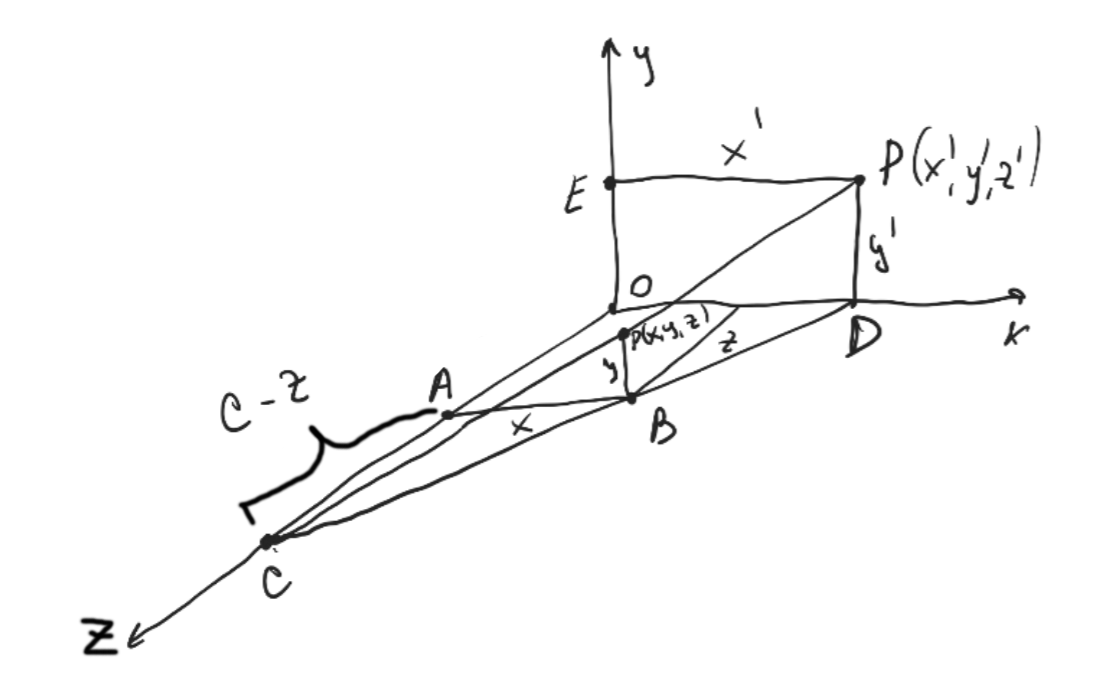
 ;

* ось z:

 .

### 1.6.3 Преобразование трехмерной сцены в пространство камеры

Допустим, необходимо спроецировать точку P(x y z) в плоскость z = 0. Также имеем в виду, что камера располагается на оси z на расстоянии c от центра координат. Рассмотрим ситуацию, представленную на Рисунке 1.14.



*Рисунок 1.14. Проецирование точку в плоскость камеры*

*в глобальной модели освещения Уиттеда.*

Мы знаем, что треугольники ABC и ODC подобны. То есть, мы можем записать:

.

(1.13)

Рассматривая треугольники CPB и CP'D, можно легко прийти к подобной записи и для координаты y:

(1.14)

Таким образом, получаем зависимость коэффициентов r = -1/c.

Для построения перспективы с камерой, находящейся на оси z, получаем:

̶˃

\* = (1.15)

̶˃

### 1.6.4 Преобразование трёхмерной сцены в пространство области изображения

Для того, чтобы преобразовать спроецированные координаты в координаты области изображения, достаточно умножить вектор координат на следующую матрицу:

(1.16)

где

W - ширина области изображения,

H - высота области изображения,

\_2W = W / 2,

\_2H = H / 2,

x – смещение относительно начал координат по оси x,

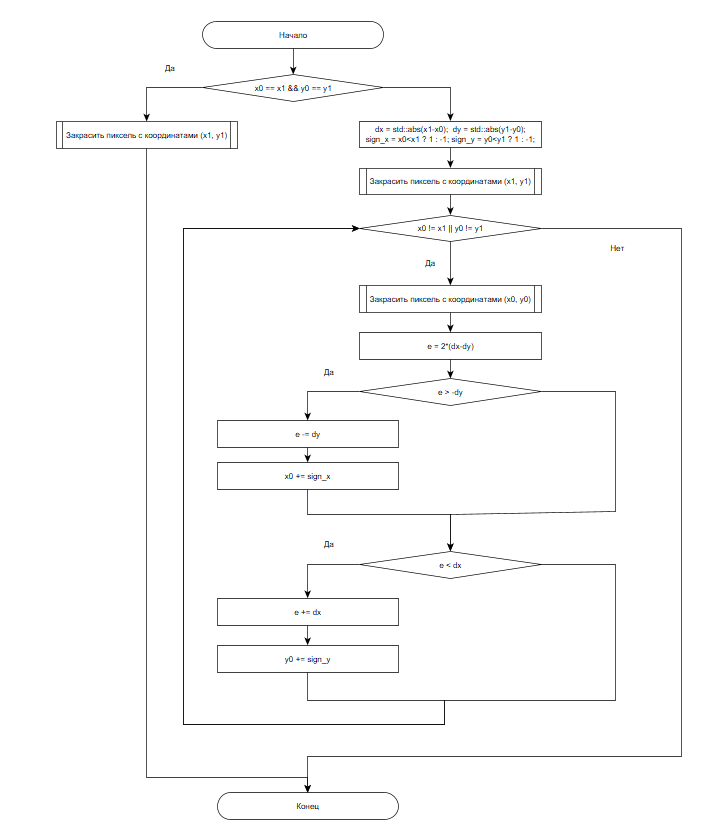
y – смещение относительно начал координат по оси y.

# 2. Конструкторский раздел

В данном разделе приводятся схемы реализованных алгоритмов.

## 2.1 Алгоритм Брезенхема растеризации отрезка

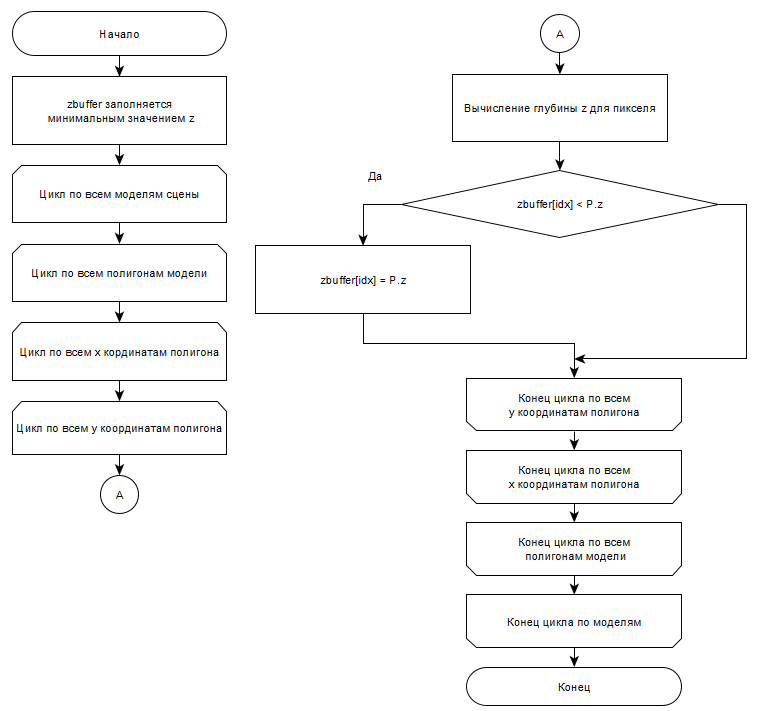
На Рисунке 2.1 приведена схема алгоритма Брезенхема.



*Рисунок 2.1. Алгоритм Брезенхема*

## 2.2 Алгоритм удаления невидимых граней с использованием z-буфера

На Рисунке 2.2 приведена схема алгоритма z-буфера.

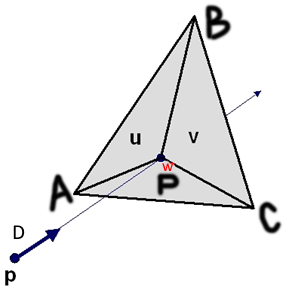


*Рисунок 2.2. Алгоритм z-буфера*

## 2.3 Алгоритм текстурирования

### 2.3.1 Уточнение принадлежности точки полигону с помощью барицентрических координат

Наложение текстур производится путем проведения барицентрического теста. Этот тест помогает проверить, необходимо ли текстурирование данного пикселя или нет. Примерная схема такого теста приведена на Рисунке 2.3.



*Рисунок 2.3. Барицентрический тест*

Дан треугольник ABC, точка P. Задача - найти барицентрические координаты точки P относительно треугольника ABC. Это тройка чисел (1 – u – v, u, v), с помощью которых мы можем найти точку P:

*(2.1)*

Это означает, что если мы поместим веса (1 – u – v, u, v) в соответствующие вершины треугольника, то центр масс системы окажется в точке P. Ровно это же самое можно переписать, сказав, что точка P будет иметь координаты (u, v) в репере (A, AB, AC):

*(2.2)*

Итак, даны векторы AB, AC, AP, нам нужно найти два вещественных u,v, которые отвечают следующему уравнению:

*(2.3)*

Данное уравнение эквивалентно системе:

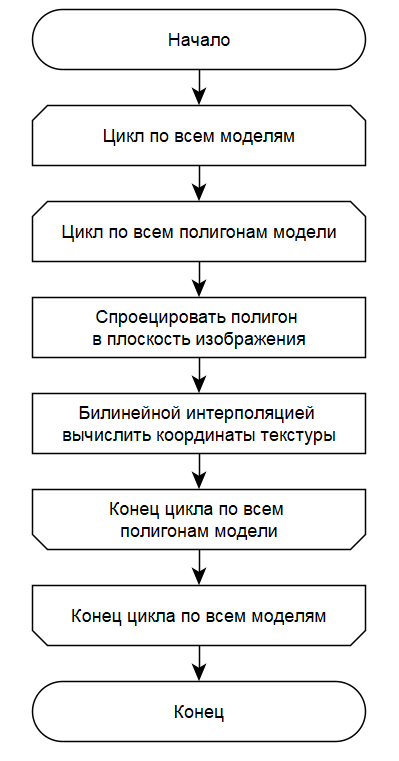
*(2.4)*

*(2.5)*

Для нахождения вектора векторно перемножим и и результат поделим на полученную 3-ю компоненту. В случае, если хотя бы одна из координат оказывается отрицательной, точка оказывается вне рассматриваемого треугольника, и нет надобности в ее отрисовке.

### 2.3.2 Наложение текстур

На вход программе поступает модель, для каждой вершины которой уже указаны соответствующие ей координаты текстур, поэтому остается только найти значение координат в промежутке между этими точками. На Рисунке 2.4 представлена схема используемого алгоритма.

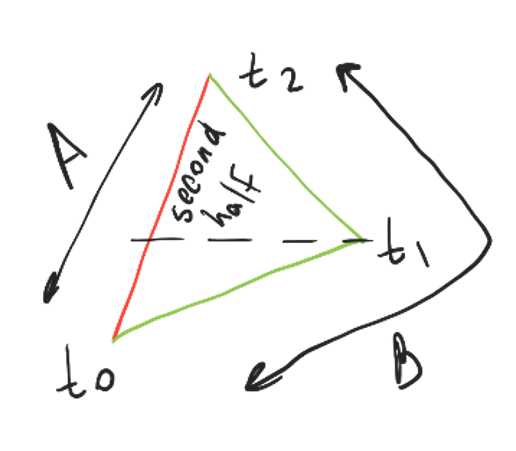


*Рисунок 2.4. Алгоритм текстурирования*

## 2.4 Алгоритмы закраски

### 2.4.1 Простая закраска полигона

В рамках проекта все модели имеют триангулированные полигоны, поэтому рассмотрим задачу закраски полигона как задачу закраски треугольника. Рассмотрим ситуацию, проиллюстрированную на Рисунке 2.5.



*Рисунок 2.5. Закраска треугольника*

На закраску подаётся треугольник с вершинами t0, t1, t2. t0, t1, t2 отсортированы по координате y. Закраску будем производить по частям от левой (красной) границы до правой (зеленой). Для этого воспользуемся билинейной интерполяцией для нахождения координат точки границы для сканирующей строки и координат точек между границами [2].

Тогда для границы A координаты будут определяться как:

, (2.6)

где A0 = t0,

A1 - A0 = t2.y – t0.y,

A - A0 = i (номер сканирующей строки),

A является структурой данных, хранящей координаты точки, с определёнными методами для сложения, вычитания и умножения.

Для границы B координаты определяются аналогично, за одной малой разницей: необходимо учитывать, какая часть закрашивается. В общем случае координаты B определяются как:

(2.7)

Если производится закраска от t0.y до t1.y, тогда:

B0 = t0,

B1 - B0 = t1.y – t0.y,

B - B0 = i (номер сканирующей строки).

Если производится закраска от t1.y до t2.y, тогда:

B0 = t1,

B1 - B0 = t2.y – t1.y,

B - B0 = i – (t1.y – t0.y) (номер сканирующей строки, отсчет ведётся от t1.y).

B является структурой данных того же типа, что и A.

После нахождения границ закраски необходимо закрасить то, что находится между ними. Для этого еще раз применяется интерполяция.

Тогда координаты закрашиваемой точки будут определяться как:

, (2.8)

где P0 = A,

P1 - P0 = B - A,

P - P0 = j – A.x (номер сканирующего столбца),

P является структурой данных того же типа, что A и B.

## 2.5 Модель освещения

Применяется модель освещения Ламберта. Согласно ей (при условии единичных векторов):

*(2.9)*

где I – итоговая интенсивность,

n – нормаль к полигону,

l – направления источника света.

Получаем, что интенсивность освещённости равна скалярному произведению вектора света и нормали к данному полигону. Нормаль вычисляется как векторное произведение двух сторон полигона.

## 2.6 Описание входных данных

### 2.6.1 Файл 3-D модели

Модели, используемые в проекте, загружаются из 3-D файла с расширением .obj. Текстуры к ним поставляются в комплекте из трех Targa изображений с расширением .tga:

* <modelname>\_diffuse.tga – изображение с накладываемой текстурой;
* <modelname>\_nm\_tangent.tga – карта нормалей;
* <modelname>\_spec.tga – карта бликов.

Файл 3-D модели имеет следующую структуру:

* Блок вершин, каждая строка имеет вид “v x y z”, где x, y, z – координаты в пространстве;
* Блок текстурных координат, каждая строка имеет вид “vt u v”, где u, v – координаты в плоскости изображения, если рассматривать ширину файла с текстурой как ось u, высоту – как ось v;
* Блок нормалей, каждая строка имеет вид “vn x y z”, где x, y, z – соответствующие параметры вектора нормали;
* Блок полигонов, каждая строка имеет вид “f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3”, где v1, v2, v3 – индекс координат для соответствующей вершины треугольного полигона, vn1, vn2, vn3 – индекс текстурных координат для соответствующей вершины треугольного полигона, vn1, vn2, vn3 – индекс координат нормали для соответствующей вершины треугольного полигона.

### 2.6.2 Файл текстуры

Файл текстуры имеет следующую структуру:

* Заголовок файла:
  + Длина идентификатора (1 байт);
  + Тип карты цветов (1 байт);
  + Тип изображения (1 байт) (в рамках проекта используются изображения с типами 2 (изображение TrueColor без сжатия), 3 (изображение монохромное без сжатия), 10 (изображение TrueColor с сжатием), 11 (изображение монохромное с сжатием))
  + Описание карты цветов (5 байт):
    - Индекс первого элемента (2 байта);
    - Длина карты цветов (2 байта);
    - Размер элемента карты цветов (1 байт);
  + Описание изображения (10 байт):
    - Положение изображения по оси x (2 байта);
    - Положение изображения по оси y (2 байта);
    - Ширина изображения (2 байта);
    - Высота изображения (2 байта);
    - Разрядность цвета (1 байт);
    - Описатель изображения (1 байт);
* Само изображение;
* Зона разработчика;
* Зона расширения;
* Подвал.

# 3. Технологический раздел

В данном разделе будет дано пояснение по выбору конкретного способа реализации, выбору того или иного средства/инструментария.

## 3.1 Обоснование выбора языка, среды и платформы программирования

Для создания проекта был выбран язык программирования C++ стандарта 2017 года. Это отличный объектно-ориентированный язык, обладающий поддержкой фреймворков для написания графического пользовательского интерфейса, большим количеством стандартных библиотек для выполнения часто используемых задач. Также данный язык программирования позволяет «прозрачно» работать с памятью – программист всегда знает, на что тратится и как выделяется память.

В качестве среды разработки была выбрана Qt. Эта среда обладает удобной системой сборки, встроенный редактором форм для упрощенной разработки интерфейса, а также является кроссплатформенным фреймворком, что позволяет запустить приложение на разных системах.

## 3.2 Описание структуры программы

### 3.2.1 Описание основных модулей программы

Для разработки приложения был выбран объектно-ориентированный подход. Ниже будут приведены основные классы и их методы, разработанные для выполнения задачи.

#### 3.2.1.1 Описание класса для работы с моделью

Класс model. Разработан для загрузки и управления ресурсами модели. Включает следующие методы и данные:

* std::vector<Vec3f> verts\_- хранит вершины загруженной модели;
* std::vector<std::vector<Vec3i> > faces\_- хранит полигоны загруженной модели;
* std::vector<Vec3f> norms\_ - хранит нормали загруженной модели;
* std::vector<Vec2f> uv\_ - хранит текстурные координаты;
* TGAImage diffusemap\_ - хранит текстуру (в случае существования);
* TGAImage normalmap\_ - хранит карту нормалей (в случае существования);
* TGAImage specularmap\_ - хранит карту бликов (в случае существования);
* void load\_texture(std::string filename, const char \*suffix, TGAImage &img) – загружает текстуру;
* Model(const char \*filename) – конструктор, инициализирует объект;
* ~Model() – деструктор, уничтожает объект;
* bool flag\_textures – флаг, сообщающий о возможности отрисовки текстур;
* float ddx – коэффициент масштабирования по оси x;
* float ddy - коэффициент масштабирования по оси y;
* int nverts() – возвращает количество вершин;
* int nfaces()- возвращает количество полигонов;
* void set\_vert(int i, Vec3f value) – заменяет данные об i-ой вершине;
* Vec3f normal(int iface, int nthvert) – возвращает указанную нормаль;
* Vec3f normal(Vec2f uv) – перегрузка предыдущего метода;
* Vec3f vert(int i) – возвращает указанную вершину;
* Vec3f vert(int iface, int nthvert) – перегрузка предыдущего метода;
* Vec2f uv(int iface, int nthvert) – возвращает текстурные координаты указанной вершины указанного полигона;
* TGAColor diffuse(Vec2f uv) – возвращает цвет пикселя при наложенной текстуре;
* std::vector<int> face(int idx) – возвращает указанный полигон;
* void rotate\_left(double angle) – осуществляет поворот вокруг оси y по часовой стрелке
* void rotate\_right(double angle) – осуществляет поворот вокруг оси y против часовой стрелки
* void rotate\_up(double angle) – осуществляет поворот вокруг оси x вдоль положительного направления оси;
* void rotate\_down(double angle) – осуществляет поворот вокруг оси x вдоль отрицательного направления оси;
* void move\_left(float dx) – осуществляет параллельный перенос на dx вдоль отрицательного направления оси x;
* void move\_right(float dx) – осуществляет параллельный перенос на dx вдоль положительного направления оси x;
* void move\_up(float dy) – осуществляет параллельный перенос на dy вдоль положительного направления оси y;
* void move\_down(float dy) – осуществляет параллельный перенос на dy вдоль отрицательного направления оси y;
* TGAImage diffuse\_m() – возвращает текстуру;
* TGAImage normal\_m() – возвращает карту нормалей;
* TGAImage specular\_m()- возвращает карту бликов.

#### 3.2.1.2 Описание класса для работы с файлами .tga изображений

Класс TGAImage. Разработан для загрузки и управления данными файлов .tga. Включает следующие методы и данные:

* unsigned char\* data – хранит прочитанные данные;
* int width – хранит ширину изображения;
* int height – хранит высоту изображения;
* int bytespp – хранит разрядность цвета;
* bool load\_rle\_data(std::ifstream &in) – читает данные из изображения с использованием сжатия;
* bool unload\_rle\_data(std::ofstream &out) - выгружает данные в изображение с использованием сжатия;
* enum Format – хранит тип изображения;
* TGAImage() – конструктор по умолчанию;
* TGAImage(int w, int h, int bpp) – инициализирует объект с указанной шириной/высотой и разрядностью цвета;
* TGAImage(const TGAImage &img) – конструктор копирования;
* ~TGAImage() – деструктор;
* bool read\_tga\_file(const char \*filename) – читает файл с указанным именем;
* bool write\_tga\_file(const char \*filename, bool rle=true) – записывает в файл с указанным именем;
* bool flip\_horizontally() – отражает изображение по горизонтали;
* bool flip\_vertically() – отражает изображение по вертикали;
* bool scale(int w, int h) – масштабирует в w раз по ширине, h раз по высоте;
* TGAColor get(int x, int y) – возвращает цвет указанного пикселя;
* bool set(int x, int y, TGAColor &c) – устанавливает в данный цвет указанный пиксель;
* bool set(int x, int y, const TGAColor &c) – устанавливает в данный цвет указанный пиксель;
* TGAImage & operator =(const TGAImage &img) – перегрузка оператора присваивания;
* int get\_width() – возвращает ширину изображения;
* int get\_height() – возвращает высоту изображения;
* int get\_bytespp() – возвращает разрядность цвета;
* unsigned char \*buffer() – возвращает массив с прочитанными данными;
* void clear() – очищает изображение.

#### 3.2.1.3 Описание класса для работы с цветом

Класс TGAColor. Разработан для управления цветом при работе с Targa изображениями. Включает следующие методы и данные:

* unsigned char bgra[4] – хранит текущие значения r, g, b, alpha компонент;
* unsigned char bytespp – хранит разрядность цвета пикселя;
* TGAColor() – конструктор по умолчанию;
* TGAColor(unsigned char R, unsigned char G, unsigned char B, unsigned char A=255) – создаёт экземпляр класса по переданным значениям компонент;
* TGAColor(const unsigned char \*p, unsigned char bpp) – создаёт экземпляр класса по переданному массиву значений компонент и значению разрядности цвета пикселя;
* unsigned char& operator[](const int i) – перегрузка оператора доступа по индексу;
* TGAColor operator \*(float intensity) const – перегрузка оператора умножения (используется для быстрого применения уровня интенсивности).

#### 3.2.1.4 Описание классов для работы с геометрией

Класс vec, а также его частные реализации для векторов длины 2 и 3. Разработан для упрощения вычислений, связанных с векторами. Включает следующие методы и данные:

* vec() – конструктор по умолчанию;
* T &operator[](const size\_t i) – перегрузка оператора доступа по индексу;
* const T &operator[](const size\_t i) const – константная версия предыдущего метода;
* T data\_[DIM] – хранит данные вектора;
* vec(T X, T Y) – конструктор вектора длины 2;
* vec<2, T>(const vec<2, U> &v) – инициализация экземпляра класса другим экземпляром того же класса вектора длины 2;
* vec<2, T> operator+ (const vec<2, T> &v) const – перегрузка оператора сложения векторов длины 2;
* T x, y – хранит данные вектора длины 2;
* vec(T X, T Y, T Z) - конструктор вектора длины 3;
* vec<3, T>(const vec<3, U> &v) - ) – инициализация экземпляра класса другим экземпляром того же класса вектора длины 3;
* float norm() – возвращает длину вектора;
* vec<3, T> &normalize(T l=1) – возвращает нормализованный вектор;
* vec<3, T> operator^ (const vec<3, T> &v) const – возвращает вектор, который является результатом векторного произведения данных векторов длины 3;
* vec<3, T> operator+ (const vec<3, T> &v) const – перегрузка оператора сложения векторов длины 3;
* T x, y, z – хранит данные вектора длины 3.

Класс mat. Разработан для упрощения вычислений, связанных с матрицами, а именно их умножений, транспонирований, нахождения детерминанта и прочего. Включает следующие методы и данные:

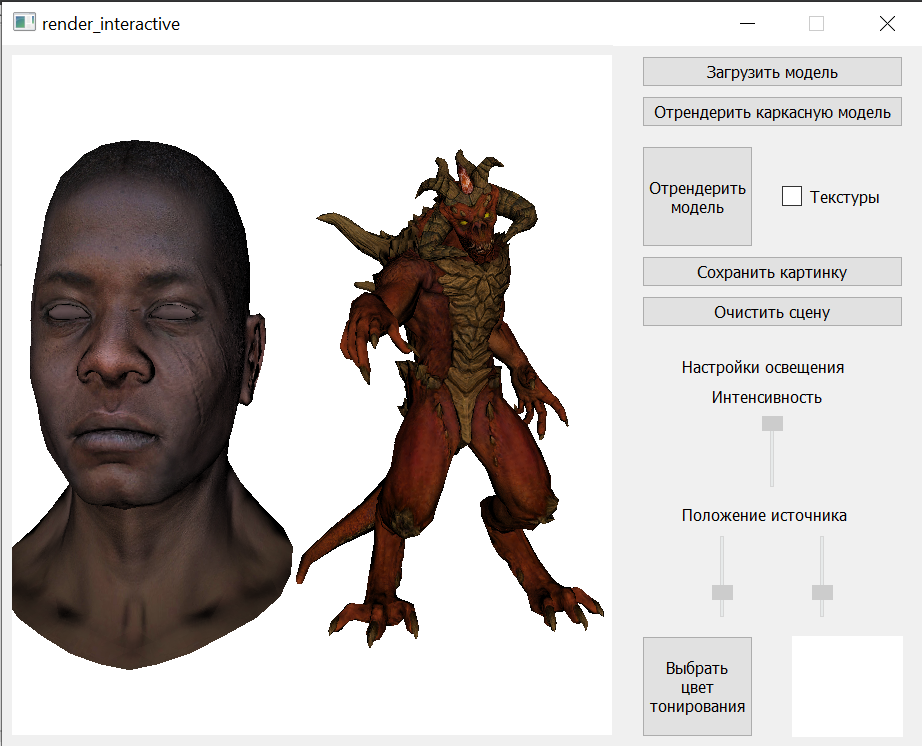
* vec<DimCols, T> rows[DimRows] - хранит данные матрицы;
* mat() - конструктор по умолчанию;
* vec<DimCols, T> &operator[](const size\_t idx) – перегрузка оператора доступа по индексу;
* const vec<DimCols, T> &operator[](const size\_t idx) const – константная версия предыдущего метода;
* vec<DimRows, T> col(const size\_t idx) const – константный метод для доступа к столбцу матрицы;
* void set\_col(size\_t idx, vec<DimRows, T> v) – изменяет указанный столбец указанными данными;
* static mat<DimRows, DimCols, T> identity() – создаёт единичную матрицу указанных размерностей;
* T det() const – константный метод нахождения детерминанта матрицы;
* mat<DimRows-1, DimCols-1, T> get\_minor(size\_t row, size\_t col) const – константный метод нахождения минора элемента матрицы;
* T cofactor(size\_t row, size\_t col) const – константный метод нахождения алгебраического дополнения для элемента;
* mat<DimRows,DimCols,T> invert() – находит обратную матрицу;
* mat<DimCols, DimRows, T> transpose() – транспонирует матрицу.

#### 3.2.1.5 Описание классов шейдеров

Класс IShader. Является абстрактным классом. Разработан для обозначения общего интерфейса разрабатываемых шейдеров. Содержит следующие методы и данные:

* Model \*model – хранит обрабатываемую модель;
* virtual ~*IShader*() – уничтожает экземпляр класса;
* virtual Vec4f *vertex*(int iface, int nthvert) = 0 – обозначает метод для определения вершинного шейдера;
* virtual bool *fragment*(Vec3f bar, TGAColor &color) = 0 – обозначает метод для определения фрагментного шейдера;
* void set\_model(Model \*m) – загружает в шейдер модель для обработки.
* Класс Shader. Разработан для наложения текстур на модель стандартным образом, не имеет никаких графических надстроек. Содержит следующие методы и данные:
* mat<2,3,float> varying\_uv – хранит текущую пару текстурных координат с учетом экранных преобразований$
* mat<4,3,float> varying\_tri – хранит координаты вершин текущего треугольника в системе координат отсечения;
* mat<3,3,float> varying\_nrm – хранит текущие параметры нормали с учетом экранных преобразований;
* mat<3,3,float> ndc\_tri – хранит нормализованные координаты текущей точки в системе координат устройства вывода;
* virtual Vec4f *vertex*(int iface, int nthvert) – выполняет преобразование координат вершин;
* virtual bool *fragment*(Vec3f bar, TGAColor &color) – выполняет вычисление цвета для каждого отрисовываемого пикселя.

### 3.2.2 Описание интерфейса приложения



*Рисунок 3.1. Общий интерфейс приложения*

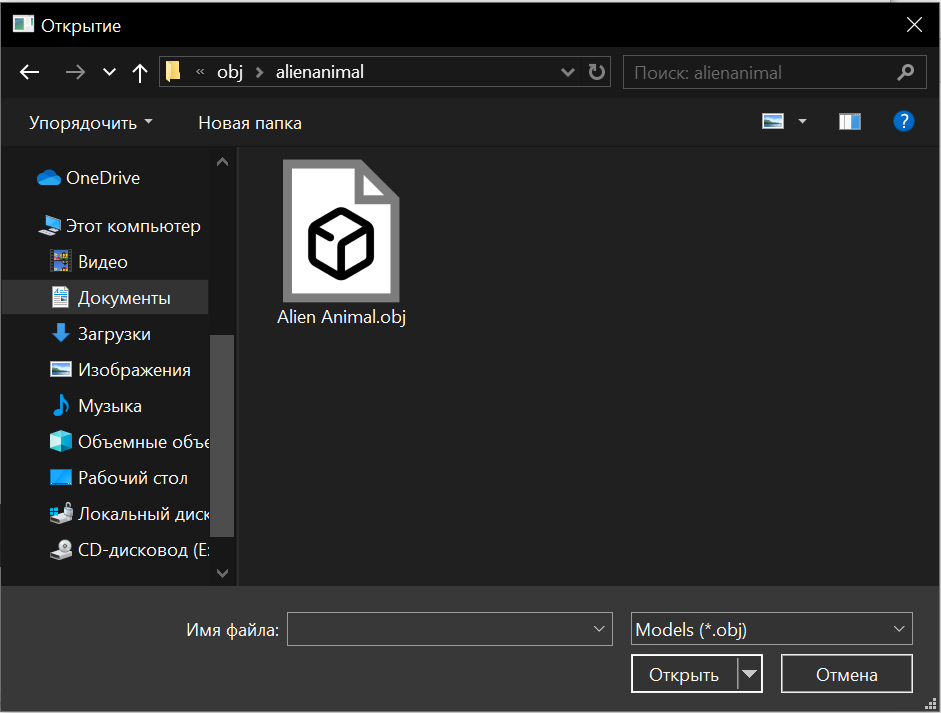
#### 3.2.2.1 Меню загрузки моделей

Для работы программы используются стандартные 3-D модели с расширением .obj. Важно, чтобы модели имели триангулированные полигоны. В случае, если предполагается, что модель должна иметь текстуру, в одну директорию с файлом самой модели необходимо положить файлы текстур, а именно: файл с самими текстурами с именем <modelname>\_diffuse.tga, файл с картой нормалей с именем <modelname>\_nm\_tangent.tga, файл с бликами и прочими деталями <modelname>\_spec.tga.

Само же меню представляет диалоговое окно, в котором нужно указать файл с 3-D моделью.



*Рисунок 3.2. Кнопка для перехода к диалоговому окну загрузки модели*



*Рисунок 3.3. Диалоговое окно выбора модели*

#### 3.2.2.2 Поле выбора режима отображения сцены

Приложение имеет несколько режимов, для переключения используется поле выбора «текстура» и отдельная кнопка для генерации сцены в каркасном режиме.



*Рисунок 3.4. Кнопка для генерации сцены в каркасном режиме*

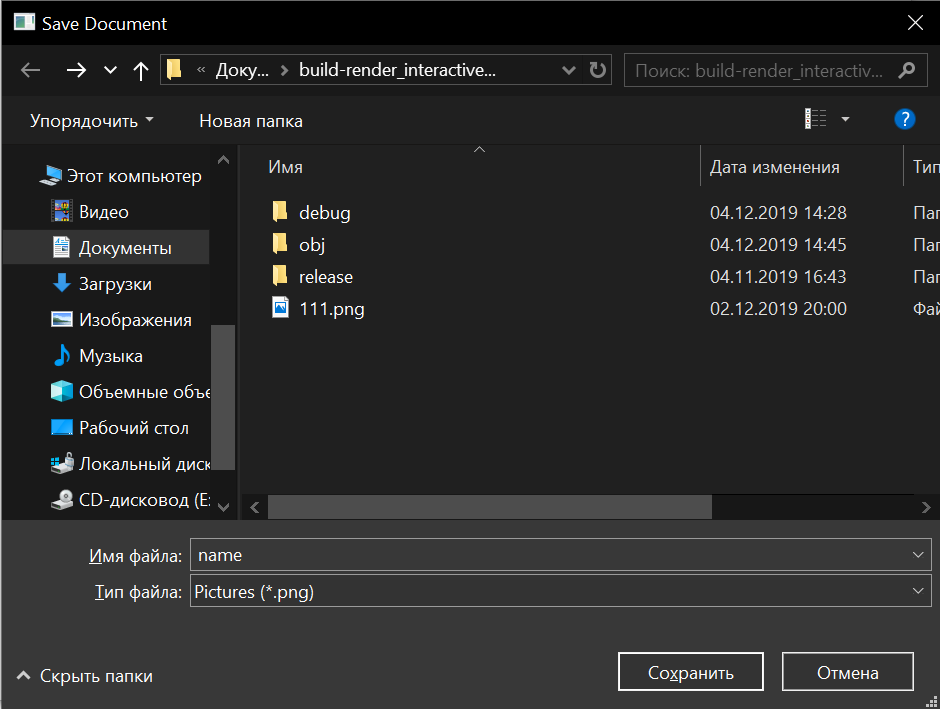


*Рисунок 3.5. Поле выбора для генерации сцены в режиме тонировки/наложения текстур*

#### 3.2.2.3 Меню сохранения состояния сцены

Текущее положение сцены можно сохранить в файл изображения с расширением .png.

Само же меню представляет диалоговое окно, в котором нужно указать имя файла с изображением, а также путь, по которому его нужно сохранить.

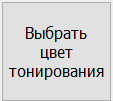


*Рисунок 3.6. Диалоговое окно сохранения состояния сцены*

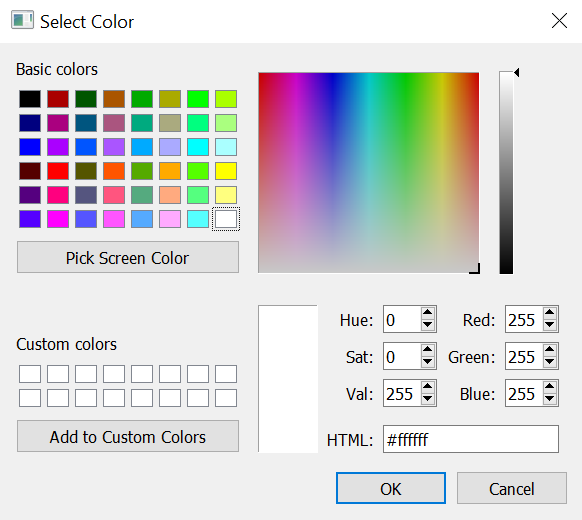
#### 3.2.2.4 Меню выбора цвета каркаса/тонировки

В программе предусмотрен выбор цвета для режима каркасной/ затонированной модели.

Само же меню представляет палитру, на которой нужно указать желаемый цвет.



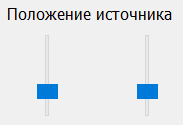
*Рисунок 3.7. Кнопка для перехода к диалоговому окну выбора цвета*



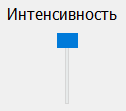
*Рисунок 3.8. Диалоговое окно выбора цвета*

#### 3.2.2.5 Меню настройки освещения

В программе предусмотрена возможность изменения положения источника света для режима генерации сцены с текстурами, а также возможность изменять интенсивность освещения для режима генерации сцены с простой тонировкой.



*Рисунок 3.9. Ползунки для управления положением источника света*



*Рисунок 3.10. Ползунок для управления интенсивностью источника света*

#### 3.2.2.6 Схема управления клавиатурой/мышью

Управление программой возможно как с помощью элементов, вынесенных справа от сцены (переключение режимов сцены, выбор сцены, изменение освещения), так и с помощью мыши клавиатуры (вращение сцены, приближение, отдаление и т.д.).

Управление мышью:

* Вращение вокруг горизонтальной оси сцены (движение мышью с зажатой левой клавишей вверх/вниз);
* Вращение вокруг вертикальной оси сцены (движение мышью с зажатой левой клавишей вправо/влево);

Управление клавиатурой:

* Нажатие клавиш W, A, S, D перемещает сцену в экранной плоскости вверх, влево, вниз, вправо соответственно;
* Комбинация клавиш Ctrl + ‘-‘, Ctrl + ‘+’ уменьшает, приближает сцену соответственно;

# Заключение

В ходе работы были проанализированы существующие алгоритмы удаления невидимых линий и поверхностей, закраски, текстурирования, а также модели освещения, указаны их преимущества и недостатки. Для решения поставленных задач были выбраны и реализованы алгоритмы с применением z-буфера, перспективного текстурирования, алгоритм простой закраски.

Реализованная в результате программа позволяет создавать трехмерные сцены из пользовательских моделей, загружаемых в виде .obj файлов, указывать интенсивность источника света, кроме того, имеется возможность перемещения, вращения, изменение размера объектов, наложения на них текстур в случае их существования.

В качестве перспектив дальнейшего развития данной программной системы можно предложить дальнейшую оптимизацию алгоритма текстурирования, например, переписав его с использованием мульти поточного программирования, а также генерацию сцены с возможностью использования геометрических примитивов, наложения любых пользовательских текстур из меню программы без необходимости манипуляций с файлами.

# Список используемой литературы

1. Куров А. В., Курс лекций по машинной графике – М., 2019
2. Градов В. М., Курс лекций по вычислительным алгоритмам – М., 2019
3. Иванов В.П., Батраков А.С. Трехмерная компьютерная графика / Под ред. Г.М. Полищука. — М.: Радио и связь, 1995. — 224 с.
4. Роджерс Д., Адамс Дж. «Математические основы компьютерной графики» - Москва, «Мир», 2001г.- 604 с.
5. Сяолинь Ву. Эффективная техника сглаживания // Computer graphics. — июль 1991. — Т. 25, № 4. — С. 143—152.
6. Алгоритм Варнока [Электронный ресурс]. – Режим доступа: https://www.intuit.ru/studies/courses/70/70/lecture/2102?page=2, свободный – (1.12.2019)
7. Алгоритм Вейлера-Азертона [Электронный ресурс]. – Режим доступа: http://compgraph.tpu.ru/weiler\_atherton.htm, свободный – (1.12.2019)
8. Обратная трассировка лучей [Электронный ресурс]. – Режим доступа: http://ray-tracing.ru/articles164.html, свободный – (1.12.2019)
9. Текстурирование [Электронный ресурс]. – Режим доступа: http://www.codenet.ru/progr/video/tex/, свободный – (1.12.2019)
10. Однотонная закраска (метод гранения) [Электронный ресурс]. – Режим доступа: https://studbooks.net/2248060/informatika/odnotonnaya\_zakraska\_metod\_graneniya, свободный – (1.12.2019)
11. Закраска Фонга [Электронный ресурс]. – Режим доступа: http://stratum.ac.ru/education/textbooks/kgrafic/additional/addit26.html, свободный – (1.12.2019)
12. Простые модели освещения [Электронный ресурс]. – Режим доступа: http://grafika.me/node/344, свободный – (1.12.2019)