



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 КУРСОВОЙ РАБОТЕ*

*НА ТЕМУ:*

*«Разработка программы построения 3Д сцен  
помещений различной планировки»*

Студент ИУ7-55Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата)

О.Н. Талышева  
(И.О.Фамилия)

Руководитель курсовой работы

\_\_\_\_\_  
(Подпись, дата)

Н.Н. Мартынюк  
(И.О.Фамилия)

2024 г

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)

(подпись)

(дата)

## З А Д А Н И Е на выполнение курсовой работы

по дисциплине      Компьютерная графика

Студент группы      ИУ7-45Б Талышева Олеся Николаевна

(Фамилия, имя, отчество)

Тема курсовой работы

Разработка программы построения 3Д сцен помещений различной планировки.

Направленность КР (учебная, исследовательская, практическая, производственная, др.)  
практическая

Источник тематики (кафедра,  
предприятие, НИР)      предприятие

**Задание**      Разработать программу построения 3Д сцен помещений различной  
планировки. Создать объекты (стена, окно, дверь), которые можно добавлять  
на сцену, удалять со сцены, перемещать, поворачивать и масштабировать на сцене.  
Предоставить возможность рассмотреть сцену из разных точек с помощью камеры.  
Обеспечить сохранение модели в файл и загрузку существующей для последующего  
редактирования.

### ***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-исследовательский раздел, заключение, список литературы,  
приложения.

2.2. Перечень графического материала (плакаты, схемы, чертежи и т. п.) На защиту  
проекта должна быть представлена презентация, состоящая из 15–20 слайдов.

Дата выдачи задания « \_\_\_\_ » \_\_\_\_\_ 20\_\_ г.

Руководитель курсовой работы

Мартынюк Н.Н.

(И.О. Фамилия)

Студент

Талышева О.Н.

(И.О. Фамилия)

**Примечание:** Задание оформляется в двух экземплярах: один выдается студенту, второй хранится  
на кафедре.

# Содержание

Введение	4
1 Аналитическая часть	6
1.1 Описание объектов сцены . . . . .	6
1.2 Выбор формы задания трёхмерных моделей . . . . .	6
1.3 Алгоритмы удаления невидимых линий . . . . .	7
1.3.1 Введение . . . . .	7
1.3.2 Алгоритм плавающего горизонта . . . . .	11
1.3.3 Алгоритм Робертса . . . . .	14
1.3.4 Алгоритм Варнака . . . . .	17
1.3.5 Алгоритм Вейлера-Азерттона . . . . .	22
1.3.6 Алгоритм художника . . . . .	25
1.3.7 Алгоритм трассировки лучей . . . . .	27
1.3.8 Алгоритм Z-буфера . . . . .	28
1.3.9 Методы построчного сканирования . . . . .	30
1.3.9.1 Алгоритм построчного Z-буфера . . . . .	31
1.3.9.2 Алгоритм S-буфера . . . . .	32
1.3.10 Выводы . . . . .	33
1.4 Алгоритмы отрисовки теней . . . . .	34
1.4.1 Введение . . . . .	34
1.4.2 Принципы работы Z-буфера . . . . .	35
1.4.3 Выводы . . . . .	35
1.5 Алгоритмы закраски . . . . .	36
1.5.1 Простая закраска (метод гранения) . . . . .	36
1.5.2 Закраска по Гуро . . . . .	37
1.5.3 Закраска по Фонгу . . . . .	37
1.5.4 Выводы . . . . .	38
1.6 Выводы . . . . .	39
2 Конструкторская часть	40
2.1 Требования к программному обеспечению . . . . .	40
2.1.1 Функциональные требования . . . . .	40
2.1.2 Нефункциональные требования . . . . .	41
2.1.3 Ожидаемые результаты . . . . .	41
2.2 Общий алгоритм решения поставленной задачи . . . . .	41
2.3 Алгоритм Z-буфера удаления невидимых линий . . . . .	42
2.4 Алгоритм Z-буфера с отрисовкой теней . . . . .	44
2.5 Представление объектов в программном обеспечении . . . . .	46
2.6 Выводы . . . . .	46

3	Технологическая часть	47
3.1	Выбор средств реализации ПО . . . . .	47
3.2	Структура программы . . . . .	48
3.2.1	Классы и их роли . . . . .	48
3.2.1.1	Пространственные элементы . . . . .	49
3.2.1.2	Работа с геометрическими преобразованиями . . . . .	49
3.2.1.3	Управление коллекцией объектов . . . . .	50
3.2.1.4	Управление сценой . . . . .	50
3.2.2	Логика работы программы . . . . .	50
3.3	Интерфейс программы . . . . .	51
3.4	Выводы . . . . .	53
4	Исследовательская часть	54
4.1	Примеры работы программы . . . . .	54
4.2	Сравнение времени работы программы . . . . .	56
4.3	Выводы . . . . .	58
	Заключение	59
	Список использованных источников	60

## ВВЕДЕНИЕ

В настоящее время введение технологий 3D моделирования активно развивается в самых разных сферах: архитектуре, дизайне, инженерии, безопасности и оперативном реагировании. Программное обеспечение для построения 3D сцен позволяет визуализировать объекты и пространства, что существенно облегчает их анализ, планирование и эксплуатацию. В особенности, технологии моделирования помещений с гибкой настройкой планировок и объектов находят важное применение для задач охраны и контроля доступа на стратегически важные объекты.

Одной из ключевых сфер, в которой данная тема является особенно актуальной, выступает обеспечение безопасности и оперативного реагирования силовых структур на проникновение на охраняемые объекты. В условиях возросших угроз безопасности и повышенных требований к защите территорий и объектов, необходимость быстрого создания точных моделей помещений с целью их анализа является приоритетной задачей. Использование 3D моделей позволяет оперативно прорабатывать сценарии вторжений, рассчитывать оптимальные пути пресечения проникновений и создавать план эвакуации или нейтрализации угрозы.

Для силовых структур важно иметь инструменты, которые позволяют в режиме реального времени моделировать различные планировки и адаптировать стратегические планы по обеспечению безопасности. Программа, позволяющая моделировать помещения с возможностью добавления объектов (стен, окон, дверей), их перемещения и изменения, управления камерой, может стать незаменимым инструментом для подготовки к операциям, тренировок и планирования реагирования в критических ситуациях.

Таким образом, разработка программы построения 3D сцен помещений имеет важное практическое значение, поскольку позволяет оперативно создавать модели для анализа и разработки решений по обеспечению безопасности.

Целью данной курсовой работы является разработка программного обеспечения для создания и редактирования 3D сцен помещений с возможностью интерактивного добавления объектов (стен, окон, дверей), их перемещения, изменения, поворота, а также обеспечения сохранения и загрузки моделей.

В рамках работы были поставлены следующие задачи:

- проанализировать требования к программе и исследовать существующие решения;
- изучить алгоритмы реализации технических решений и выбрать наиболее подходящие для работы с 3D сценами;
- разработать архитектуру и реализовать программу;
- протестировать программу целиком и её отдельные модули;

- исследовать производительность программы;
- подготовить отчётную документацию.

## 1. Аналитическая часть

В данном разделе будут представлены описание объектов сцены, рассмотрены алгоритмы удаления невидимых линий, отрисовки теней и закраски, а также проведено сравнение этих методов и обоснован выбор для реализации в разрабатываемом ПО.

### 1.1. Описание объектов сцены

Сцена состоит из различных объектов, включая стены, окна и двери. Каждый объект может быть представлен в виде параллелепипеда с заданными параметрами длины, ширины и высоты. Особое место среди объектов занимает пол – это плоскость, состоящая из заданного пользователем количества квадратов, размерами которых измеряются следующие объекты и относительно которых задаётся расположение стен, окон и дверей на экране.

1. Стена – это параллелепипед с заданными размерами, ограничивающий пространство. Стены могут быть различной толщины и высоты и по разному располагаться на сцене в зависимости от характеристик помещения. Эти объекты используются для формирования границ помещения.
2. Дверь – это стена с отверстием, которое расположено у нижней части объекта и размеры которого задаются пользователем.
3. Окно – это стена с отверстием, которое расположено выше уровня пола, а его размеры и расположение выбираются клиентом.
4. Источник света – это точка, расположенная далеко в бесконечности. Он излучает световые лучи во все стороны, что позволяет освещать сцену и создавать тени.
5. Камера – представляет собой объект, который может вращаться и рассматривать сцену с разных сторон и расстояний. Камера позволяет настраивать угол обзора, масштаб и положение относительно объектов сцены, что даёт возможность пользователю детально осмотреть помещение.

Эти объекты могут иметь различные параметры, что позволяет создавать уникальные конфигурации помещений, соответствующие потребностям пользователя и требованиям дизайна.

### 1.2. Выбор формы задания трёхмерных моделей

В процессе 3D моделирования создаются геометрические модели, т.е. модели, отражающие геометрические свойства изделий. Различают геометрические модели каркасные (проволочные), поверхностные, объемные (твёрдотельные).

1. Каркасная модель представляет форму детали в виде конечного множества линий, лежащих на поверхностях детали. Для каждой линии известны координаты концевых точек и указана их инцидентность ребрам или поверхностям.
2. Поверхностная модель отображает форму детали с помощью задания ограничивающих её поверхностей, например, в виде совокупности данных о гранях, ребрах и вершинах. Возможны различные виды задания поверхностей — плоскости, поверхности вращения, линейчатые поверхности.
3. Объёмная модель позволяет представить сложные изделия с обеспечением логической связанности информации, в частности, благодаря введению понятия о материале. В такой модели хранится информация, позволяющая отличать материал от пустоты, при этом пустота может рассматриваться как особый вид материала. [8]

В процессе 3D-моделирования выбор геометрической модели зависит от целей и характеристик задачи. Каркасные модели предоставляют лишь ограниченную информацию о форме объекта, что не всегда подходит для детализированного представления. Объёмные модели дают полное представление о внутреннем содержании, но они требуют значительно больше вычислительных ресурсов для обработки.

В рамках данной работы была выбрана поверхностная модель. Этот выбор обусловлен тем, что поверхностные модели позволяют эффективно представлять форму объекта с помощью ограничивающих его поверхностей, что идеально подходит для визуализации объектов на сцене. Поверхностная модель даёт возможность адекватно отобразить внешний вид объектов (стен, окон, дверей), при этом её использование не требует таких значительных вычислительных затрат, как объёмные модели. Такая модель также достаточно гибка для выполнения операций, связанных с рендерингом и проверкой взаимодействий объектов на сцене.

### 1.3. Алгоритмы удаления невидимых линий

#### 1.3.1 Введение

Одной из основных задач компьютерной графики является визуализация трёхмерных сцен. Подобные задачи возникают в системах автоматизированного проектирования, пакетах моделирования физических процессов, средствах компьютерной анимации и виртуальной реальности.

При отображении трёхмерной сцены на экране некоторые из объектов сцены могут заслонить другие объекты. Заслонённые части объектов невидимы и не должны рисоваться, или должны рисоваться иначе, чем видимые части, например, пунктиром. Если этого не делать, то изображение будет выглядеть неправильно.

Такие задачи решают с помощью алгоритмов удаления невидимых линий и поверхностей. Если сцена отображается в каркасном виде, линиями, то нужно удалять

невидимые линии. Каркасное изображение обычно строится из отрезков – рёбер, и алгоритм должен выделить части отрезков, заслонённых объектами сцены. Если объекты сцены отображаются в виде закрашенных поверхностей, то нужно удалять невидимые части этих поверхностей. Обычно в качестве поверхностей используются выпуклые многоугольники, чаще всего – треугольники. В курсовой работе необходимо реализовать один из алгоритмов удаления невидимых линий и поверхностей и произвести анализ его производительности. В соответствующих разделах приведены описания основных алгоритмов: Робертса, Варнака, Z-буфера, художника, трассировки лучей, построчного сканирования.

Все алгоритмы удаления невидимых линий и поверхностей можно разделить на две группы:

1. В одних, сначала определяется видимость для участков линий или поверхностей, а затем рисуются видимые части. К таким алгоритмам относится, например, алгоритм Робертса. В подобных алгоритмах необходимо сравнить каждый объект сцены (линию или поверхность) с остальными объектами, способными заслонить его.
2. В других, для каждого пикселя изображения определяется, какой из объектов сцены в нём виден. К таким алгоритмам относятся, например, алгоритмы трассировки лучей или Z-буфера. В этом случае нужно для каждого пикселя выбрать ближайший к наблюдателю объект сцены.

В обоих случаях требуется много вычислений, из-за того, что необходимо перебирать все объекты сцены. Для сокращения объёма вычислений используется свойство когерентности (англ. coherence – связность) расположенных рядом объектов. Можно выделить три вида когерентности:

1. Когерентность в картинной плоскости (плоскости экрана) – расположенные рядом пиксели, скорее всего, имеют одинаковые свойства, например, принадлежат одному и тому же объекту сцены, или видимы или, наоборот, заслонены одним и тем же объектом сцены.
2. Когерентность в объектном пространстве (пространстве сцены) – расположенные рядом объекты сцены, скорее всего, имеют одинаковые свойства, например, видимы или, наоборот, заслонены одним и тем же объектом сцены.
3. Когерентность во времени – при перемещении наблюдателя, на соседних кадрах будут видимы примерно одни и те же объекты сцены

В некоторых алгоритмах когерентность используется явно. Например, в алгоритме Варнака картинная плоскость рекурсивно делится на области расположенных рядом

пикселей, и, если вся область видима или, наоборот, заслонена, то дальнейшее её разделение не требуется. Другие алгоритмы можно модифицировать, используя свойство когерентности, существенно повышая их производительность. [3]

Анализ видимости объектов можно производить как в исходном трехмерном пространстве, так и на картинной плоскости. Это приводит к разделению методов на два класса:

1. методы, работающие непосредственно в пространстве самих объектов;
2. методы, работающие в пространстве картинной плоскости, т. е. работающие с проекциями объектов.

Получаемый результат представляет собой либо набор видимых областей или отрезков, заданных с машинной точностью (имеет непрерывный вид), либо информацию о ближайшем объекте для каждого пикселя экрана (имеет дискретный вид).

Методы первого класса дают точное решение задачи удаления невидимых линий и поверхностей, никак не привязанное к растровым свойствам картинной плоскости.

Они могут работать как с самими объектами, выделяя те их части, которые видны, так и с их проекциями на картинную плоскость, выделяя на ней области, соответствующие проекциям видимых частей объектов, и, как правило, практически не привязаны к растровой решетке и свободны от погрешностей дискретизации. Так как эти методы работают с непрерывными исходными данными и получающиеся результаты не зависят от растровых свойств, то их иногда называют непрерывными методами (continuous methods).

Простейший вариант непрерывного подхода заключается в сравнении каждого объекта со всеми остальными, что дает временные затраты, пропорциональные  $n^2$ , где  $n$  — количество объектов в сцене.

Однако следует иметь в виду, что непрерывные методы, как правило, достаточно сложны.

Методы второго класса (point-sampling methods) дают приближенное решение задачи видимости, определяя видимость только в некотором наборе точек картинной плоскости — в точках растровой решетки. Они очень сильно привязаны к растровым свойствам картинной плоскости и фактически заключаются в определении для каждого пикселя той грани, которая является ближайшей к нему вдоль направления проектирования. Изменение разрешения приводит к необходимости полного перерасчета всего изображения.

Простейший вариант дискретного метода имеет временные затраты порядка  $Cn$ , где  $C$  — общее количество пикселов экрана, а  $n$  — количество объектов.

Всем методам второго класса традиционно свойственны ошибки дискретизации (aliasing artifacts). Однако, как правило, дискретные методы отличаются известной пропорциональностью.

Кроме этого существует довольно большое количество смешанных методов, использующих работу как в объектном пространстве, так и в картинной плоскости, методы, выполняющие часть работы с непрерывными данными, а часть — с дискретными.

Большинство алгоритмов удаления невидимых граней и поверхностей тесно связано с различными методами сортировки. Некоторые алгоритмы проводят сортировку явно, в некоторых она присутствует в скрытом виде. Приближенные методы отличаются друг от друга фактически только порядком и способом проведения сортировки.

Очень распространенной структурой данных в задачах удаления невидимых линий и поверхностей являются различные типы деревьев — двоичные (BSP-trees), четверичные (Quadtree), восьмеричные (Octree) и др.

Методы, практически применяющиеся в настоящее время, в большинстве являются комбинациями ряда простейших алгоритмов, неся в себе целый ряд разного рода оптимизаций. [4]

Основные методы оптимизации:

1. Метод отсечения нелицевых граней (culling)

Позволяет примерно вдвое сократить количество рассматриваемых граней.

Для определения того, является заданная грань лицевой или нет достаточно взять произвольную точку этой грани и проверить выполнение условия  $(\mathbf{N}, \mathbf{L}) \leq 0$ , где  $\mathbf{N}$  — нормаль к грани,  $\mathbf{L}$  — направление проецирования.

2. Метод оболочек (Bounding Volumes, Bounding-Volume-Hierarchy (BVH))

Если оболочки не пересекаются, то и содержащиеся в них объекты тоже пересекаться не будут. Однако, если оболочки пересекаются, то сами объекты пересекаться не обязаны. В качестве ограничивающих тел чаще всего используются прямоугольные ограничивающие параллелепипеды. Оболочка описывается числами  $(X_{min}, Y_{min}, Z_{min})$  и  $(X_{max}, Y_{max}, Z_{max})$  из координат точек исходного объекта (4 для 2D).

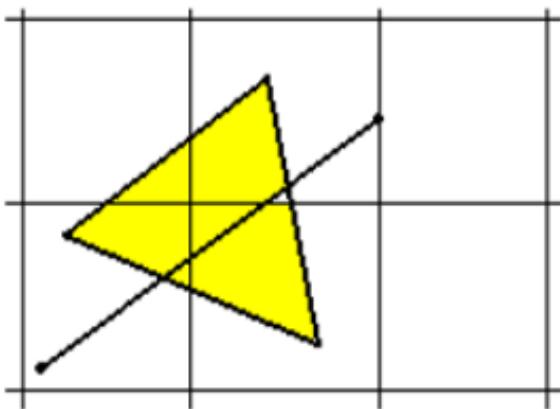
3. Разбиение пространства (картинной плоскости)

Еще один метод, облегчающий сравнение объектов, позволяющий использовать когерентность как в пространстве, так и в картинной плоскости.

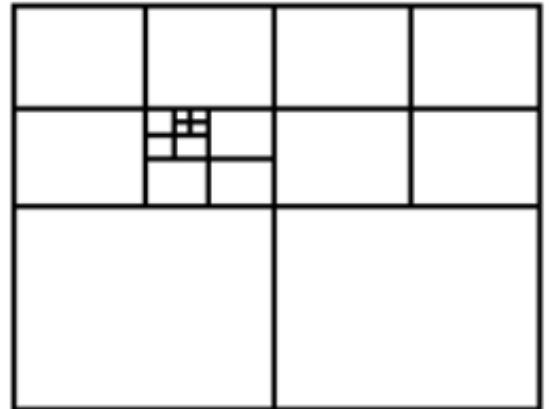
С этой целью разбиения стоятся уже на этапе пре-процессирования, и для каждой клетки разбиения составляется список всех объектов (граней), которые ее пересекают.

Простейшим вариантом разбиения является равномерное разбиение пространства на набор равных прямоугольных клеток (см. рисунок 1). Составляется список объектов, пересекающих клетку разбиения. Для отыскания всех объектов, которые закрывают рассматриваемый объект при проецировании, проверяются только объекты, попадающие в те же клетки разбиения картинной плоскости.

Для сцен с неравномерным распределением объектов имеет смысл использовать неравномерное (адаптивное) разбиение пространства или плоскости (см. рисунок 1).



**равномерное разбиение**



**неравномерное разбиение**

Рисунок 1 – Равномерное и неравномерное разбиения пространства объектов

#### 4. Иерархические древовидные структуры

При работе с большими объемами используются различные древовидные (иерархические) структуры. Стандартными формами таких структур являются восьмеричные (Octrees, для 3D), тетрарные (Quadtrees, для 2D), бинарные или BSP-деревья (Binary Space Partitioning Trees) и деревья ограничивающих тел. Иерархии позволяют упорядочивать грани объектов, производить быстрое и эффективное отсечение граней, не удовлетворяющих каким-либо из условий. [5]

Ниже будут рассмотрены основные алгоритмы удаления невидимых линий и плоскостей.

##### 1.3.2 Алгоритм плавающего горизонта

Алгоритм плавающего горизонта чаще всего используется для удаления невидимых линий трёхмерного представления функций, описывающих поверхность в виде:

$$F(x, y, z) = 0. \quad (1)$$

Подобные функции возникают во многих приложениях в математике, технике, естественных науках и других дисциплинах.

Предложено много алгоритмов, использующих этот подход. Поскольку в приложениях в основном интересуются описанием поверхности, этот алгоритм обычно работает в пространстве изображения. Главная идея данного метода заключается в сведении

трехмерной задачи к двумерной путем пересечения исходной поверхности последовательностью параллельных секущих плоскостей, имеющих постоянные значения координат  $x$ ,  $y$  или  $z$ .

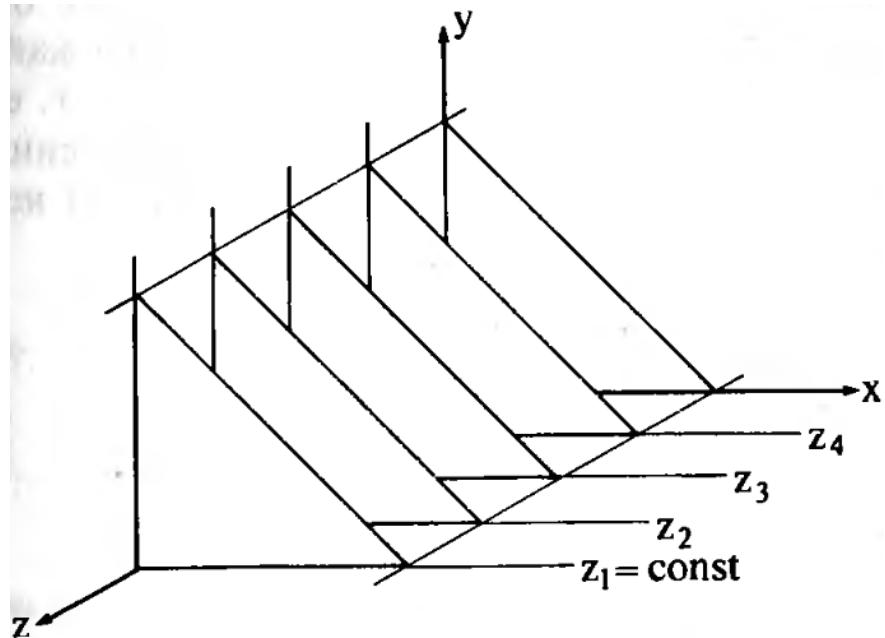


Рисунок 2 – Секущие плоскости с постоянной координатой.

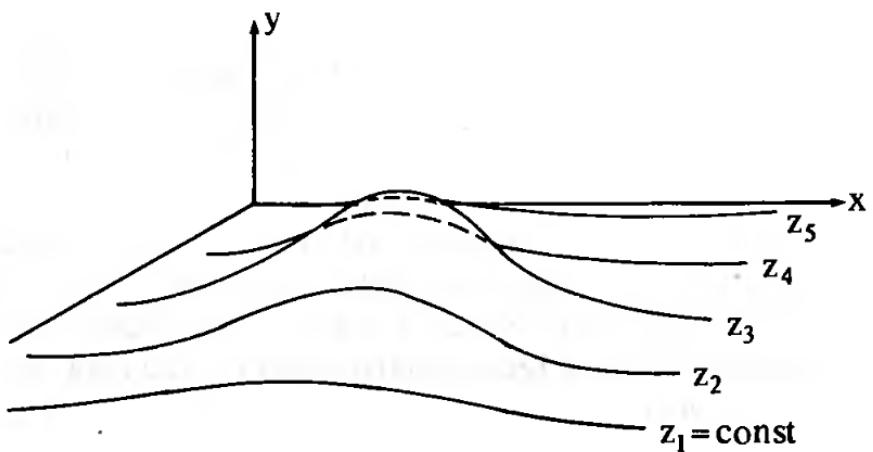


Рисунок 3 – Кривые в секущих плоскостях с постоянной координатой.

На рис. 2 приведен пример, где указанные параллельные плоскости определяются постоянными значениями  $z$ . Функция  $F(x, y, z) = 0$  сводится к последовательности кривых, лежащих в каждой из этих параллельных плоскостей, например, к последовательности

$$y = f(x, z) \quad \text{или} \quad x = g(y, z) \quad (2)$$

где  $z$  постоянно на каждой из заданных параллельных плоскостей.

Итак, поверхность теперь складывается из последовательности кривых, лежащих в каждой из этих плоскостей, как показано на рис. 3. Здесь предполагается, что полу-

ченные кривые являются однозначными функциями независимых переменных. Если спроектировать полученные кривые на плоскость  $z = 0$ , как показано на рис. 4, то сразу становится ясна идея алгоритма удаления невидимых участков исходной поверхности. Алгоритм сначала упорядочивает плоскости  $z = \text{const}$  по возрастанию расстояния до них от точки наблюдения. Затем для каждой плоскости, начиная с ближайшей к точке наблюдения, строится кривая, лежащая на ней, т.е. для каждого значения координаты  $x$  в пространстве изображения определяется соответствующее значение  $y$ .

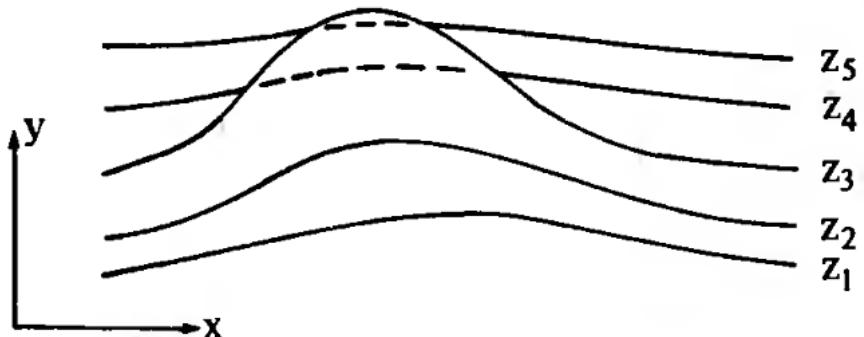


Рисунок 4 – Проекция кривых на плоскость  $z = 0$ .

Алгоритм удаления невидимой линии заключается в следующем:

Если на текущей плоскости при некотором заданном значении  $x$  соответствующее значение  $y$  на кривой больше максимума или меньше минимума по  $y$  для всех предыдущих кривых при этом  $x$ , то текущая кривая видима. В противном случае она невидима. Если на участке от предыдущего  $x_n$  до текущего  $x_{n+k}$  значения  $x$  видимость кривой изменяется, то вычисляется точка пересечения  $x_i$ . Если на участке от  $x_n$  до  $x_{n+k}$  сегмент кривой полностью видим, то он изображается целиком; если он стал невидимым, то изображается фрагмент от  $x_n$  до  $x_i$ ; если же он стал видимым, то изображается фрагмент от  $x_i$  до  $x_{n+k}$ . Заполнить массивы верхнего и нижнего плавающих горизонтов.

Типичный результат работы алгоритма можно наблюдать на рисунке 5. [1]

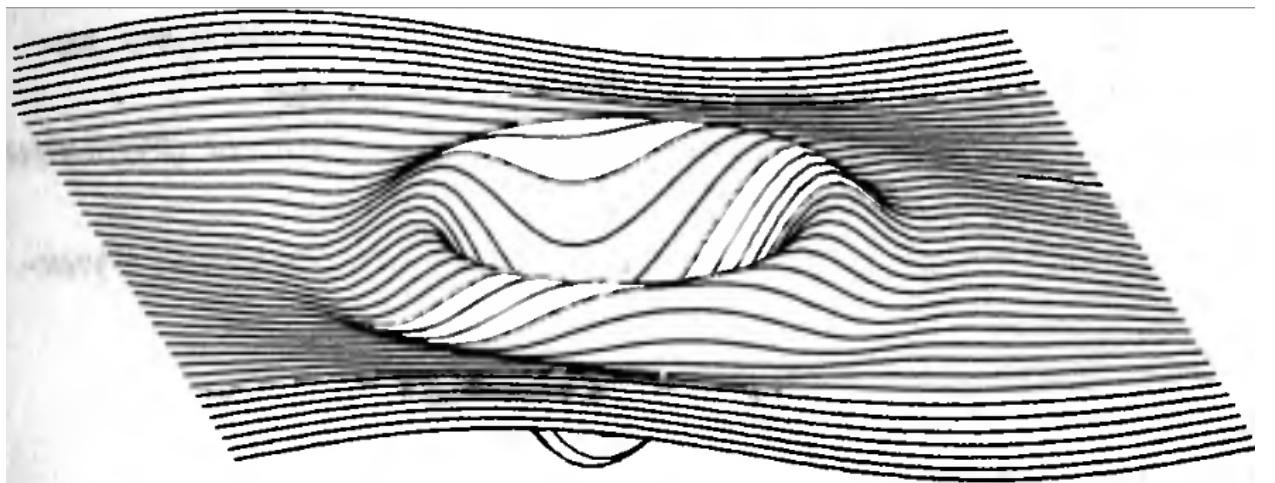


Рисунок 5 – Функция  $y = \frac{1}{5} \sin x \cos z - \frac{3}{2} \cos\left(\frac{7a}{4}\right) \times \exp(-a)$ ,  $a = (x - \pi)^2 + (z - \pi)^2$ , изображенная в интервале  $(0, 2\pi)$  с помощью алгоритма плавающего горизонта.

### 1.3.3 Алгоритм Робертса

Рассмотрим алгоритм удаления невидимых линий Робертса, требующий, чтобы каждая грань была выпуклым многоугольником.

Сначала отбрасываются все рёбра, обе определяющие грани которых являются нелицевыми (ни одно из таких рёбер заведомо не видно).

Следующим шагом является проверка на закрывание каждого из оставшихся рёбер со всеми лицевыми гранями многогранника. Возможны следующие случаи:

- грань ребра не закрывает (рис. 6);

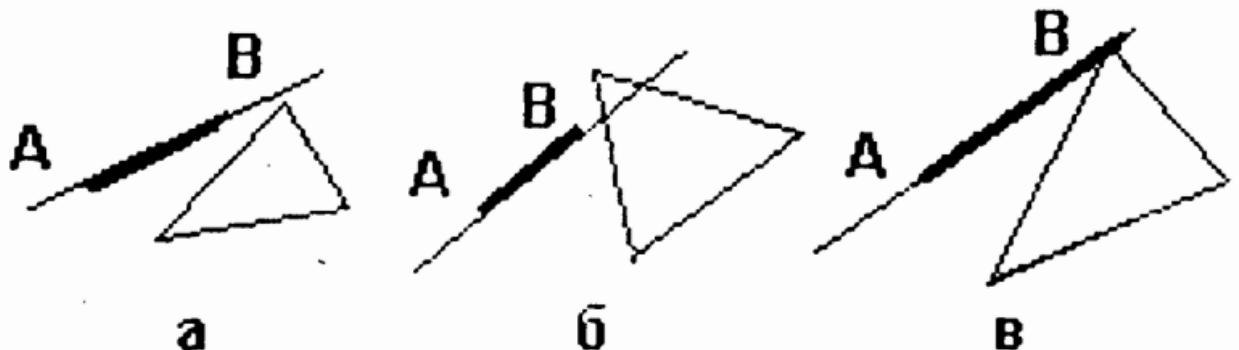


Рисунок 6 – Иллюстрация алгоритма Робертса: (а) грань не закрывает ребро, (б) грань частично закрывает ребро, (в) грань полностью закрывает ребро.

- грань полностью закрывает ребро (тогда оно удаляется из списка рассматриваемых рёбер) (рис. 7, а);
- грань частично закрывает ребро (в этом случае ребро разбивается на несколько частей, видимыми из которых являются не более двух; само ребро удаляется из списка, но в список проверяемых рёбер добавляются те его части, которые данной гранью не закрываются).

Рассмотрим, как осуществляются эти проверки.

Пусть задано ребро  $AB$ , где точка  $A$  имеет координаты  $(x_a, y_a)$ , а точка  $B$  –  $(x_b, y_b)$ .

Прямая, проходящая через отрезок  $AB$ , задаётся уравнениями:

$$x = x_a + t(x_b - x_a), \quad y = y_a + t(y_b - y_a), \quad (3)$$

причем сам отрезок соответствует значениям параметра  $0 \leq t \leq 1$ .

Данную прямую можно задать неявным образом как  $F(x, y) = 0$ , где

$$F(x, y) = (y_b - y_a)(x - x_a) - (y - y_a)(x_b - x_a). \quad (4)$$

Предположим, что проекция грани задается набором проекций вершин  $P_1, \dots, P_k$  с координатами  $(x_i, y_i)$ ,  $i = 1, \dots, n$ . Обозначим через  $F_i$  значение функции  $F$  в точке  $P_i$ .

Рассмотрим  $i$ -й отрезок проекции грани  $P_iP_{i+1}$ . Этот отрезок пересекает прямую  $AB$  тогда и только тогда, когда функция  $F$  принимает значения разных знаков на концах этого отрезка, а именно при

$$F_i F_{i+1} \leq 0. \quad (5)$$

Случай, когда  $F_i F_{i+1} = 0$ , будем отбрасывать, чтобы дважды не засчитывать прямую, проходящую через вершину, для обоих выходящих из неё отрезков.

Итак, мы считаем, что пересечение имеет место в двух случаях:

$$F_i \geq 0, F_{i+1} < 0, \quad (6)$$

$$F_i \leq 0, F_{i+1} > 0. \quad (7)$$

Точка пересечения определяется соотношениями:

$$x = x_i + s(x_{i+1} - x_i), \quad (8)$$

$$y = y_i + s(y_{i+1} - y_i), \quad (9)$$

где

$$s = \frac{F_i}{F_i - F_{i+1}}. \quad (10)$$

Отсюда легко находится значение параметра  $t$ :

$$t = \begin{cases} \frac{x-x_a}{x_b-x_a}, & |x_b - x_a| \geq |y_b - y_a|, \\ \frac{y-y_a}{y_b-y_a}, & |y_b - y_a| > |x_b - x_a|. \end{cases} \quad (11)$$

Возможны следующие случаи:

1. Отрезок не имеет пересечений с проекцией грани, кроме, быть может, одной точки. Это может иметь место, когда

- прямая  $AB$  не пересекает рёбра проекции грани (рис. 6, а);
- прямая  $AB$  пересекает ребра в двух точках  $t_1$  и  $t_2$ , но либо  $t_1 < 0, t_2 < 0$ , либо  $t_1 > 1, t_2 > 1$  (рис. 6, б);
- прямая  $AB$  проходит через одну вершину, не задевая внутренности треугольника (рис. 6, в).

Очевидно, что в этом случае соответствующая грань никак не может закрывать собой ребро  $AB$ .

2. Проекция ребра полностью содержится внутри проекции грани (рис. 7, а). Тогда есть две точки пересечения прямой  $AB$  и границы грани и  $t_1 < 0 < t_2 < 1$ . Если грань лежит ближе к картинной плоскости, чем ребро, то ребро полностью невидимо и удаляется.

3. Прямая  $AB$  пересекает ребра проекции грани в двух точках и либо  $t_1 < 0 \leq t_2 \leq 1$ , либо  $0 < t_1 \leq 1 < t_2$  (рис. 7, б и в). Если ребро  $AB$  находится дальше от картинной плоскости, чем соответствующая грань, то оно разбивается на две части, одна из которых полностью закрывается гранью и потому отбрасывается. Проекция второй части лежит вне проекции грани и поэтому этой гранью не закрывается.

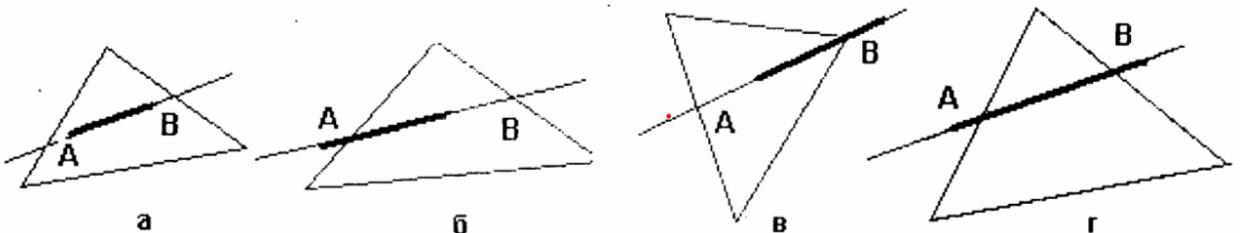


Рисунок 7

4. Прямая  $AB$  пересекает ребра проекции грани в двух точках, причем  $0 < t_1 < t_2 < 1$  (рис. 7, г). Если ребро  $AB$  лежит дальше от картинной плоскости, чем соответствующая грань, то оно разбивается на три части, средняя из которых отбрасывается.

Для определения того, что лежит ближе к картинной плоскости — отрезок  $AB$  (проекция которого лежит в проекции грани) или сама грань, через эту грань проводится плоскость:

$$(n, p) + c = 0 \quad (12)$$

( $n$  — нормальный вектор грани), разбивающая всё пространство на два полупространства. Если оба конца отрезка  $AB$  лежат в том же полупространстве, в котором находится и наблюдатель, то отрезок лежит ближе к грани; если оба конца находятся в другом полупространстве, то отрезок лежит дальше. Случай, когда концы лежат в разных полупространствах, здесь невозможен (это означало бы, что отрезок  $AB$  пересекает внутреннюю часть грани).

Если общее количество граней равно  $n$ , то временные затраты для данного алгоритма составляют  $O(n^2)$ .

Количество проверок можно заметно сократить, если воспользоваться разбиением картинной плоскости.

Разобьем видимую часть картинной плоскости (экран) на  $N_1 \times N_2$  равных частей (клеток) и для каждой клетки  $A_{ij}$  построим список всех лицевых граней, чьи проекции имеют с данной клеткой непустое пересечение.

Для проверки произвольного ребра на пересечение с гранями отберем сначала все те клетки, которые проекция данного ребра пересекает. Ясно, что проверять на пересечение с ребром имеет смысл только те грани, которые содержатся в списках этих клеток.

В качестве шага разбиения обычно выбирается  $O(l)$ , где  $l$  — характерный размер ребра в сцене. Для любого ребра количество проверяемых граней практически не зависит

сит от общего числа граней и совокупные временные затраты алгоритма на проверку пересечений составляют  $O(n)$ , где  $n$  — количество ребер в сцене.

Поскольку процесс построения списков заключается в переборе всех граней, их проектировании и определении клеток, в которые попадают проекции, то затраты на составление всех списков также составляют  $O(n)$ . [4]

#### 1.3.4 Алгоритм Варнака

Джон Варнак, американский учёный, один из основателей компании Adobe Systems. Алгоритм Варнака позволяет определить, какие грани или части граней объектов сцены видимы, а какие заслонены гранями других объектов. Так же как и в алгоритме Робертса анализ видимости происходит в картинной плоскости. В качестве граней обычно выступают выпуклые многоугольники, алгоритмы работы с ними эффективнее, чем с произвольными многоугольниками. Окно, в котором необходимо отобразить сцену, должно быть прямоугольным. Алгоритм работает рекурсивно, на каждом шаге анализируется видимость граней и, если нельзя легко определить видимость, окно делится на 4 части и анализ повторяется отдельно для каждой из частей (см. рис. 8).

Случаи, когда можно легко определить видимость, следующие:

- a) в окне нет ни одной грани, в этом случае ничего рисовать не нужно;
- b) в окне ровно одна грань, в этом случае достаточно отобразить эту грань, так как нет других граней, которые могли бы её заслонить;
- c) размеры окна равны размерам одного пикселя, в этом случае из граней выбирается ближайшая и её цветом закрашивается этот пикセル;
- d) ближайшая грань охватывает всё окно, в этом случае она заслоняет все остальные грани и достаточно отобразить только её.

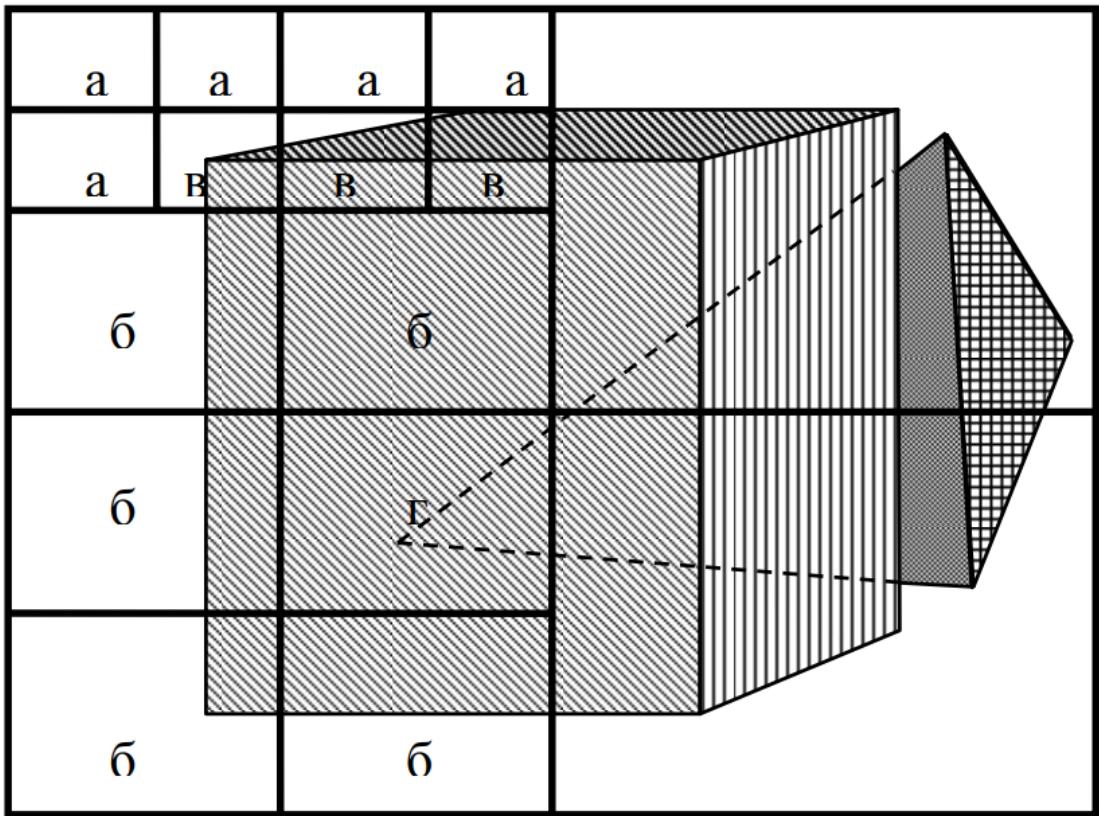


Рисунок 8 – Определение видимости алгоритмом Варнака.

Алгоритм можно разделить на следующие элементарные задачи:

- определение, в какие части окна попадает грань;
- определение, что грань охватывает окно;
- определение, что грань является ближайшей.

Существует два различных подхода к решению этих задач.

Сначала рассмотрим вариант решения первой задачи, когда при разделении окна грани разрезаются на части вертикальной и горизонтальной прямыми, делящими окно. Получившиеся многоугольники полностью лежат в соответствующих частях окна, и заносятся в соответствующий этой части набор. При последующем делении окна они разрезаются на более мелкие части. Определить, что такой многоугольник охватывает всё окно можно по условию, что каждая из четырёх вершин окна совпадает с одной из вершин многоугольника. Определить, что грань ближайшая можно по глубине вершин многоугольника. При перспективной проекции в качестве параметра глубины удобно использовать обратную величину  $w = 1/z$ , где  $z$  – координата глубины, так как  $w$  в отличие от  $z$  можно линейно интерполировать в экранных координатах. Чем больше  $w$ , тем ближе вершина к наблюдателю. Так как грань не обязательно параллельна экрану, то глубина у вершин разная. Найдя минимум и максимум глубины всех вершин можно определить интервал глубины  $[W_{\min}, W_{\max}]$ . В случае, когда окно равно пиксели,

достаточно выбрать грань с максимальным  $W_{max}$ , она и будет ближайшей. Если же нужно убедиться, что грань ближе всех остальных граней, то необходимо проверить условие, что  $W_{min}$  этой грани больше  $W_{max}$  всех остальных граней.

Второй подход не требует деления граней на части, для вычислений всегда используются исходные грани.

Для решения первой задачи нужно определить факт пересечения двух выпуклых многоугольников – грани и окна. Это можно сделать, например, по следующему условию: многоугольники не пересекаются, если все вершины одного из них лежат с внешней стороны от прямой, образующей ребро другого (см. рис. 9). Нужно перебрать все рёбра грани и сравнить с вершинами окна, и все четыре границы окна сравнить с вершинами грани.

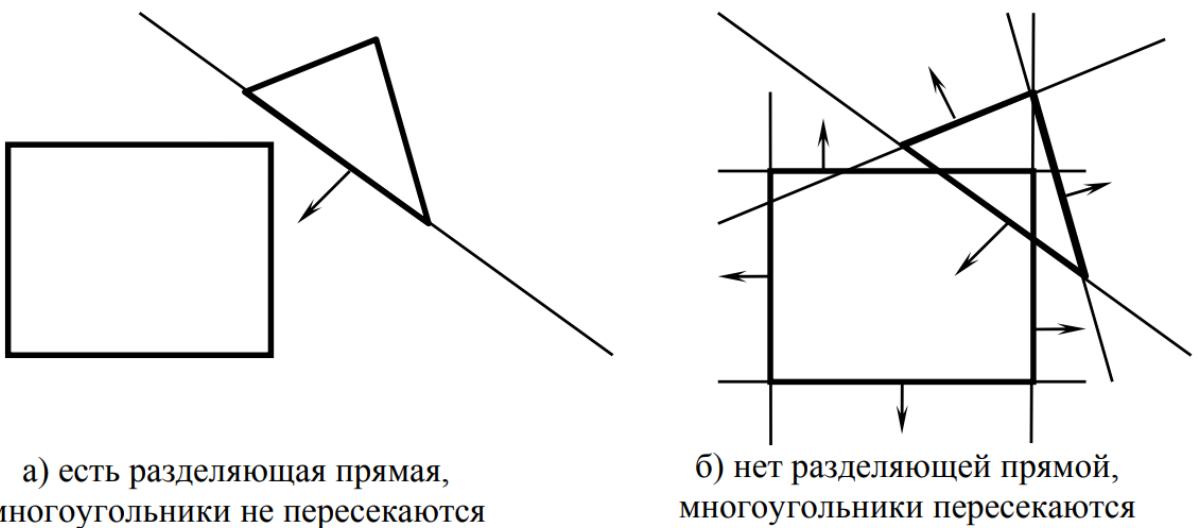


Рисунок 9 – Определение пересечения выпуклых многоугольников.

Для решения второй задачи достаточно проверить, что все 4 угла окна лежат внутри или на границе грани. Это можно определить по расстояниям от углов окна до прямых, образующих рёбра грани.

Третья задача решается сложнее, чем в первом варианте. Грань не лежит полностью в окне, поэтому рассчитывать интервал её глубины по вершинам будет неправильно. Необходимы вершины той её части, которая лежит в окне. Это вершины грани, попадающие в окно, вершины окна, попадающие в грань и точки пересечения рёбер грани с рёбрами окна (см. рис. 10).

По этим точкам рассчитывается интервал глубины  $[W_{min}, W_{max}]$  и используется для определения ближайшей грани.

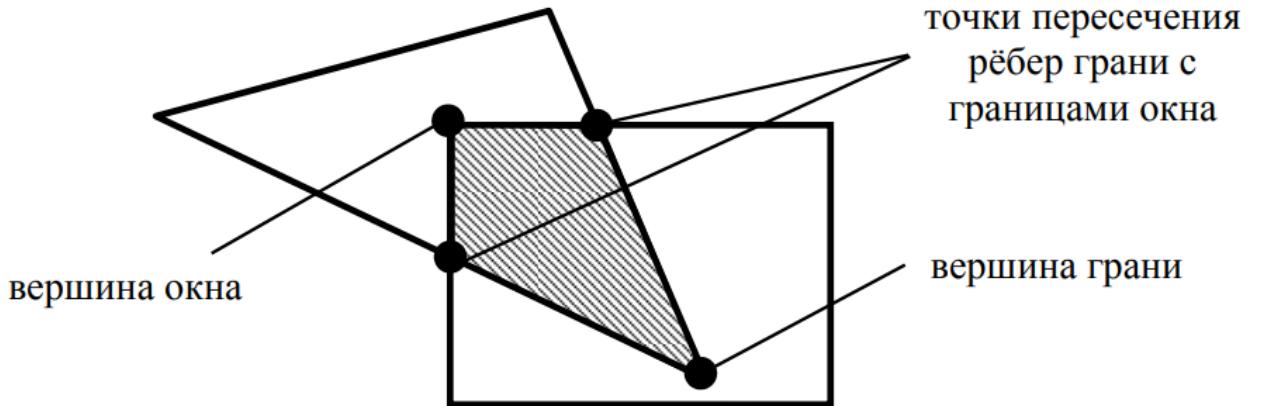


Рисунок 10 – Определение вершин пересечения грани с окном.

У каждого из подходов есть свои достоинства и недостатки. Подход с разрезанием граней проще в реализации, но требует дополнительной памяти для хранения частей граней. Второй подход лишён этого недостатка, но более сложен в реализации. Производительность при обоих подходах примерно одинакова. Её можно оценить, исходя из количества граней  $N$  и количества пикселей  $C$  как  $O(CN)$ , так как в худшем случае все грани могут быть размером с окно и окно в худшем случае придётся делить до пикселя. Уменьшить количество граней нельзя, но можно сократить количество делений окна. Например, его можно сократить, если дополнительно обрабатывать случай на общем ребре двух граней. По описанному алгоритму окно, в которое попадает общее ребро двух граней, будет делиться до пикселя. Это происходит потому, что не выполняется ни один из четырёх случаев, в окне несколько граней и ни одна из этих граней не охватывает окно (см. рис. 11).

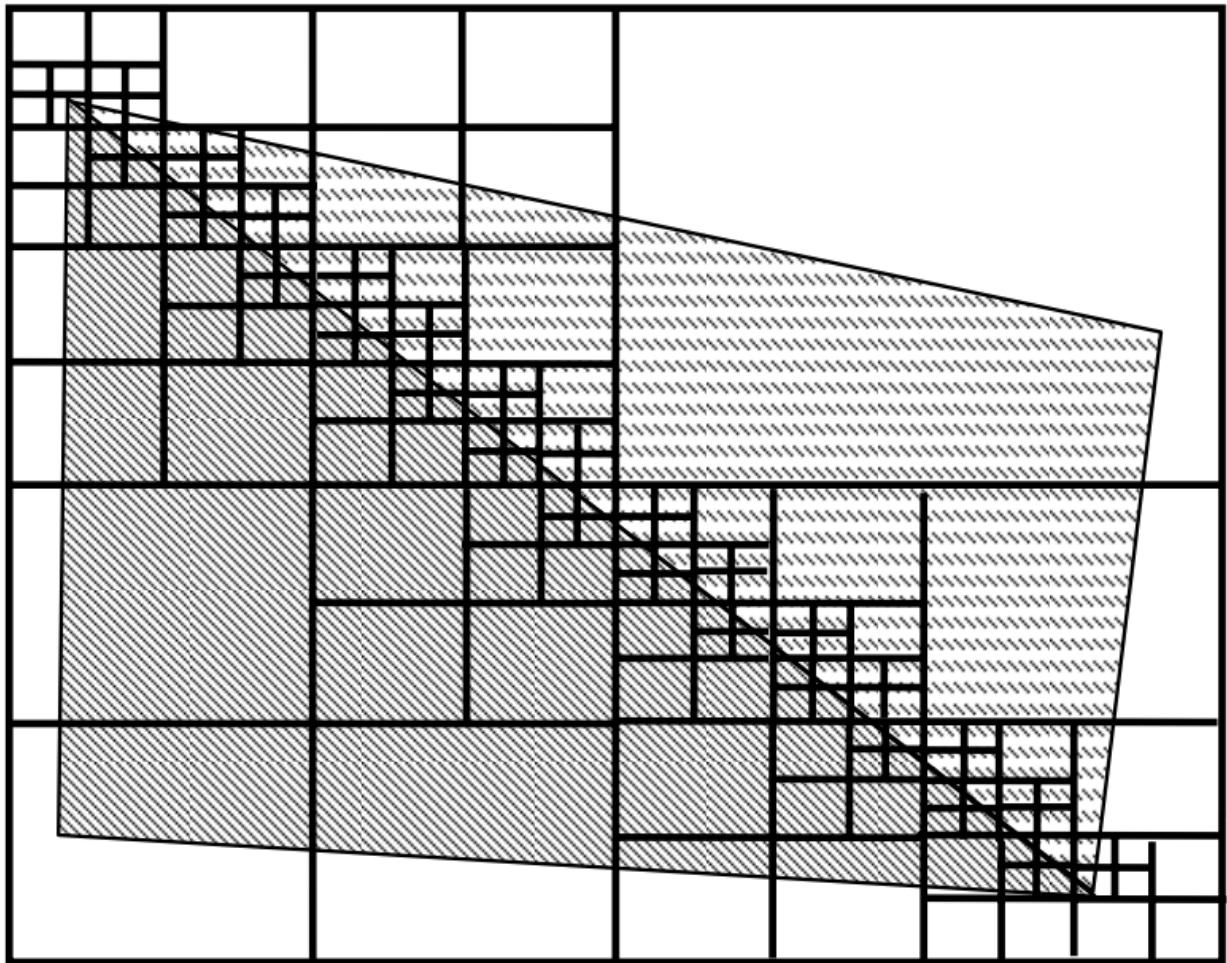


Рисунок 11 – Разделение окна вдоль общего ребра двух граней.

Можно избежать лишних делений, если выделить общие рёбра граней, и добавить ещё один простой случай: две грани с общим ребром полностью охватывают окно и являются ближайшими. Ребро в этом случае должно пересекать окно в двух точках, вершины окна с одной стороны ребра должны лежать внутри грани лежащей с этой стороны ребра, а вершины окна с другой стороны ребра должны лежать внутри второй грани (см. рис. 12).

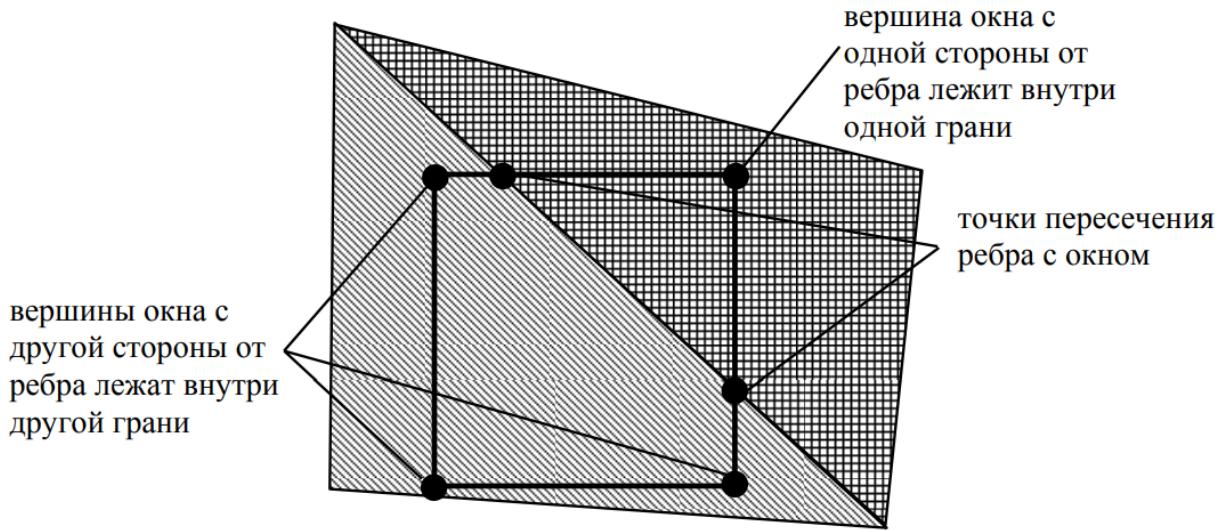


Рисунок 12 – Определение случая с двумя гранями с общим ребром.

Интервал глубины  $[W_{min}, W_{max}]$  должен рассчитываться по вершинам обеих граней, попавших в окно частей. [3]

### 1.3.5 Алгоритм Вейлера-Азертонса

Вейлер и Азертон попытались минимизировать количество шагов в алгоритме разбиения типа алгоритма Варнока путем разбиения окна вдоль границ многоугольника. Выходными данными этого алгоритма, который для достижения необходимой точности работает в пространстве объекта, служат многоугольники. Поскольку выходом являются многоугольники, то алгоритм можно легко использовать для удаления как невидимых линий, так и невидимых поверхностей. Алгоритм удаления невидимых поверхностей состоит из четырех шагов.

1. Предварительная сортировка по глубине.
2. Отсечение по границе ближайшего к наблюдателю многоугольника, называемое сортировкой многоугольников на плоскости.
3. Удаление многоугольников, экранированных многоугольником, ближайшим к точке наблюдения.
4. Если требуется, то рекурсивное подразделение и окончательная сортировка для устранения всех неопределенностей.

Предварительная сортировка по глубине нужна для формирования списка приблизительных приоритетов. Предположим, что точка наблюдения расположена в бесконечности на положительной полуоси  $z$ , тогда ближайшим к ней и первым в списке будет тот многоугольник, который обладает вершиной с максимальной координатой  $z$ .

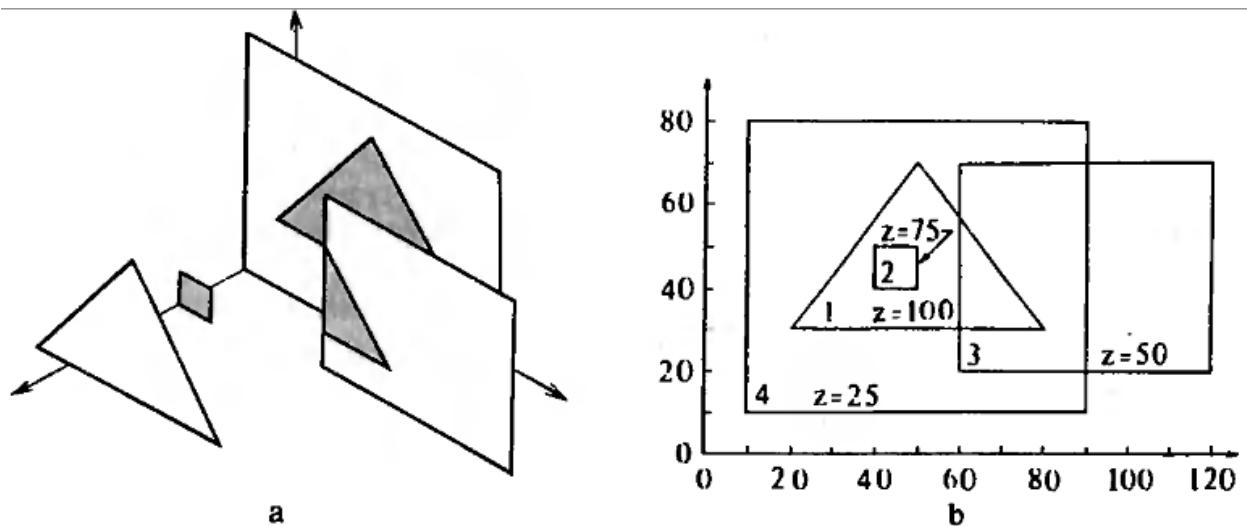
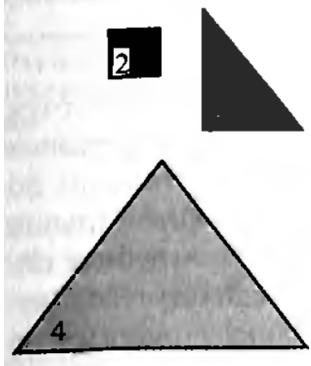


Рисунок 13 – Отсечение многоугольников по приоритетам для алгоритма удаления невидимых поверхностей Вейлера — Азертона.

В качестве отсекающего многоугольника используется копия первого многоугольника из предварительного списка приоритетов по глубине. Отсекаться будут оставшиеся в этом списке многоугольники, включая и первый многоугольник. Вводятся два списка: внутренний и внешний. С помощью алгоритма отсечения Вейлера — Азертона все многоугольники отсекаются по границам отсекающего многоугольника. Фактически это двумерная операция отсечения проекций отсекающего и отсекаемого многоугольников. Та часть каждого отсекаемого многоугольника, которая оказывается внутри отсекающего, если она имеется, попадает во внутренний список. Оставшаяся часть, если таковая есть, попадает во внешний список. Этот этап алгоритма является сортировкой на плоскости или ху-сортировкой. Пример приведен на рис. 13. На рис. 14 показаны внутренний и внешний списки для сцены на рис. 13. Теперь сравниваются глубины каждого многоугольника из внутреннего списка с глубиной отсекающего многоугольника. С использованием координат  $(x, y)$  вершин отсекаемых многоугольников и уравнений несущих плоскостей вычисляются глубины (координаты  $z$ ) каждой вершины. Затем они сравниваются с минимальной координатой  $z(z_{min})$  для отсекающего многоугольника. Если глубина ни одной из этих вершин не будет больше  $z_{min}$ , то все многоугольники из внутреннего списка экранируются отсекающим многоугольником (рис. 13). Эти многоугольники удаляются, и выбирается следующий многоугольник. Заметим, что во внутренний список остается лишь отсекающий многоугольник. Работа алгоритма затем продолжается с внешним списком.

**Внутренний список**



**Внешний список**

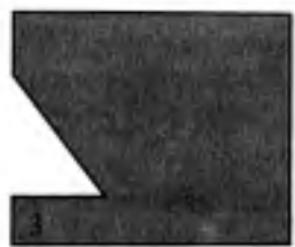
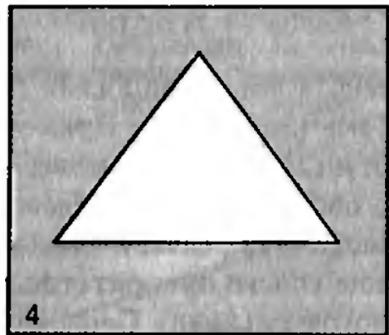


Рисунок 14 – Внутренний и внешний списки многоугольников.

Если координата  $z$  какого-либо многоугольника из внутреннего списка окажется больше, чем  $z_{c \min}$ , то такой многоугольник по крайней мере частично экранирует отсекающий многоугольник. На рис. 15 показано, как это может произойти. В подобном случае результат предварительной сортировки по глубине ошибочен. Поэтому алгоритм рекурсивно подразделяет плоскость  $(x, y)$ , используя многоугольник, нарушивший порядок, в качестве нового отсекающего многоугольника. Отсечению подлежат многоугольники из внутреннего списка, причем старый отсекающий многоугольник теперь сам будет подвергнут отсечению новым отсекающим многоугольником. Подчеркнем, что новый отсекающий многоугольник является копией исходного многоугольника, а не его остатка после первого отсечения. Использование копии неотсекаемого многоугольника позволяет минимизировать число разбиений.

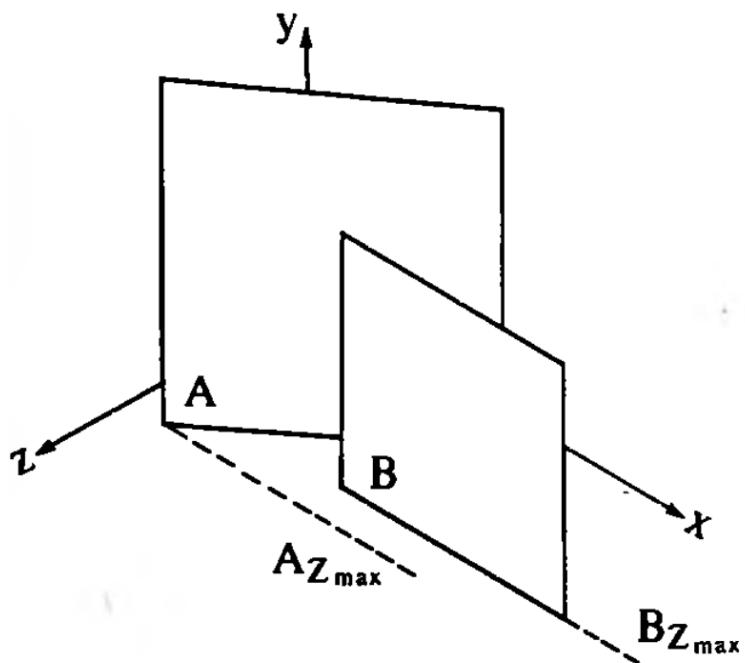


Рисунок 15 – Условие возникновения ошибочного результата в предварительной сортировке по  $z$ .

Заслуживает внимание еще одна дополнительная деталь этого алгоритма. Ко-

гда некоторый многоугольник циклически перекрывается с отсекающим, т. е. когда он лежит как впереди, так и позади отсекающего (рис. 16, а), то в рекурсоном разбиении необходимости нет. дело в том, что все экранируемое циклическим многоугольником уже было удалено на предыдущем шаге отсечения. Необходимо лишь произвести отсечение исходного многоугольника по границам циклического многоугольника и изобразить результат. Ненужное рекурсивное разбиение можно предотвратить, если создать список многоугольников, которые уже использовались как отсекающие. Тогда, если при рекурсивном разбиении текущий отсекающий многоугольник появляется в этом списке, значит, обнаружен циклически перекрывающийся многоугольник. Следовательно, не требуется никакого дополнительного разбиения. Заметим, что данный алгоритм непосредственно обрабатывает случаи циклического перекрытия сразу нескольких многоугольников, как показано на рис. 16, б. [1]

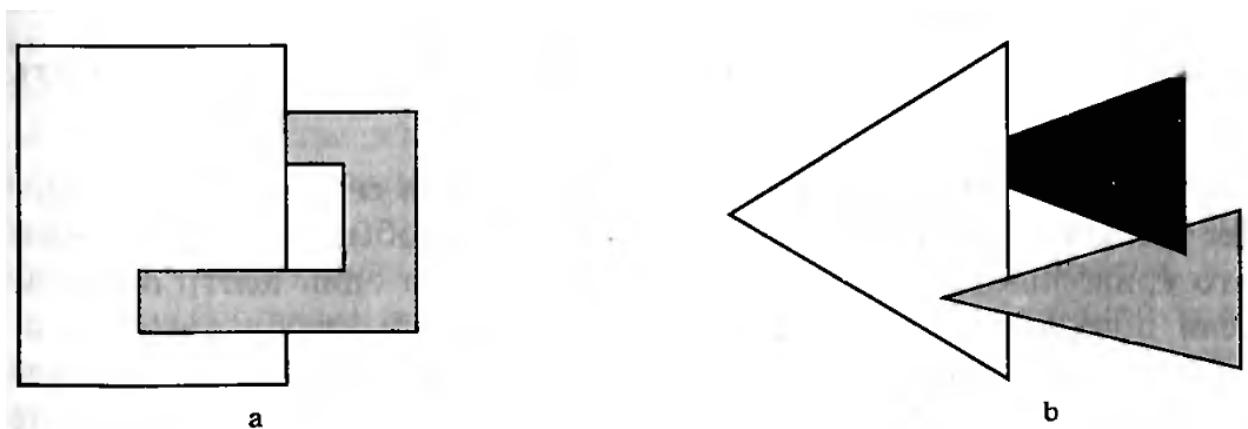


Рисунок 16 – Циклически перекрывающиеся многоугольники.

### 1.3.6 Алгоритм художника

Алгоритм художника позволяет определить, какие пиксели граней сцены видимы, а какие заслонены гранями других объектов. Для этого все грани сортируются и рисуются в порядке от дальних к ближним, в результате чего ближние грани автоматически заслоняют дальние (см. рис. 17).

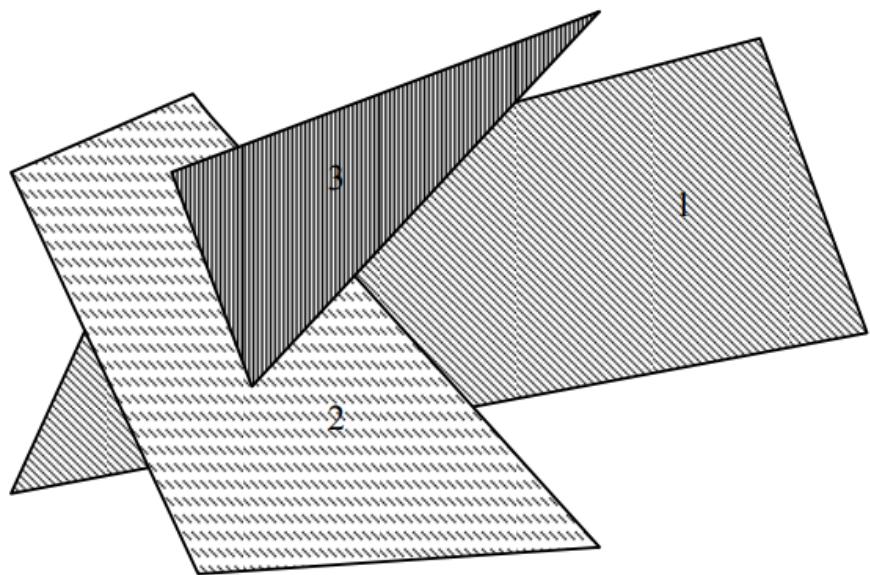


Рисунок 17 – Отрисовка граней по алгоритму художника.

Сортировка граней от дальних к ближним задача нетривиальная, недостаточно отсортировать грани по координате, соответствующей глубине. Каждой грани обычно соответствует некоторый интервал глубины, и если интервалы двух граней пересекаются, то нужны дополнительные проверки, чтобы определить, какая грань ближе. Кроме того, встречаются ситуации, когда грани отсортировать невозможно, например, когда грани образуют цикл (см. рис. 18а). В этом случае нужно разделить одну из граней на части (см. рис. 18б), и тогда становится возможным определить порядок рисования.

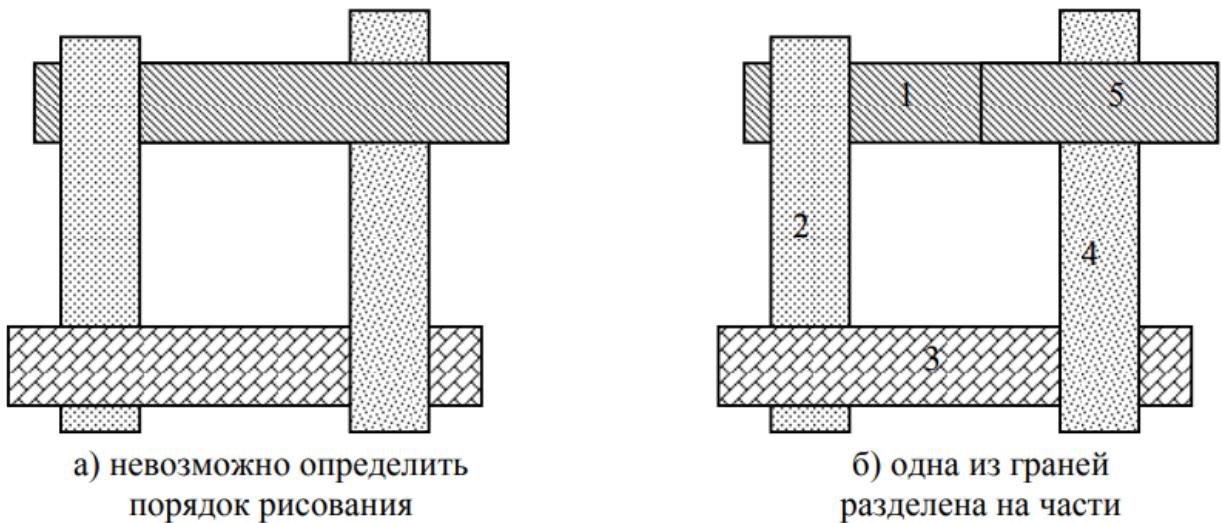


Рисунок 18 – Циклически закрывающие друг друга грани.

Одним из методов сортировки граней является метод бинарного разделения пространства. В качестве исходных данных выступает набор граней в пространстве и точка, в которой находится наблюдатель. Если разделить пространство плоскостью одной из граней, то грани, лежащие в полупространстве, в котором находится наблюдатель, будут ближе к нему, чем те, что лежат в другом полупространстве. Поэтому порядок

рисования граней будет следующим:

1. грани, лежащие в полупространстве, где нет наблюдателя;
2. грани, лежащие на разделяющей плоскости;
3. грани, лежащие в полупространстве, где находится наблюдатель.

Аналогично сортируются и грани в полупространствах, в каждом выбирается грань, её плоскостью полупространство делится на две части и так до тех пор, пока в каждой части не останется ни одной грани.

Время, необходимое на сортировку, зависит от количества граней на всех этапах разделения пространства. Например, при первом делении нужно сравнить с разделяющей плоскостью все  $N$  граней. Количество граней может возрасти из-за разделения граней на части, поэтому лучше выбирать такие плоскости, которые делят минимум граней. Также алгоритм будет работать эффективнее, если с каждой стороны от разделяющей плоскости будет примерно одинаковое количество граней.

Рассмотрим два крайних случая:

1. Все грани оказываются с одной стороны от плоскости, кроме одной, которая является разделяющей. Тогда количество сравнений на первом шаге будет  $N$ , на втором  $N - 1$ , на третьем  $N - 2$  и так далее до 1. Общее количество делений будет  $N$ , а сложность  $O(N^2)$ .
2. Грани с обеих сторон распределены поровну. Количество сравнений на первом шаге будет  $N$ , на втором  $N/2 + N/2$  (по  $N/2$  с каждой стороны), на третьем  $N/4 + N/4 + N/4 + N/4$  и так далее, пока не останется  $1 + 1 + \dots + 1$ . Общее количество делений будет  $\log N$ , а сложность  $O(N \cdot \log N)$ . [3]

### 1.3.7 Алгоритм трассировки лучей

Алгоритм трассировки лучей позволяет определить, какие пиксели граней сцены видимы, а какие заслонены гранями других объектов. Определение происходит непосредственно для каждого пикселя. Для этого ищется пересечение луча, исходящего из точки наблюдателя через центр пикселя, с гранями сцены и выбирается ближайшая к наблюдателю грань, она и будет видима в данном пикселе (см. рис. 19).

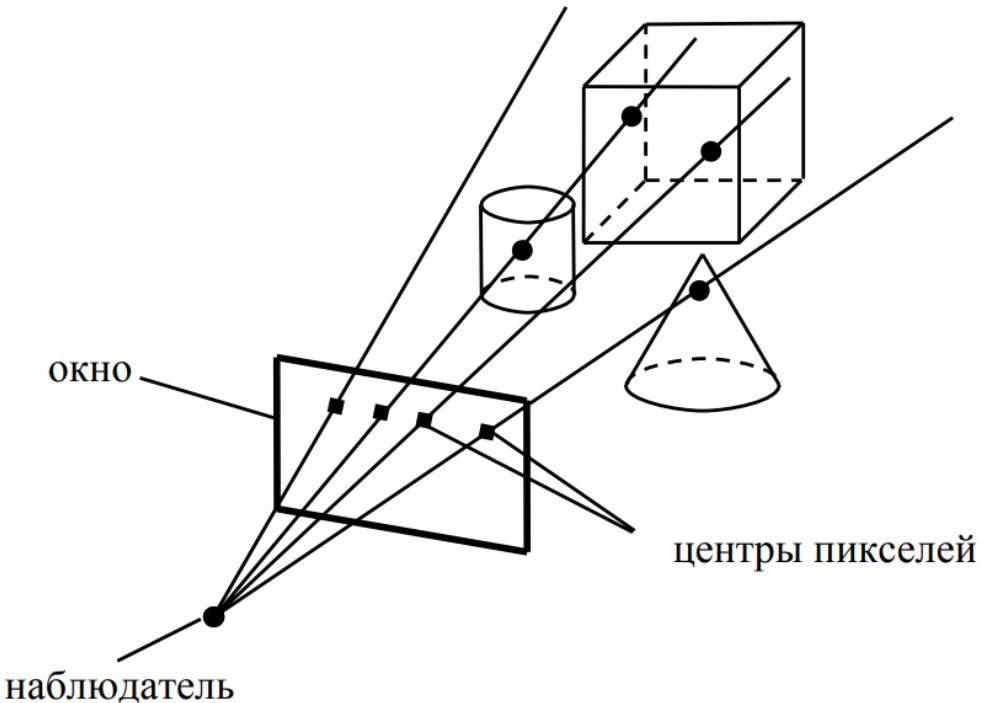


Рисунок 19 – Трассировка лучей.

Скорость работы алгоритма можно оценить как  $O(CN)$ , где  $N$  – количество граней,  $C$  – количество пикселей, так как необходимо луч для каждого пикселя проверить на пересечение с каждой из граней.

Для повышения производительности можно воспользоваться когерентностью в пространстве. Границы обычно образуют поверхность объектов, то есть располагаются в пространстве компактными группами. Вокруг таких групп можно описать простые оболочки, например, сферы, и сначала проверять луч на пересечение с оболочкой, и только если он с ней пересекается, искать пересечение с отдельными гранями.

Также можно использовать когерентность в картинной плоскости. Если для текущего пикселя была найдена ближайшая грань, то она скорее всего будет ближайшей и для соседних пикселей. Таким образом, для соседнего пикселя можно сначала проверить луч на пересечение с этой гранью. Если оно имеет место, то с остальными гранями достаточно проверять пересечение не всего луча, а только отрезка от наблюдателя до найденной точки пересечения. Это позволит быстрее отбросить грани или описанные тела, находящиеся дальше от наблюдателя, чем эта точка пересечения. [3]

### 1.3.8 Алгоритм Z-буфера

Алгоритм, использующий z-буфер это один из простейших алгоритмов удаления невидимых поверхностей. Впервые он был предложен Кэтмулом. Работает этот алгоритм в пространстве изображения. Идея z-буфера является простым обобщением идеи о буфере кадра. Буфер кадра используется для запоминания атрибутов (интенсивности) каждого пикселя в пространстве изображения, z-буфер - это отдельный буфер глубины, используемый для запоминания координаты  $z$  или глубины каждого видимого пикселя

в пространстве изображения. В процессе работы глубина или значение  $z$  каждого нового пикселя, который нужно занести в буфер кадра, сравнивается с глубиной того пикселя, который уже занесен в  $z$ -буфер. Если это сравнение показывает, что новый пиксель расположен впереди пикселя, находящегося в буфере кадра, то новый пиксель заносится в этот буфер и, кроме того, производится корректировка  $z$ -буфера новым значением  $z$ . Если же сравнение дает противоположный результат, то никаких действий не производится. По сути, алгоритм является поиском по  $x$  и  $y$  наибольшего значения функции  $z(x, y)$ .

Главное преимущество алгоритма – его простота. Кроме того, этот алгоритм решает задачу об удалении невидимых поверхностей и делает тривиальной визуализацию пересечений сложных поверхностей. Сцены могут быть любой сложности. Поскольку габариты пространства изображения фиксированы, оценка вычислительной трудоемкости алгоритма не более чем линейна. Поскольку элементы сцены или картинки можно заносить в буфер кадра или в  $z$ -буфер в произвольном порядке, их не нужно предварительно сортировать по приоритету глубины. Поэтому экономится вычислительное время, затрачиваемое на сортировку по глубине.

Основной недостаток алгоритма - большой объем требуемой памяти. Если сцена подвергается видовому преобразованию и отсекается до фиксированного диапазона значений координат  $z$ , то можно использовать  $z$ -буфер с фиксированной точностью. Информацию о глубине нужно обрабатывать с большей точностью, чем координатную информацию на плоскости  $(x, y)$ ; обычно бывает достаточно 20-ти бит. Буфер кадра размером 512'512'24 бит в комбинации с  $z$ -буфером размером 512'512'20 бит требует почти 1.5 мегабайт памяти. Однако снижение цен на память делает экономически оправданным создание специализированных запоминающих устройств для  $z$ -буфера и связанной с ним аппаратуры.

Альтернативой созданию специальной памяти для  $z$ -буфера является использование для этой цели оперативной памяти. Уменьшение требуемой памяти достигается разбиением пространства изображения на 4, 16 или больше квадратов или полос. В предельном варианте можно использовать  $z$ -буфер размером в одну строку развертки. Для последнего случая имеется интересный алгоритм построчного сканирования. Поскольку каждый элемент сцены обрабатывается много раз, то сегментирование  $z$ -буфера, вообще говоря, приводит к увеличению времени, необходимого для обработки сцены. Однако сортировка на плоскости, позволяющая не обрабатывать все многоугольники в каждом из квадратов или полос, может значительно сократить этот рост.

Другой недостаток алгоритма  $z$ -буфера состоит в трудоемкости и высокой стоимости устранения лестничного эффекта, а также реализации эффектов прозрачности и просвечивания. Поскольку алгоритм заносит пиксели в буфер кадра в произвольном порядке, то нелегко получить информацию, необходимую для методов устранения лестничного эффекта, основывающихся на предварительной фильтрации. При реализации эффектов прозрачности и просвечивания пиксели могут заноситься в буфер кадра в

некорректном порядке, что ведет к локальным ошибкам.

Формальное описание алгоритма z-буфера таково:

1. Заполнить буфер кадра фоновым значением интенсивности или цвета.
  2. Заполнить z-буфер минимальным значением z.
  3. Преобразовать каждый многоугольник в растровую форму в произвольном порядке.
  4. Для каждого Пиксел(x,y) в многоугольнике вычислить его глубину  $z(x,y)$ .
  5. Сравнить глубину  $z(x,y)$  со значением Zбуфер(x,y), хранящимся в z-буфере в этой же позиции.

Если  $z(x,y) > Z\text{буфер}(x,y)$ , то записать атрибут этого многоугольника (интенсивность, цвет и т.п.) в буфер кадра и заменить  $Z\text{буфер}(x,y)$  на  $z(x,y)$ . В противном случае никаких действий не производить. [6]

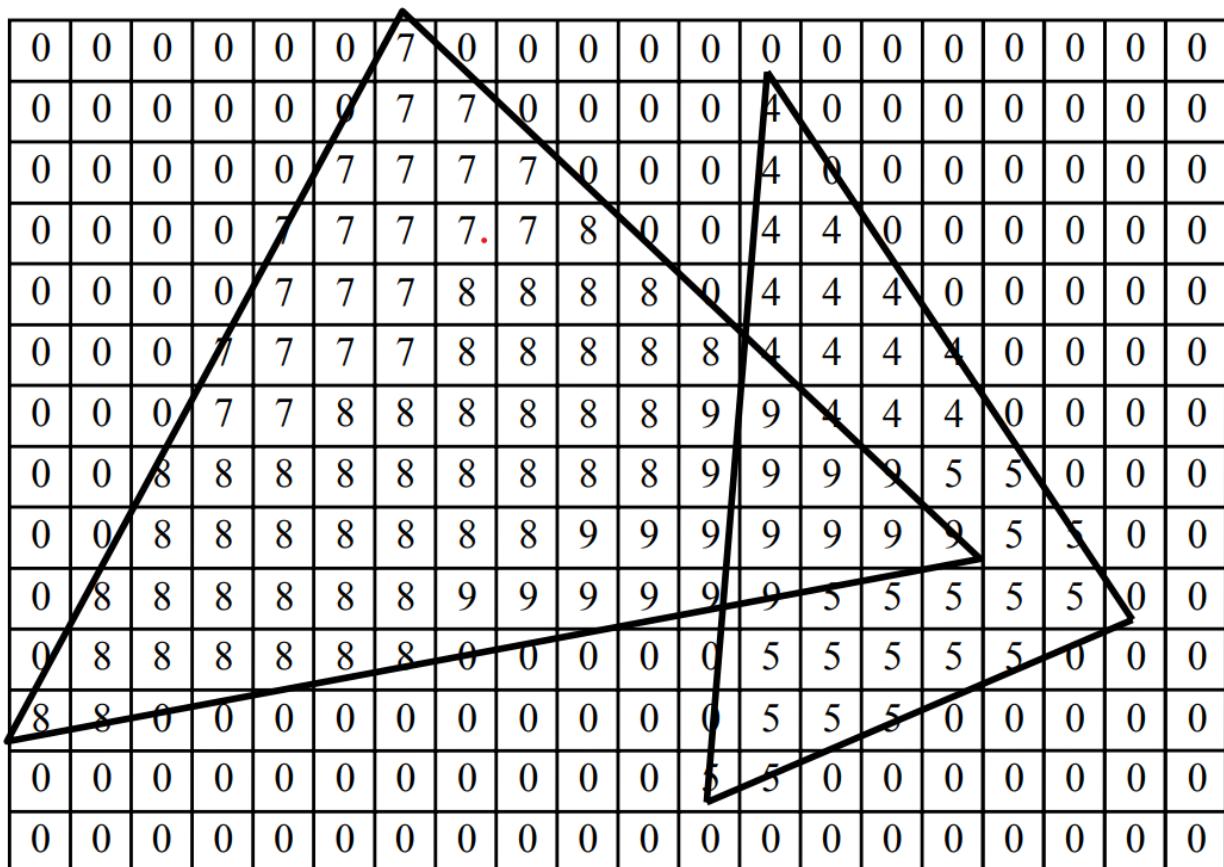


Рисунок 20 – Значения в Z-буфере.

### 1.3.9 Методы построчного сканирования

Суть метода построчного сканирования заключается в том, что через каждую строку пикселей окна проводится плоскость, дающая сечение сцены. При пересечении

поверхностей объектов этими плоскостями образуются некоторые линии, например, при пересечении граней – отрезки, и задача видимости решается для этого набора отрезков (см. рис. 21). Таким образом, задача определения видимых участков граней в трёхмерном пространстве преобразуется в набор задач определения видимости отрезков на плоскости, которые решаются проще и эффективнее.

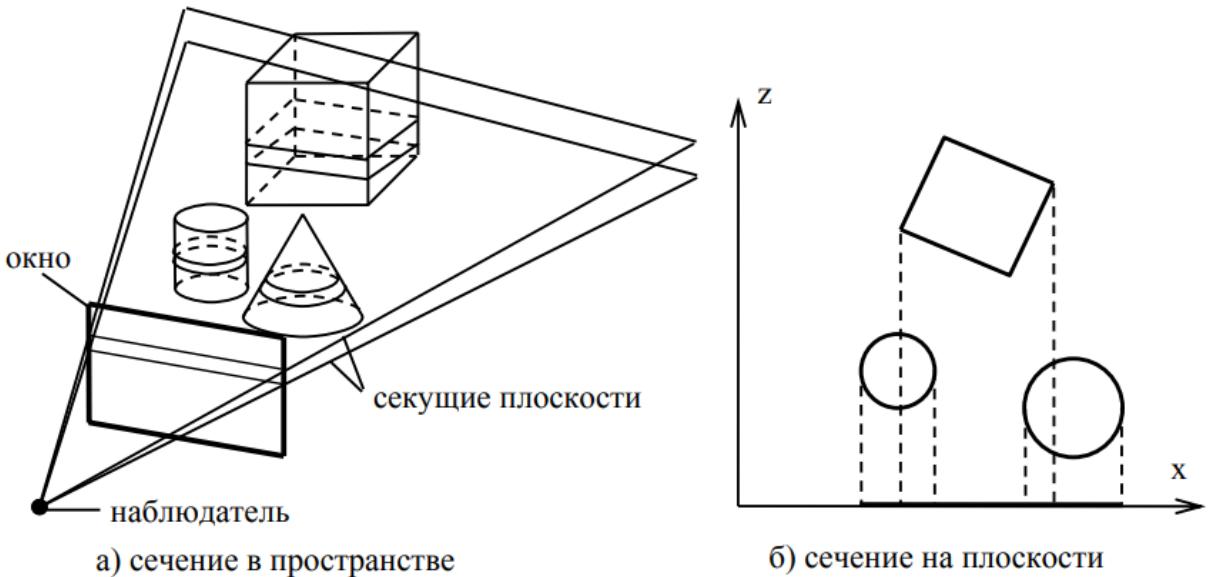


Рисунок 21 – Метод построчного сканирования.

Пересечение плоскости и выпуклой грани является отрезком, на экране он отображается в виде горизонтального отрезка пикселей – спана. [3]

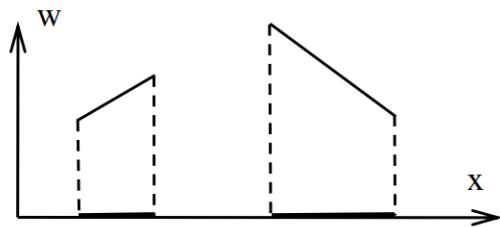
#### 1.3.9.1 Алгоритм построчного Z-буфера

Алгоритм работает аналогично обычному Z-буферу (см. разд. 1.3.8), единственное отличие в том, что вместо двухмерного массива значений глубины, достаточно одномерного массива размером в строку пикселей. Каждому пикселу соответствует свой элемент массива, перед рисованием строки в него заносится максимальное значение глубины, а при рисовании пикселей спанов их глубина сравнивается с глубиной в Z-буфере и, если пикセル ближе, он рисуется, а его глубина заносится в Z-буфер. Построчный алгоритм более эффективен, чем полноэкранный, так как Z-буфер требует значительно меньшего объёма памяти и полностью умещается в кэше процессора. Чтобы ещё немного повысить производительность можно воспользоваться когерентностью соседних строк, грани видимые в одной строке, скорее всего, видимы и в соседних. Если рисовать сначала те грани, которые были видимы в предыдущей строке, а затем все остальные, то Z-буфер будет сразу заполняться наиболее близкими значениями глубины этих граней, а большинство пикселей остальных граней окажутся дальше, что позволит сократить количество операций рисования пикселей и модификации значений глубины в Z-буфере. [3]

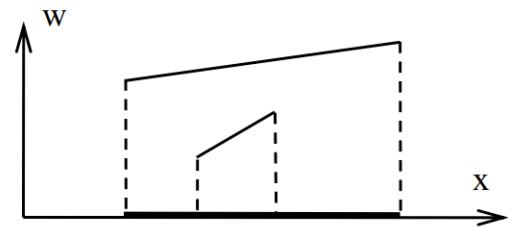
### 1.3.9.2 Алгоритм S-буфера

S-буфер – это буфер отрезков видимых наблюдателю (S – от segment, отрезок). Отрезки, полученные в результате пересечения граней плоскостью, поочерёдно заносятся в S-буфер. При этом очередной отрезок сначала сравнивается с отрезками буфера. Могут возникнуть следующие случаи:

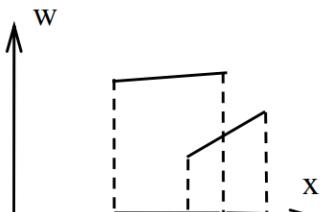
- проекции отрезков не пересекаются (см. рис. 22а), ни один из отрезков не заслоняет другого;
- новый отрезок может полностью заслонить отрезок буфера (см. рис. 22б), в этом случае заслонённый отрезок удаляется из буфера;
- новый отрезок может заслонить часть отрезка буфера (см. рис. 22в), в этом случае отрезок буфера делится на части и заслонённая часть удаляется из буфера;
- отрезок из буфера может полностью заслонить новый отрезок (см. рис. 22б), в этом случае новый отрезок не виден и не заносится в буфер;
- отрезок из буфера может заслонить часть нового отрезка (см. рис. 22в), в этом случае новый отрезок делится на части, заслонённая часть удаляется, а оставшиеся сравниваются с остальными отрезками буфера;
- отрезки пересекаются (см. рис. 22г), в этом случае оба отрезка делятся на части, заслонённые части удаляются, оставшиеся части отрезка буфера заносятся обратно в буфер, оставшиеся части нового отрезка сравниваются с остальными отрезками буфера.



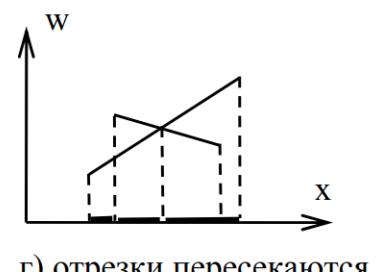
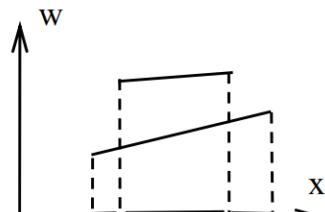
а) проекции не пересекаются



б) один отрезок заслоняет другой



в) один отрезок заслоняет часть другого

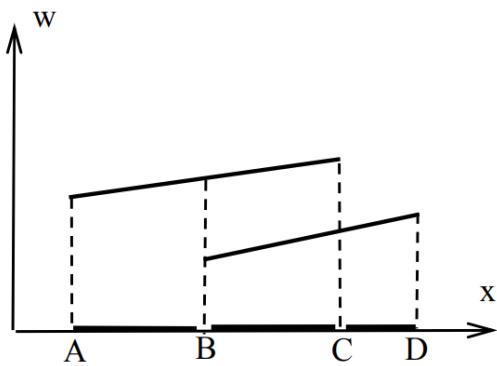


г) отрезки пересекаются

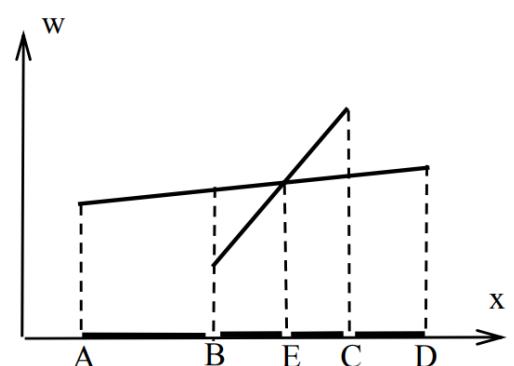
Рисунок 22 – Сравнение двух отрезков.

На рисунке 22 ось  $x$  – горизонтальная ось окна, ось  $w$  – параметр глубины, чем больше  $w$ , тем точка ближе к наблюдателю.

Определить, что проекции отрезков не пересекаются можно по условию, что максимальная координата  $x$  одного отрезка не превышает минимальную координату  $x$  другого. Когда проекции пересекаются, нужно определить, какие части отрезков заслонены (см. рис. 23). Если отсортировать точки концов отрезков ( $A, B, C, D$ ) вдоль оси  $x$ , то можно выделить три интервала ( $A-B, B-C, C-D$ ). На первом интервале видна часть отрезка, которому принадлежит точка  $A$ , на третьем интервале видна часть отрезка, которому принадлежит точка  $D$ .



а) один отрезок заслоняет часть другого



б) отрезки пересекаются

Рисунок 23 – Определение заслонённых частей отрезков.

Чтобы определить, какие части видимы на втором интервале, нужно рассчитать глубину точек отрезков на его концах. Для двух точек она известна, так как точки являются концами отрезков, для двух оставшихся можно рассчитать, используя уравнение прямой:

$$w = w_1 + (w_2 - w_1) \cdot \frac{x - x_1}{x_2 - x_1}, \quad (13)$$

где  $x_1, w_1, x_2, w_2$  – координаты двух концов отрезка,  $x, w$  – координаты точки пересечения. Если глубина обеих точек одного отрезка больше, чем у точек второго, то второй отрезок заслонён первым на этом интервале (см. рис. 23а). Если в начале интервала больше глубина одного отрезка, а в конце – другого, то отрезки пересекаются (см. рис. 23б). В этом случае необходимо найти точку пересечения отрезков, например, приравняв вышеприведённое уравнение прямой для прямых обоих отрезков. Интервал  $B-C$  поделится точкой пересечения  $E$  на два интервала ( $B-E, E-C$ ). На первом будет виден отрезок, глубина которого в точке  $B$  больше, чем у другого, на втором – отрезок, глубина которого в точке  $C$  больше, чем у другого. [3]

### 1.3.10 Выводы

Таблица 1 представляет собой сравнительный анализ алгоритмов удаления невидимых линий по основным критериям, важным для задачи создания и редактирования 3D-сцен.

### Обозначения:

$n$  – количество граней объектов сцены;

$C$  – количество пикселей окна;

$S$  – количество строк окна.

Алгоритм	Необходимость в сортировке	Временная сложность	Возможность реализации оптических эффектов
Плавающий горизонт	да	$O(n \cdot \log n)$	да
Робертса	нет	$O(n^2)$	нет
Варнака	нет	$O(n \cdot \log n)$	нет
Вейлера-Азертона	да	$O(n^2)$	да
Художника	да	$O(n \cdot \log n)$	да
Трассировки лучей	нет	$O(C \cdot n)$	да
Z-буфера	нет	$\mathbf{O(n)}$	нет
Построчного сканирования	нет	$O(n \cdot S)$	нет

Таблица 1 – Сравнение алгоритмов удаления невидимых линий по критериям

Алгоритм Z-буфера выбран для задачи построения цифровой модели помещения по следующим причинам:

1. Точность: Z-буфер позволяет точно определять видимость объектов, так как каждый пиксель имеет соответствующую глубину, что важно для сцены, состоящей из параллелепипедов.
2. Скорость: Несмотря на то, что Z-буфер может требовать значительных вычислительных ресурсов для сложных сцен, он остаётся достаточно быстрым для большинства случаев. Однако для сложных сцен с высоким разрешением может потребоваться больше времени на перерасчёт и больше памяти из-за хранения матрицы глубины для каждого пикселя.
3. Память: Хотя Z-буфер требует много памяти для хранения данных о глубине, это компенсируется его простотой и высокой точностью. Однако в интерактивных приложениях, где сцена изменяется динамически, перерасчёт для каждого пикселя может быть медленным.

#### 1.4. Алгоритмы отрисовки теней

##### 1.4.1 Введение

Тени являются неотъемлемой частью реалистичных изображений в компьютерной графике. Они помогают передать пространственные соотношения между объектами сцены, добавляют глубину и реализм визуализации. Существует множество методов

отрисовки теней, которые варьируются по сложности и требуемым вычислительным ресурсам. В связи с выбором алгоритма Z-буфера для удаления невидимых линий и плоскостей, удобно использовать его и для алгоритма построения теней. Такой подход позволяет сохранить простоту структуры программы, исключить сложности при интеграции двух разных методов и, как следствие, сократить время, необходимое на отладку алгоритма.

#### 1.4.2 Принципы работы Z-буфера

Алгоритм Z-буфера основывается на хранении информации о глубине объектов сцены для каждого пикселя экрана. При расчёте теней алгоритм проходит два этапа:

1. Первый проход: Сцена анализируется с позиции источника света. Вычисляются точки, видимые со стороны источника, и их глубины заносятся в теневой Z-буфер.
2. Второй проход: Сцена визуализируется с позиции наблюдателя. Для каждого пикселя проверяется, находится ли он в тени, путём сравнения его координат с данными теневого Z-буфера.

Преимущества Z-буфера для отрисовки теней:

- Универсальность: метод подходит для сцен любой сложности, включая динамические сцены с движущимися объектами и источниками света.
- Реализация в реальном времени: алгоритм хорошо масштабируется на современных графических процессорах, обеспечивая приемлемую скорость работы.
- Простота интеграции: Z-буфер легко адаптируется для построения теней, поскольку он изначально используется для удаления невидимых поверхностей.

Ограничения метода:

- Артефакты на границах теней: из-за дискретизации возможны погрешности, такие как акне (z-fighting) или ступенчатые края.
- Сложность мягких теней: алгоритм изначально рассчитан на жёсткие тени. Реализация эффектов мягкости требует дополнительных вычислений.

#### 1.4.3 Выводы

Алгоритм Z-буфера является эффективным выбором для построения теней в условиях, когда требуется высокая скорость и приемлемое качество визуализации. Несмотря на некоторые ограничения, он остаётся стандартным методом, широко применяемым в компьютерной графике благодаря простоте и универсальности. [10]

## 1.5. Алгоритмы закраски

В компьютерной графике широко используются алгоритмы закраски для наложения освещения на объекты сцены. В этом разделе будут рассмотрены три наиболее популярные модели: простая закраска (метод гранения), закраска по Гуро (Gouraud shading) и закраска по Фонгу (Phong shading).

### 1.5.1 Простая закраска (метод гранения)

Это наиболее простой и требующий наименьших вычислительных ресурсов метод закраски поверхности. Цвет всей поверхности рассчитывается согласно закону Ламберта, который определяется следующим образом:

$$I = I_L \cdot k_d \cdot \cos \theta, \quad (14)$$

где:

- $I$  – результирующая интенсивность света;
- $I_L$  – интенсивность источника света;
- $k_d$  – коэффициент диффузного отражения материала;
- $\theta$  – угол между нормалью к поверхности и направлением на источник света.



Рисунок 24 – Изображение, полученное методом гранения.

В результате, при малом количестве граней, многогранник выглядит именно как многогранник, так как цвета отдельных граней существенно отличаются. Тем не менее, создаётся эффект объёмности изображения.

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. [11]

### 1.5.2 Закраска по Гуро

Модель закраски по Гуро предполагает линейную интерполяцию цвета между вершинами полигона. Это требует, чтобы нормали были заданы в вершинах, так как цвет рассчитывается на основе нормалей. Преимущества и недостатки модели:

- Преимущества:
  - Высокая скорость работы: дорогостоящие расчёты освещения производятся только в вершинах.
  - Легко оптимизируется благодаря линейной интерполяции.
- Недостатки:
  - Потеря бликов на поверхности при низкой детализации модели.
  - Размытые отражения, особенно заметные на крупных полигонах.

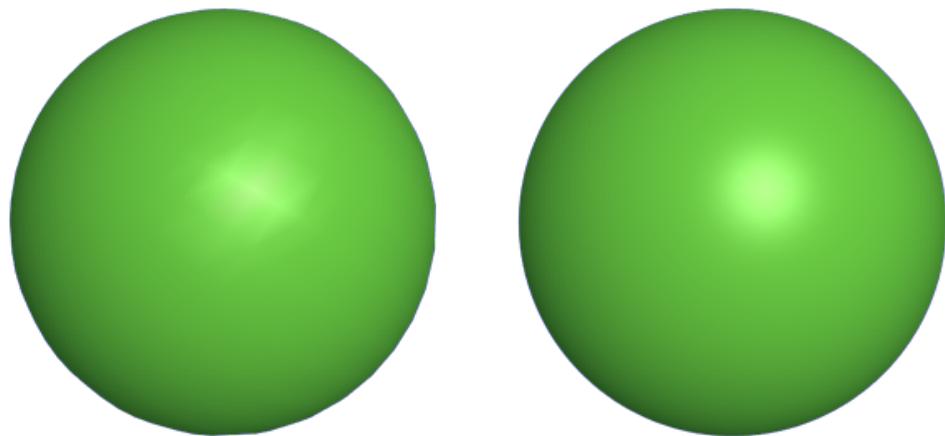


Рисунок 25 – Сфера с закраской по Гуро: около 2000 (слева) и 32000 треугольников (справа).

### 1.5.3 Закраска по Фонгу

Закраска по Фонгу работает иначе: между вершинами интерполируется не цвет, а нормаль, а освещение рассчитывается для каждого пикселя. Это позволяет добиться более высокой визуальной точности.

- Преимущества:
  - Качественное изображение без смазывания бликов.
  - Высокая степень реализма благодаря попиксельному расчёту освещения.
- Недостатки:
  - Высокие вычислительные затраты, особенно для сцен с высоким разрешением.

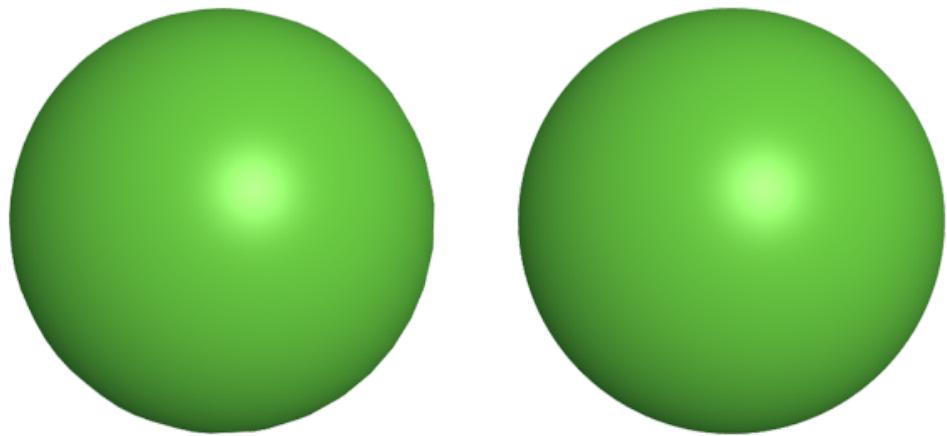


Рисунок 26 – Сфера с закраской по Фонгу: около 2000 (слева) и 32000 треугольников (справа).

#### 1.5.4 Выводы

Для задачи построения 3D-сцен помещений с возможностью добавления, удаления, перемещения, поворота и масштабирования объектов-параллелепипедов, а также сохранения и загрузки моделей, наиболее рациональным выбором является использование простой закраски (метод гранения). Этот метод был выбран по следующим причинам:

##### 1. Низкая вычислительная сложность:

Простой метод закраски не требует сложных вычислений, что обеспечивает высокую производительность программы, особенно при редактировании сцены в реальном времени.

##### 2. Соответствие задаче:

Плоские поверхности (стены, окна, двери) легко визуализируются с помощью метода гранения, создавая эффект объёмности без избыточной детализации.

##### 3. Простота реализации:

Метод гранения легко интегрируется в систему, оставляя больше времени и ресурсов для разработки основных функций работы с объектами и камерой.

#### 4. Достаточное качество изображения:

Визуализация, созданная методом гранения, достаточно информативна для проектирования и редактирования помещений, где точность освещения не является ключевым требованием.

Таким образом, метод гранения обеспечивает оптимальный баланс между качеством изображения и скоростью работы, что делает его предпочтительным для данной задачи.

#### 1.6. Выводы

В ходе работы проведён анализ алгоритмов удаления невидимых линий, построения теней и методов затенения для применения в задаче создания цифровой модели помещения и её визуализации. На основе сравнительного анализа выбран алгоритм Z-буфера, который обеспечивает точность и универсальность, подходящую для работы с параллелепипедами. Этот же алгоритм адаптирован для построения теней, что упрощает реализацию и интеграцию. Для визуализации выбрана модель простой закраски, обеспечивающая достаточный уровень реализма для задачи отрисовки многогранников.

Итогом работы стали изученные выбранные алгоритмы, которые позволяют создавать точную и визуально правдоподобную цифровую модель помещения с возможностью оптимизации для более сложных сцен.

## 2. Конструкторская часть

### 2.1. Требования к программному обеспечению

На основании введения и поставленных задач сформулированы следующие требования к разрабатываемому программному обеспечению для построения 3D сцен помещений:

#### 2.1.1 Функциональные требования

##### 1. Моделирование помещений и объектов:

- Возможность добавления базовых 3D объектов: стен, окон, дверей.
- Поддержка настройки параметров объектов (длина, ширина, высота, аналогично для отверстия).
- Функции перемещения, изменения параметров, вращения и удаления объектов на сцене.
- Обеспечение точного размещения объектов с проверкой пересечений и соблюдением заданных границ.

##### 2. Управление камерой:

- Предоставление функций управления камерой для осмотра сцены:
  - Вращение вокруг всех трёх осей.
  - Изменение масштаба для просмотра деталей и общей картины.

##### 3. Работа со сценами:

- Сохранение текущей сцены в файл.
- Загрузка ранее сохранённых сцен для продолжения работы.

##### 4. Пользовательский интерфейс:

- Разработка интуитивно понятного интерфейса для работы с объектами и сценой.
- Наличие панелей инструментов для выбора объектов, редактирования их параметров и управления камерой.
- Поддержка визуализации сцены в реальном времени.

##### 5. Тестирование:

- Проверка корректности работы с объектами, сценой и камерой в различных условиях.
- Тестирование функций сохранения и загрузки сцен.

### 2.1.2 Нефункциональные требования

#### 1. Платформенная независимость:

- Программа должна поддерживать работу на основных операционных системах (Windows, macOS, Linux).

#### 2. Простота и удобство использования:

- Дружелюбный интерфейс, не требующий глубоких технических знаний.

#### 3. Надёжность:

- Минимизация сбоев программы при работе с большими сценами.
- Обработка ошибок и предупреждений при неправильных действиях пользователя.

#### 4. Расширяемость:

- Возможность добавления новых типов объектов и функций без значительных изменений в структуре программы.

#### 5. Безопасность данных:

- Сохранение и загрузка файлов без потери данных и повреждения структуры сцены.

### 2.1.3 Ожидаемые результаты

Программное обеспечение должно позволить быстро и эффективно моделировать 3D сцены помещений, обеспечивая интерактивное добавление и редактирование объектов. Оно станет полезным инструментом для анализа, планирования и обучения в области безопасности и оперативного реагирования.

## 2.2. Общий алгоритм решения поставленной задачи

#### 1. Определение параметров сцены:

- Установить размеры сцены, на которой будут размещаться объекты.

#### 2. Размещение элементов сцены:

- Разместить необходимые объекты (стены, окна, двери) на сцене с заданием их параметров.

#### 3. Визуализация сцены с тенями:

- На основе текущего положения наблюдателя использовать модифицированный алгоритм, основанный на Z-буфере, для расчёта теней, падающих от объектов сцены.
- Выполнить рендеринг сцены с учётом теней и освещения.

### 2.3. Алгоритм Z-буфера удаления невидимых линий

Алгоритм Z-буфера удаления невидимых линий на псевдокоде:

1. Заполнить буфер кадра фоновым значением интенсивности или цвета.
2. Заполнить Z-буфер минимальным значением глубины  $z$ .
3. Преобразовать каждый многоугольник в растровую форму в произвольном порядке.
4. Для каждого пикселя  $(x,y)$  в многоугольнике вычислить его глубину  $z(x,y)$ .
5. Сравнить вычисленную глубину  $z(x,y)$  со значением в Z-буфере в этой же позиции  $Z_{\text{буфер}}(x,y)$ .
6. Запись в буфер кадра:
  - Если  $z(x,y) > Z_{\text{буфер}}(x,y)$ , то записать атрибуты многоугольника (интенсивность, цвет и т. п.) в буфер кадра и заменить значение в Z-буфере на  $z(x,y)$ .
  - В противном случае никаких действий не производить. [1]

Блок-схема алгоритма Z-буфера удаления невидимых линий представлена на рисунке 27.

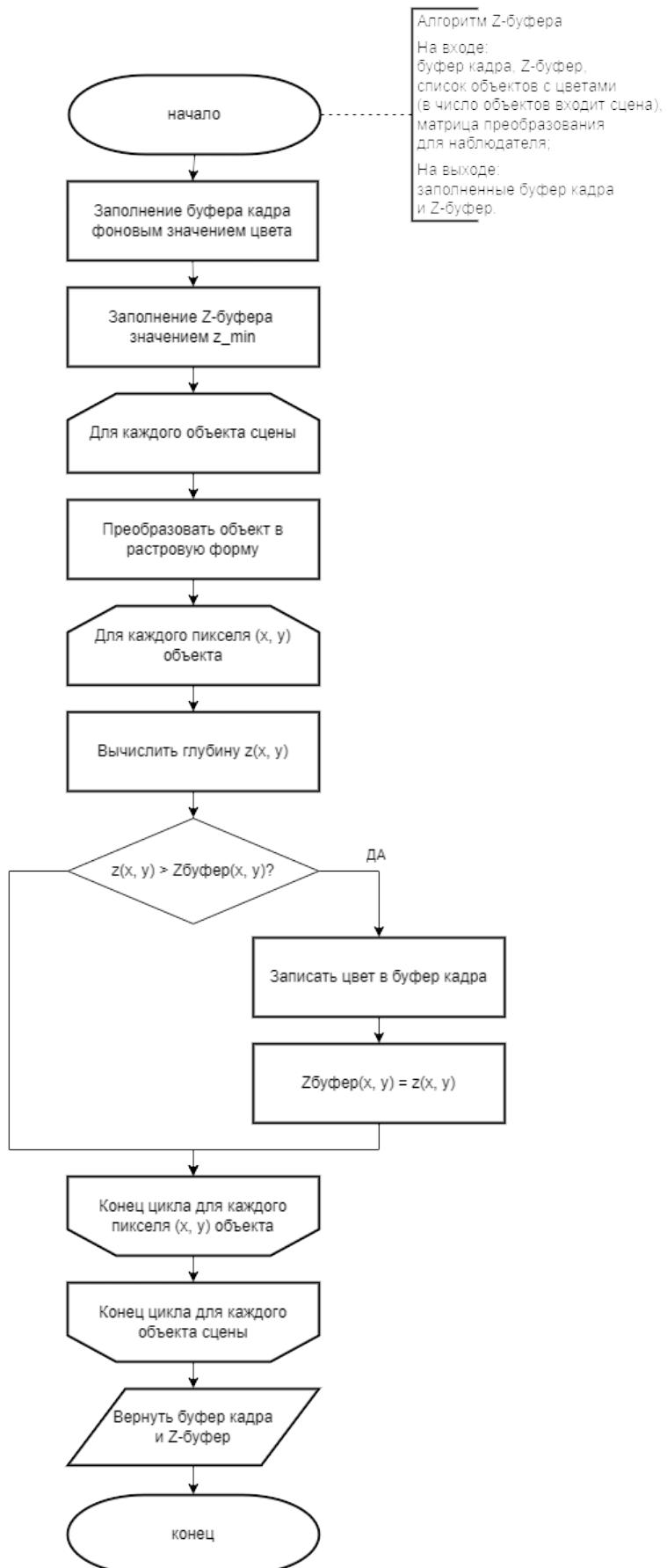


Рисунок 27 – Блок-схема алгоритма Z-буфера удаления невидимых линий.

## 2.4. Алгоритм Z-буфера с отрисовкой теней

Алгоритм Z-буфера с отрисовкой теней на псевдокоде:

1. Выполнить расчет Z-буфера для основной сцены:
  - (a) Заполнить буфер кадра фоновым значением интенсивности или цвета.
  - (b) Заполнить Z-буфер минимальным значением глубины  $z$ .
  - (c) Преобразовать каждый многоугольник в растровую форму в произвольном порядке.
  - (d) Для каждого пикселя  $(x, y)$  в многоугольнике:
    - Вычислить глубину пикселя  $z(x, y)$ .
    - Сравнить  $z(x, y)$  с текущим значением  $Z_{\text{буфер}}(x, y)$ .
    - Если  $z(x, y) > Z_{\text{буфер}}(x, y)$ , записать атрибуты пикселя в буфер кадра и обновить  $Z_{\text{буфер}}(x, y)$ .
2. Выполнить расчет Z-буфера для вида из источника света:
  - (a) Заполнить буфер кадра фоновым значением.
  - (b) Заполнить Z-буфер минимальным значением глубины  $z$ .
  - (c) Преобразовать каждый многоугольник в растровую форму относительно вида из источника света.
  - (d) Для каждого пикселя вычислить его глубину  $z(x, y)$  и записать в Z-буфер для создания карты теней.
3. Наложить матрицу теней на основную сцену:
  - (a) Для каждого пикселя  $(x, y)$  в Z-буфере основной сцены:
    - Извлечь глубину  $z(x, y)$  и преобразовать координаты  $(x, y, z)$  в координаты вида из источника света  $(x', y', z')$ .
    - Если  $x', y'$  находятся в пределах карты теней, сравнить  $z(x, y)$  с глубиной из карты теней  $z_{\text{light}}(x', y')$ .
    - Если  $z(x, y) > z_{\text{light}}(x', y') + \text{EPS}$ , уменьшить яркость пикселя или применить цвет тени.

Блок-схема алгоритма Z-буфера с отрисовкой теней представлена на рисунке 28.

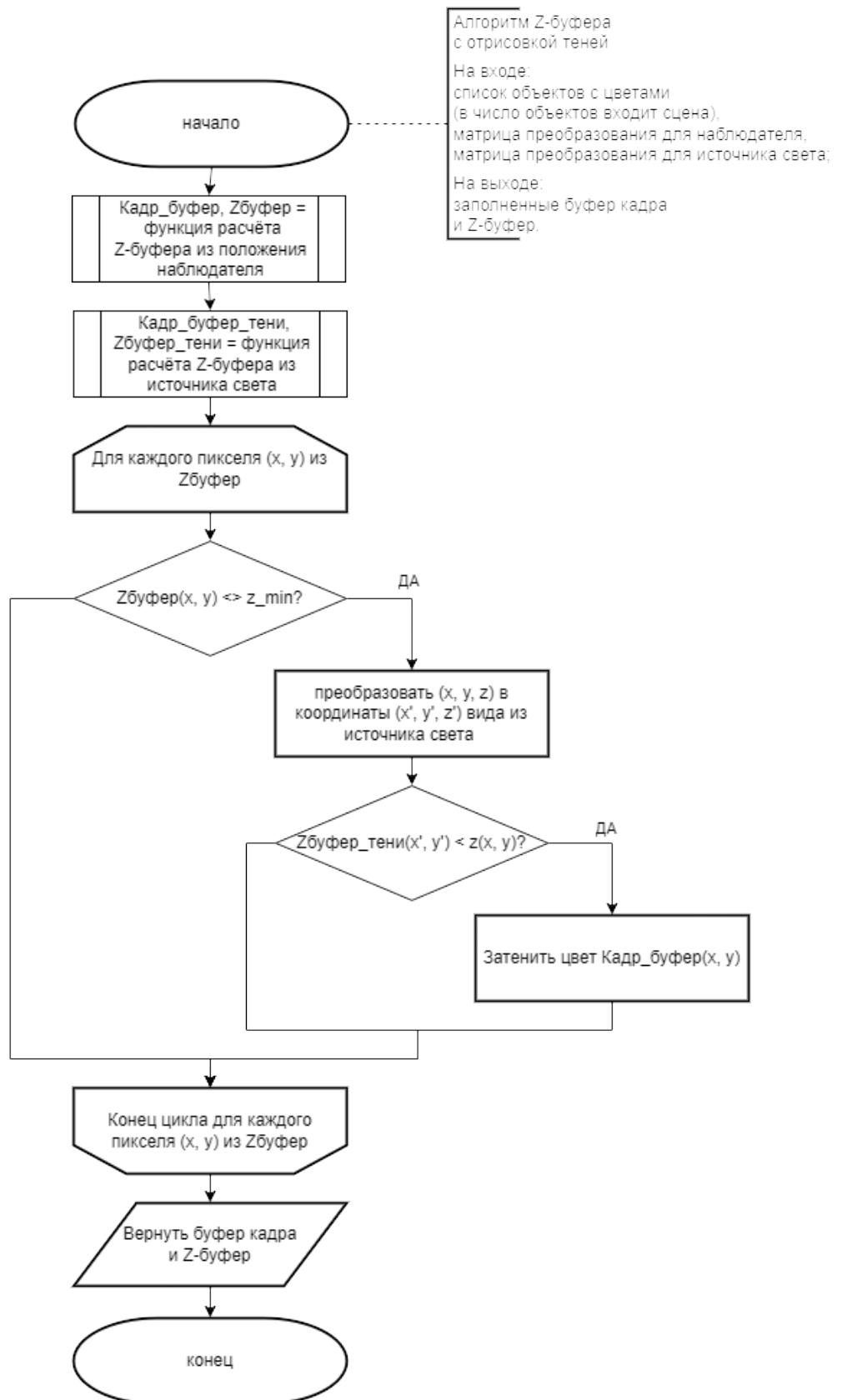


Рисунок 28 – Блок-схема алгоритма Z-буфера с отрисовкой теней.

## 2.5. Представление объектов в программном обеспечении

### 1. Точка трёхмерного пространства

Представлена координатами по осям  $x$ ,  $y$ ,  $z$ .

### 2. Стена

Задаётся как параллелепипед, параметры которого включают длину, высоту, ширину и координаты начальной точки.

### 3. Окно

Подвид стены, представлено параллелепипедом с параметрами длины, высоты, ширины и расположения (координаты начальной точки).

### 4. Дверь

Аналогично окну, задаётся параллелепипедом с уникальными параметрами длины, высоты, ширины и позиции в пространстве.

### 5. Сцена

Представлена набором плиток (квадратных полигонов), каждая из которых определяется координатами  $(x, y, z)$ , где  $z = 0$  для пола.

### 6. Камера

Описывается матрицей преобразования, включающей параметры положения камеры (координаты) и направления взгляда (углы поворота по осям  $x$ ,  $y$ ,  $z$ ).

### 7. Источник света

Представлен аналогично камере, с использованием матрицы преобразования, включающей параметры положения источника света (углы поворота по осям  $x$ ,  $y$ ,  $z$ ).

## 2.6. Выводы

В рамках конструкторской части были сформулированы основные требования к программному обеспечению для моделирования 3D сцен помещений. Также были рассмотрены два ключевых алгоритма рендеринга: алгоритм Z-буфера для удаления невидимых линий и его расширение с отрисовкой теней. Оба алгоритма направлены на создание более реалистичной визуализации сцен с учетом взаимодействия объектов и источника света. Представление объектов в программном обеспечении включает точное описание 3D объектов (стены, окна, двери) и их параметров, а также обработку сцены, камеры и источника света с использованием матрицы преобразования.

Таким образом, работа направлена на создание системы, которая будет эффективно решать задачи моделирования помещений с высокой степенью взаимодействия и визуализации, обеспечивая удобство работы и надежность при взаимодействии с пользователем.

### 3. Технологическая часть

#### 3.1. Выбор средств реализации ПО

Для разработки программы выбран язык программирования Python. Данный выбор обусловлен следующими причинами:

##### 1. Богатая экосистема инструментов и библиотек

Python поддерживает необходимые для проекта библиотеки, такие как:

- Tkinter для создания графического пользовательского интерфейса (GUI).
- NumPy для выполнения математических операций, включая преобразования координат и вычисления.
- math для работы с математическими функциями.
- ProcessPoolExecutor из модуля concurrent.futures для организации мультипроцессинга.
- pytest для написания модульных тестов, что помогает автоматизировать проверку функционала программы и выявлять ошибки.

##### 2. Использование наработок из курса «Компьютерная графика»

Python и Tkinter уже применялись в лабораторных работах по курсу «Компьютерная графика», где была отработана инфраструктура для создания приложений. Это позволило заимствовать проверенные решения, такие как:

- Интерфейсные элементы и их взаимодействие.
- Фрагменты кода для работы с объектами сцены.
- Систему тестирования и сборки, что ускорило разработку и снизило риск ошибок.

##### 3. Поддержка объектно-ориентированного программирования (ООП)

Python предоставляет удобные механизмы для работы с ООП, что позволило разделить программу на независимые модули и классы (например, объекты сцены, камера, управление интерфейсом).

##### 4. Платформенная независимость

Программы на Python работают на различных операционных системах (Windows, macOS, Linux), что делает приложение доступным для широкого круга пользователей.

##### 5. Простота и читаемость кода

Python отличается лаконичным синтаксисом и высокой читаемостью кода, что ускоряет процесс разработки и тестирования программы. Это особенно важно для курсового проекта, где требуется не только реализовать функционал, но и ясно представить код и структуру программы.

### 3.2. Структура программы

Программа представляет собой объектно-ориентированную архитектуру, предназначенную для работы с элементами сцены и пространственными объектами, такими как стены, двери и окна. Взаимодействие между классами организовано вокруг обработки и отображения трёхмерной модели помещения. Структура классов разработанного программного обеспечения представлена на рисунке 29.

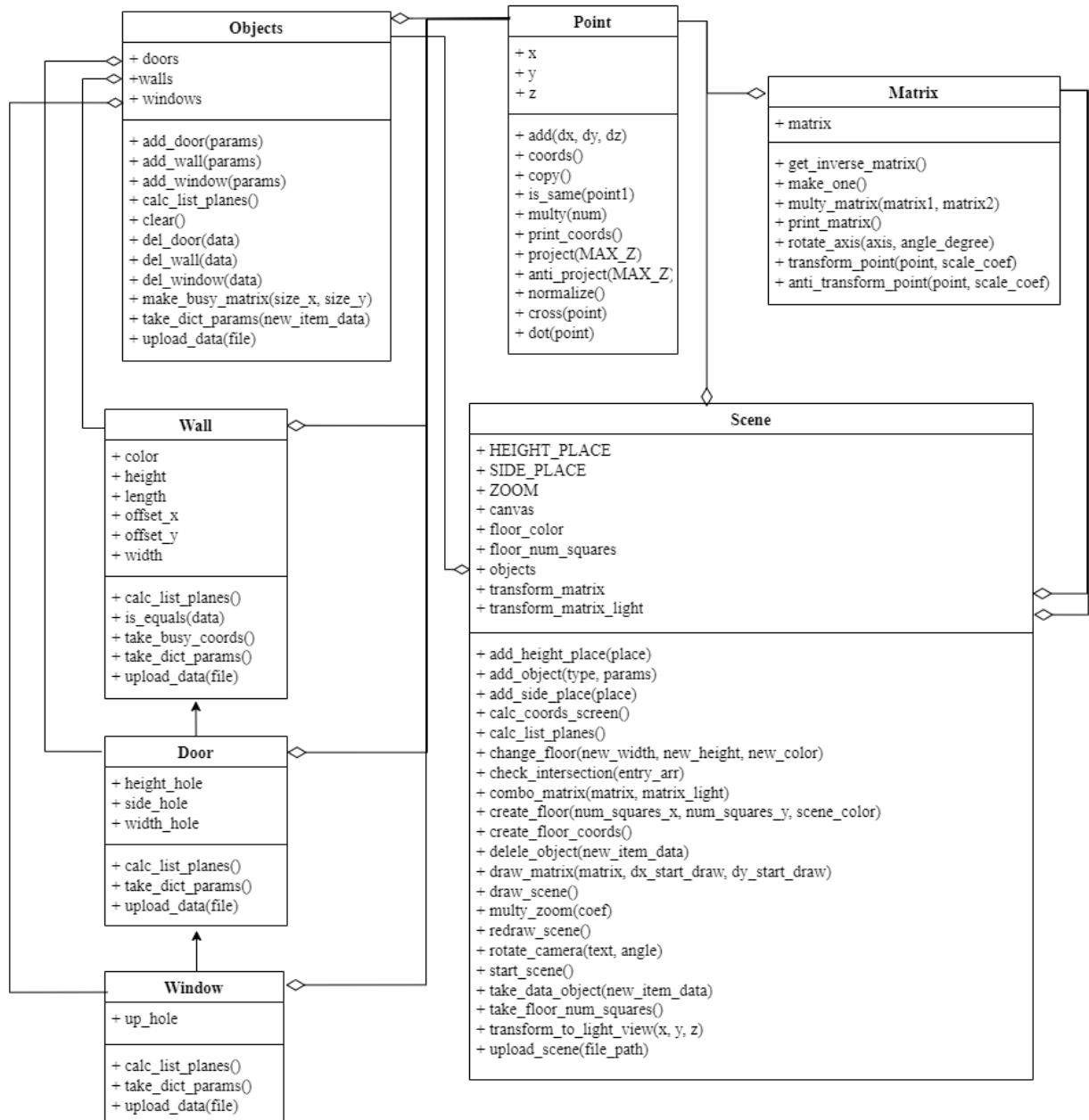


Рисунок 29 – Структура классов разработанного программного обеспечения.

#### 3.2.1 Классы и их роли

Основные классы и их функции можно разбить на несколько ключевых компонентов.

### 3.2.1.1 Пространственные элементы

Эти классы представляют объекты сцены (стены, двери, окна) и содержат свойства, необходимые для их описания, а также методы для вычисления их геометрии.

- Wall:
  - Хранит такие параметры, как цвет, длина, высота, ширина и смещение.
  - Методы:
    - \* `calc_list_planes()`: вычисляет геометрические плоскости объекта.
    - \* `is_equals(data)`: проверяет эквивалентность с другим объектом.
    - \* `upload_data(file)`: загружает параметры объекта из файла.
- Door (наследует Wall):
  - Добавляет дополнительные свойства, такие как высота, ширина и сторона, где расположено отверстие.
- Window (наследует Wall):
  - Включает параметр `up_hole`, связанный с высотой окна.

### 3.2.1.2 Работа с геометрическими преобразованиями

- Matrix:
  - Хранит матрицы преобразований и предоставляет инструменты для выполнения операций с ними.
  - Методы:
    - \* `get_inverse_matrix()`: получение обратной матрицы.
    - \* `mult_matrix()`: умножение двух матриц.
    - \* `rotate_axis()`: поворот объектов вокруг оси.
    - \* `transform_point()`: трансформация координат точки.
- Point:
  - Представляет точку в пространстве с координатами  $(x, y, z)$ .
  - Методы:
    - \* `add(dx, dy, dz)`: добавляет смещение к точке.
    - \* `multiply(num)`: умножает координаты на заданное число.
    - \* `project(MAX_Z)`: выполняет проекцию точки на плоскость.

### 3.2.1.3 Управление коллекцией объектов

Objects:

- Хранит списки объектов сцены (двери, стены, окна).
- Методы:
  - `add_door(params)`, `add_wall(params)`, `add_window(params)`: добавление объектов.
  - `calc_list_planes()`: вычисление плоскостей для всех объектов.
  - `clear()`: очистка списка объектов.
  - `check_intersection(entry_arr)`: проверка пересечений между объектами.

### 3.2.1.4 Управление сценой

Scene:

- Основной класс для управления сценой и её параметрами.
- Свойства:
  - Константы, такие как размеры сцены (`HEIGHT_PLACE`, `SIDE_PLACE`).
  - Цвет пола, параметры света, матрица трансформации и объекты сцены.
- Методы:
  - `add_object(type, params)`: добавление нового объекта в сцену.
  - `rotate_camera(text, angle)`: вращение камеры.
  - `draw_scene()`: отрисовка сцены.
  - `check_intersection(entry_arr)`: проверка пересечений объектов сцены.

## 3.2.2 Логика работы программы

### 1. Инициализация сцены

Создаётся объект `Scene`, в который добавляются стены, двери и окна через методы `add_object` или `add_wall`.

### 2. Работа с геометрией

Методы классов `Wall`, `Door` и `Window` используются для вычисления параметров объектов, а класс `Matrix` помогает выполнять геометрические преобразования.

### 3. Проверка пересечений

Программа может проверять пересечения между объектами сцены с помощью метода `check_intersection`.

#### 4. Отрисовка

Класс Scene отвечает за создание визуализации сцены с использованием геометрических данных.

#### 3.3. Интерфейс программы

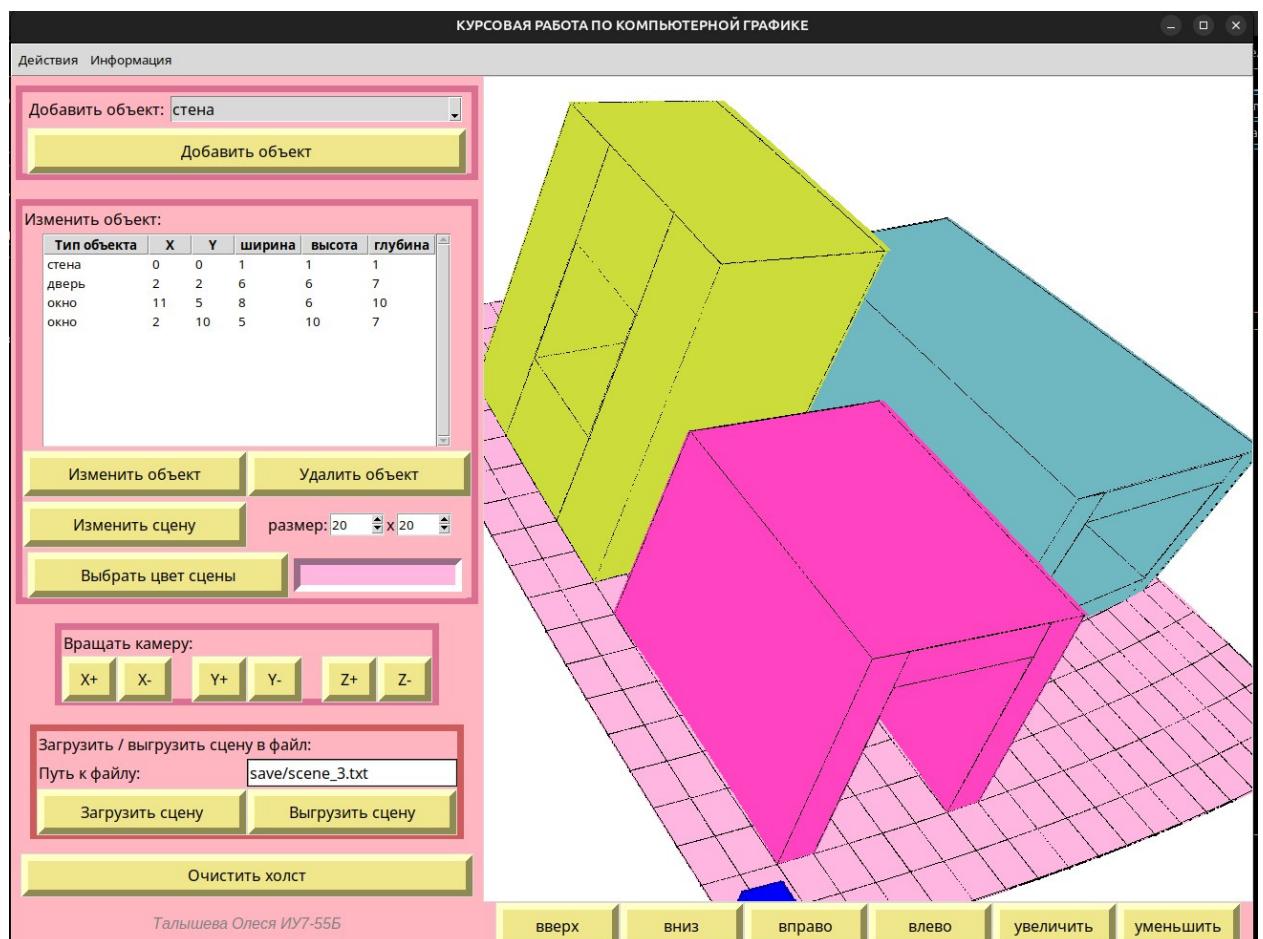


Рисунок 30 – Интерфейс главного окна программного обеспечения.

Программное обеспечение предлагает интуитивно понятный графический интерфейс для работы с 3D-сценами. Основной экран разделён на несколько функциональных областей, каждая из которых отвечает за выполнение конкретных задач (см. рисунок 30).

В верхней части интерфейса находится блок для добавления объектов. Пользователь может выбрать тип объекта из выпадающего списка, например, "Окно" или "Стена" после чего нажать кнопку "Добавить объект чтобы добавить его в сцену (см. рисунок 31). Все добавленные объекты отображаются в таблице в центральной части окна, где указаны их основные параметры: тип, координаты, размеры и глубина.



а) интерфейс окна добавления стены

б) интерфейс окна добавления двери

в) интерфейс окна добавления окна

Рисунок 31 – Интерфейс добавления объектов программы (а - стена, б - дверь, в - окно.

Для изменения параметров уже существующих объектов предусмотрены кнопки "Изменить объект" и "Удалить объект". Это позволяет либо обновить свойства выбранного объекта, либо удалить его из сцены.

Кроме того, пользователь может настроить параметры сцены, задав её размер через соответствующие поля ввода и изменив цвет с помощью кнопки "Выбрать цвет сцены". Управление камерой также доступно — можно задать угол вращения вокруг выбранной оси (X, Y или Z).

Для сохранения или загрузки сцен предусмотрен функционал работы с файлами. Пользователь может указать путь к файлу и воспользоваться кнопками "Загрузить сцену" или "Выгрузить сцену чтобы сохранить текущую сцену или восстановить её из файла. Также имеются кнопки для перемещения камеры вверх, вниз, влево или вправо, а также для изменения масштаба (увеличение или уменьшение).

При добавлении новых объектов, таких как стена или дверь, открывается отдельное окно диалога. Например, при добавлении окна пользователь может указать его размеры, положение в сцене и параметры отверстия (ширина, высота и подъём).

Есть возможность выбрать цвет объекта и, подтвердив настройки, добавить его в сцену. Для стен доступны аналогичные параметры, но без настройки отверстий.

Таким образом, интерфейс приложения предоставляет полный набор инструментов для создания и управления 3D-сценами, сохраняя удобство использования благодаря логично организованной структуре и понятным элементам управления.

### 3.4. Выводы

В данном разделе обоснован выбор языка программирования и инструментов, использованных для реализации проекта, проведён анализ структуры программы, включая назначение и взаимодействие её основных классов. Также было подробно рассмотрено устройство пользовательского интерфейса, включая функциональные возможности и элементы управления приложением.

## 4. Исследовательская часть

В данном разделе буду рассмотрены результаты работы программы на разных входных данных, а также проведены сравнения параллельной и последовательной работы программы и ускорения, связанного с распараллеливанием.

### 4.1. Примеры работы программы

На рисунке 32 представлена работа программы на примере комнаты из двух окон, двери и 4 стенах при углах наклона сцены 0 градусов по всем осям.

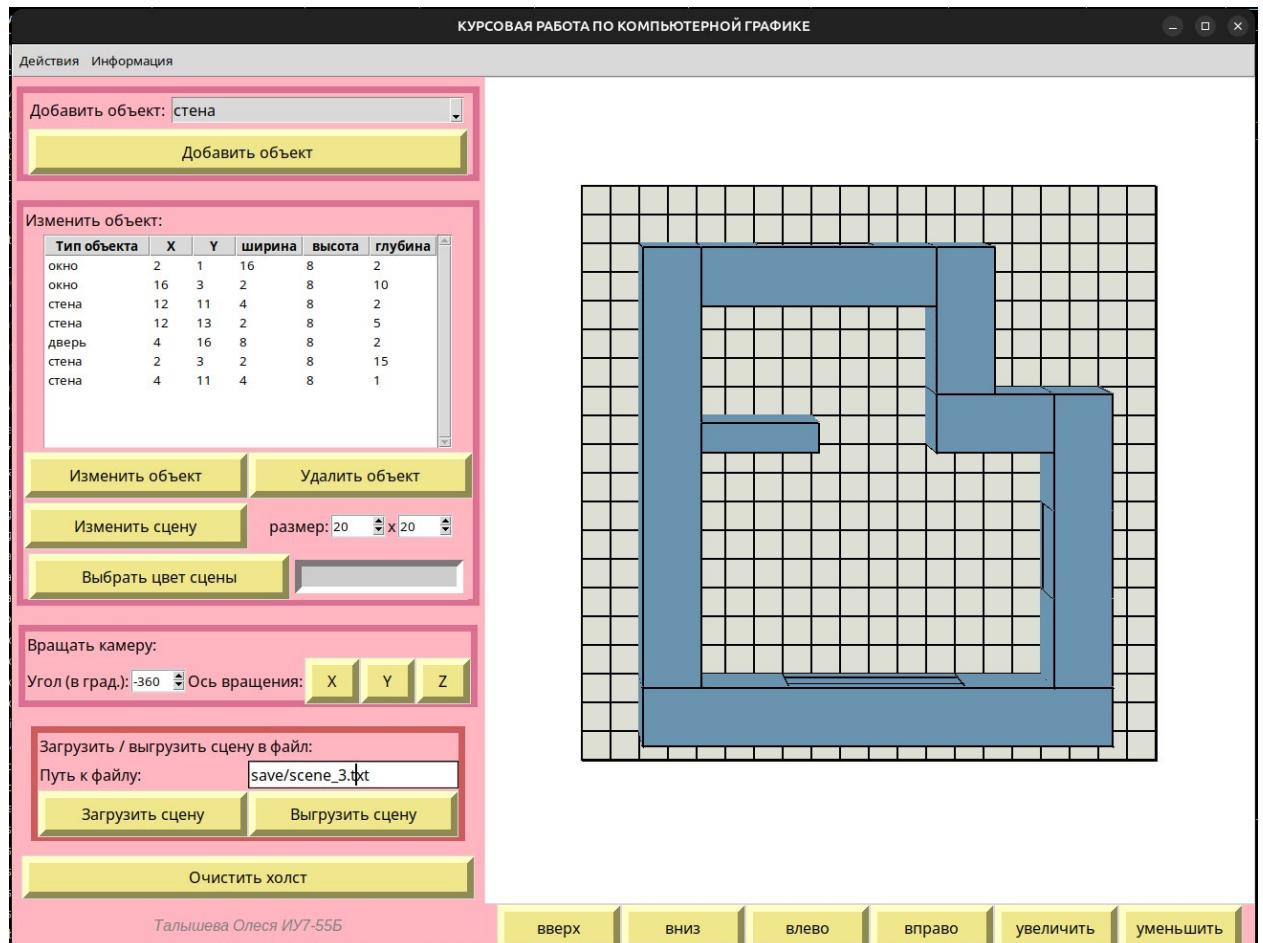


Рисунок 32 – Пример комнаты при углах наклона сцены 0 по всем осям.

На рисунке 33 представлена работа программы при той же введённой комнате при углах наклона сцены 60, -20 и 0 градусов по осям x, y и z соответственно.

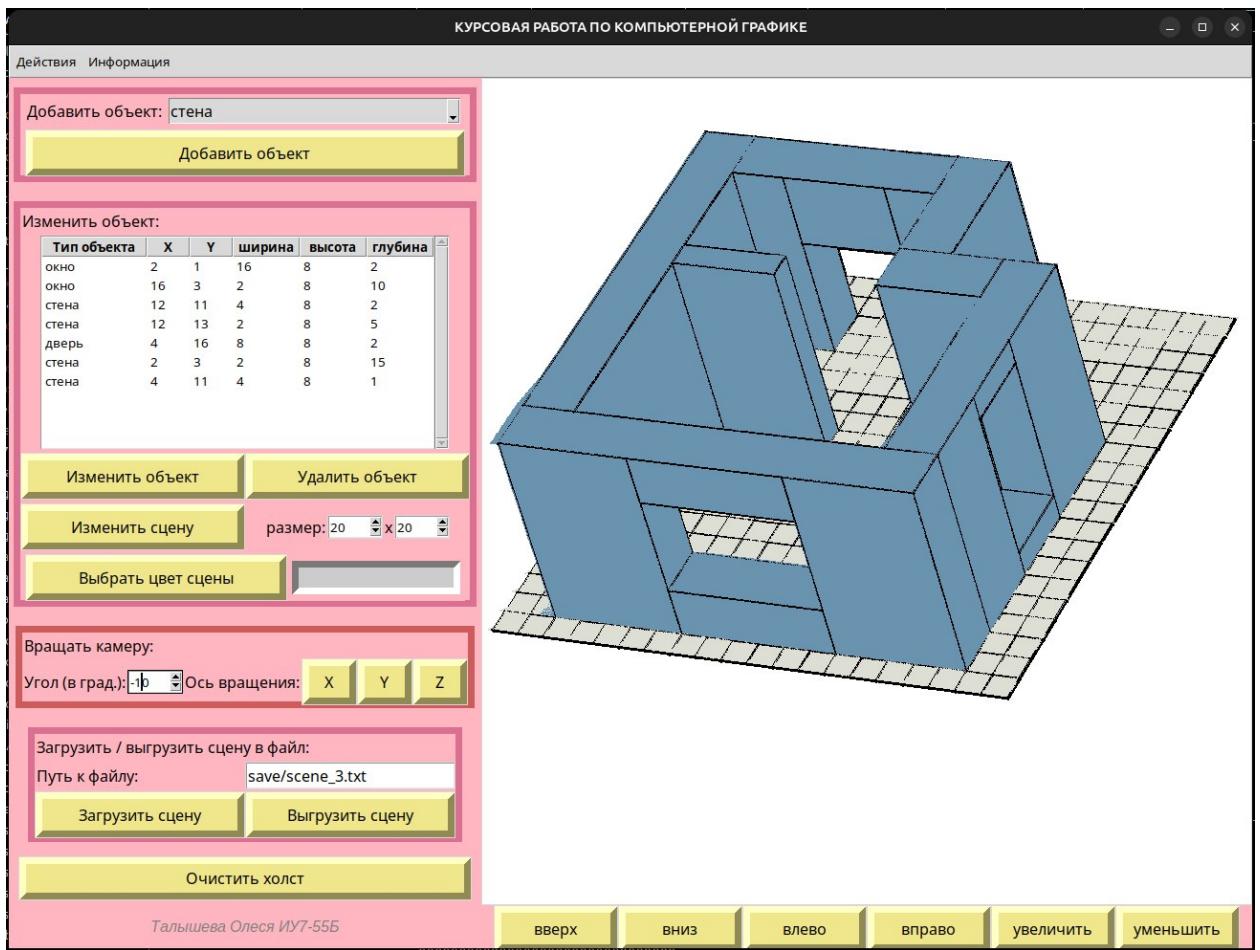


Рисунок 33 – Пример комнаты при углах наклона сцены 60, -20 и 0 градусов по осям x, y и z соответственно.

На рисунке 34 представлена работа программы при тех же введённых данных при углах наклона сцены 50, 50 и 180 градусов по осям x, y и z соответственно.

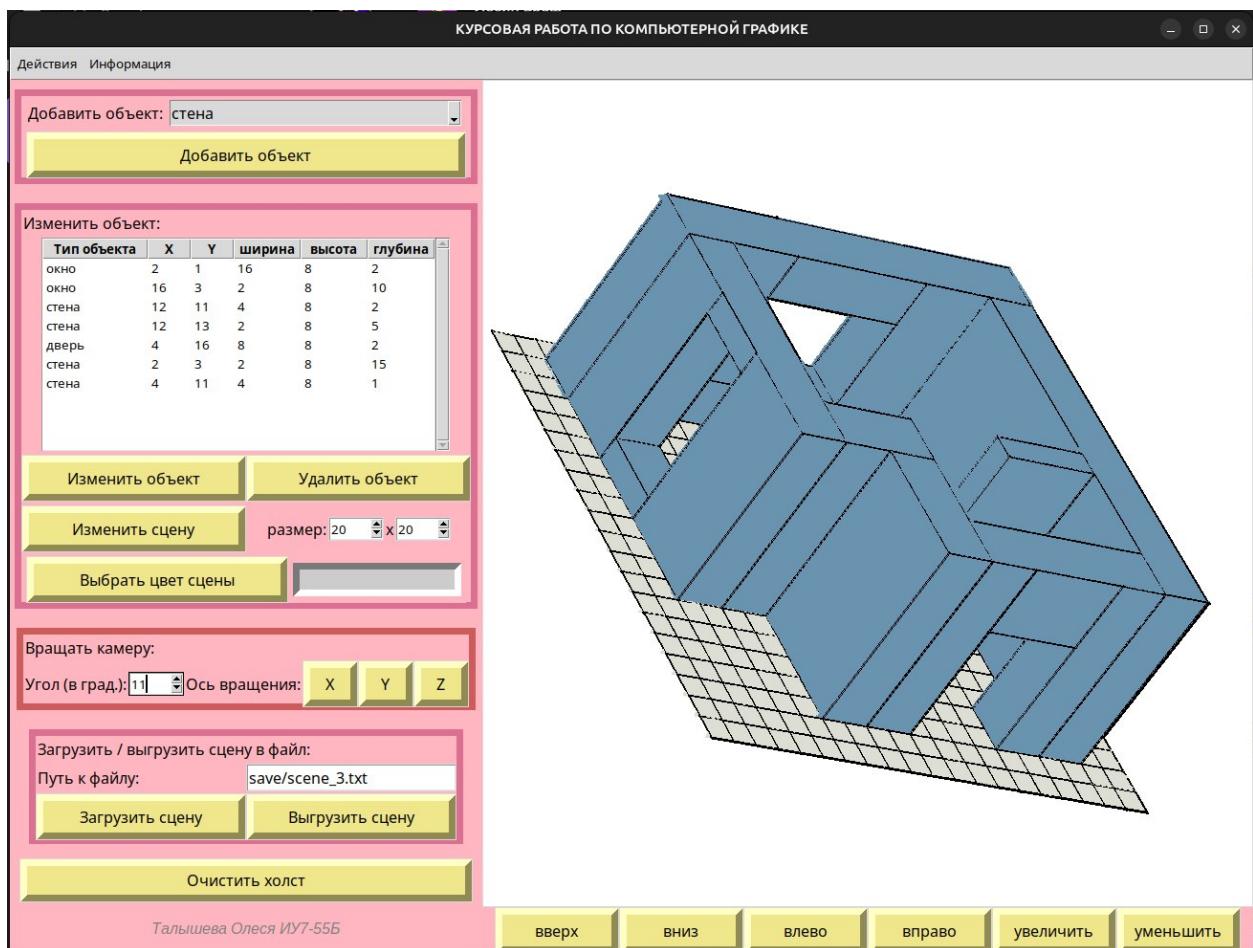


Рисунок 34 – Пример комнаты при углах наклона сцены 50, 50 и 1800 градусов по осям x, y и z соответственно.

#### 4.2. Сравнение времени работы программы

На таблице2 и рисунке 35 представлено сравнение последовательного и параллельного сбора информации для матрицы пикселей Z-буфера для удаления невидимых линий и теневого Z-буфера для времени работы программы.

Таблица 2 – Таблица сравнения времени работы программы.

Количество плоскостей	Последовательно (сек)	Параллельно (сек)	Ускорение
0	0.4017	1.9646	0.2045
1	1.3719	1.7637	0.7779
3	3.2953	1.7784	1.8529
6	5.9741	1.8698	3.1950
10	9.4483	2.0571	4.5930
15	13.9707	1.9893	7.0228
21	19.8980	2.2876	8.6983
28	24.7090	2.3127	10.6839
36	31.4424	2.3481	13.3905
45	40.6557	2.4021	16.9251
55	47.8978	2.4263	19.7408
66	56.9380	2.4817	22.9427
78	68.0594	2.4900	27.3330
91	77.5769	2.4611	31.5207
105	88.3629	2.5393	34.7987
120	101.3772	2.5574	39.6407
136	115.6641	2.6948	42.9207
153	129.3212	2.6200	49.3585
171	143.3958	2.6571	53.9665
190	159.1838	2.7908	57.0396

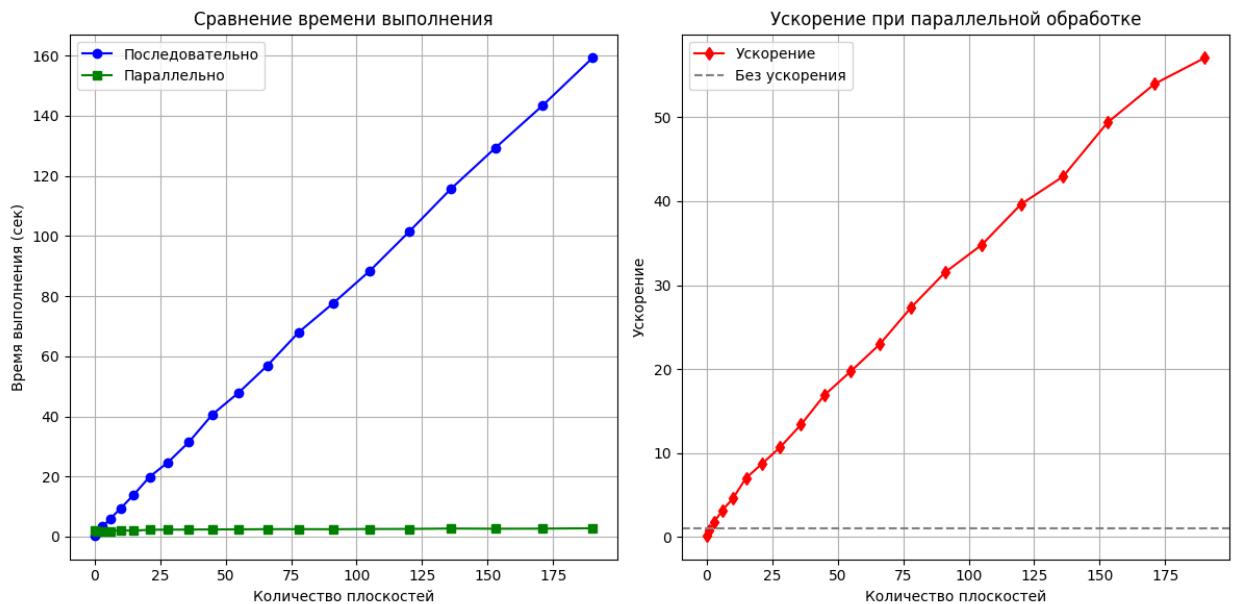


Рисунок 35 – Графики сравнения времени работы программы.

Левый график отображает время выполнения программы при последовательном (синяя кривая) и при параллельном вычислении Z-буфера (зелёная кривая). Для последовательного режима время выполнения линейно возрастает с увеличением числа плоскостей. Для параллельного режима время выполнения остаётся практически постоянным, с незначительными изменениями при увеличении числа плоскостей.

Правый демонстрирует ускорение работы программы при параллельной обработке (красная кривая). Таким образом, ускорение возрастает по мере увеличения числа плоскостей, что отражает эффективность параллельной обработки. При большом количестве плоскостей ускорение достигает значительных значений (более 140 раз).

#### 4.3. Выводы

По приведённым исследованиям видно, что:

1. Параллельная обработка значительно превосходит последовательную по времени выполнения, особенно при увеличении числа плоскостей.
2. Ускорение параллельной программы почти линейно растёт с увеличением числа задач, что подтверждает эффективность параллелизации.
3. Для небольшого числа плоскостей выигрыш от параллелизации менее заметен из-за накладных затрат на управление потоками.
4. Для больших объёмов задач параллельная программа демонстрирует почти идеальное масштабирование.

Следовательно, включение параллельной обработки позволяет существенно сократить время выполнения ресурсоёмких задач.

## ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы было разработано программное обеспечение для создания и редактирования 3D сцен помещений с возможностью интерактивного добавления объектов (стен, окон, дверей), их перемещения, изменения, поворота, а также обеспечения сохранения и загрузки моделей. Цель достигнута.

Выполнены все задачи:

- были проанализированы требования к программе и исследованы существующие решения;
- изучены алгоритмы реализации технических решений и выбраны наиболее подходящие для работы с 3D сценами;
- разработана архитектура и реализована программы;
- протестирована программа целиком и её отдельные модули;
- проведены исследования производительности программы;
- подготовлена отчётная документация.

В рамках работы была разработана программа для создания и редактирования 3D сцен помещений с возможностью добавления и управления объектами, а также работы с камерой. Проведённые исследования подтвердили эффективность использования алгоритма Z-буфера для удаления невидимых линий и построения теней, а также модели простой закраски для создания реалистичной визуализации.

Анализ производительности показал, что параллельная обработка значительно сокращает время выполнения задач, особенно при увеличении сложности сцены, что подтверждает целесообразность её применения.

Разработанное программное обеспечение обладает широкими возможностями и может быть использовано в задачах проектирования, анализа и безопасности. Перспективы развития включают добавление новых объектов и возможных действий пользователя, оптимизацию для работы с более сложными сценами.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989. — 512 с.
2. ИТЕР. Модель системы защиты [Электронный ресурс]. — Режим доступа: [https://iter.ru/model\\_sfz.html#section\\_9](https://iter.ru/model_sfz.html#section_9), дата обращения: 21.09.2024.
3. Польский С. В. Компьютерная графика: учеб.-метод. пособие. — М.: ГОУ ВПО МГУЛ, 2008. — 38 с.
4. Шикин Е. В., Боресков А. В. Компьютерная графика. Полигональные модели. — М.: Диалог-МИФИ, 2001. — 464 с.
5. Турлапов В. Е. Удаление невидимых поверхностей. Оптимизация. Тени [Электронный ресурс]. — Нижегородский государственный университет имени Н. И. Лобачевского, 2013. — Режим доступа: [http://www.graph.unn.ru/rus/materials/CG/CG13\\_HSROptimization.pdf](http://www.graph.unn.ru/rus/materials/CG/CG13_HSROptimization.pdf), дата обращения: 21.11.2024.
6. Алгоритм, использующий z-буфер [Электронный ресурс]. — Режим доступа: <https://compgraph.tpu.ru/zbuffer.htm>, дата обращения: 21.11.2024.
7. Шикин Е. В., Боресков А. В. Компьютерная графика. Динамика, реалистические изображения. — М.: Диалог-МИФИ, 1995. — 288 с.
8. Дерягина О. В., Корниенко В. В., Лагерь А. И., Скоробогатова Т. Е. Компьютерная графика: учебное пособие. — Красноярск: Красноярский государственный аграрный университет, 2014. — 322 с. — Режим доступа: [http://www.kgau.ru/distance/etc\\_06/komp-grafika/index.htm](http://www.kgau.ru/distance/etc_06/komp-grafika/index.htm), дата обращения: 21.11.2024.
9. Компьютерная графика: теория, алгоритмы, примеры на C++ и OpenGL [Электронный ресурс] / Освещение и модель затенения. URL: [https://compgraphics.info/3D\\_lighting/shading\\_model.php](https://compgraphics.info/3D_lighting/shading_model.php) (дата обращения: 21.11.2024).
10. Романюк А.Н., Куриенный М.В. Алгоритмы построения теней // Компьютеры + программы. — 2000. — № 8–9. — С. 5–6.
11. Однотонная закраска (метод гранения) [Электронный ресурс]. — Режим доступа: [https://studbooks.net/2248060/informatika/odnotonnaya\\_zakraska\\_metod\\_graneniya](https://studbooks.net/2248060/informatika/odnotonnaya_zakraska_metod_graneniya), дата обращения: 02.12.2024.