



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:*

**Построение сцены на основе трёхмерных объектов с  
использованием операций пересечения,  
объединения и вычитания**

Студент ИУ7-55Б  
(Группа)

\_\_\_\_\_  
(Подпись, дата) Р. А. Бакалдин  
(И.О.Фамилия)

Руководитель курсовой работы

\_\_\_\_\_  
(Подпись, дата) Д. А. Погорелов  
(И.О.Фамилия)

2023 г.

# Содержание

Содержание .....	2
Введение.....	4
1 Аналитический раздел .....	5
1.1 Структура трёхмерной сцены.....	5
1.2 Алгоритмы построения CSG моделей.....	5
1.2.1 Алгоритм, выполняющий разбиение по линии пересечения .....	6
1.2.2 Алгоритм, использующий BSP-дерево .....	7
1.3 Алгоритмы закрашивания .....	10
1.3.1 Простая закрашка .....	10
1.3.2 Закраска по Гуро .....	10
1.3.3 Закраска по Фонгу .....	11
1.4 Алгоритмы удаления невидимых линий и поверхностей .....	11
1.4.1 Алгоритм Робертса.....	12
1.4.2 Алгоритм Варнока.....	12
1.4.3 Алгоритм, использующий z-буфер.....	13
1.4.4 Алгоритм обратной трассировки лучей.....	14
1.5 Вывод из аналитического раздела .....	15
2 Конструкторский раздел.....	16
2.3 Структуры данных .....	16
2.3.1 Структуры данных, используемые в CSG.....	16
2.3.2 Структуры данных, используемые в визуализации сцены.....	17
2.4 Разработка алгоритмов .....	17
2.4.1 Общий алгоритм работы программы.....	17
2.4.2 Алгоритм построения CSG моделей.....	18
2.4.3 Выполнение координатных преобразований.....	21
2.4.4 Алгоритм z-буфера .....	22
2.4.5 Алгоритм простой закрашки.....	23
2.5 Структура программы .....	23
2.6 Вывод из конструкторского раздела.....	23
3 Технологический раздел.....	24
3.1 Выбор и обоснование средств разработки .....	24
3.2 Формат файла сцены.....	24

3.3	Полученная структура программы.....	25
3.4	Исходный код реализованных алгоритмов .....	26
3.5	Взаимодействие с программой .....	28
3.5.1	Установка зависимостей .....	29
3.5.2	Сборка программы.....	29
3.5.3	Запуск программы.....	29
3.6	Примеры работы программы .....	30
3.7	Вывод из технологического раздела .....	31
4	Исследовательский раздел .....	32
4.1	Цель проводимых измерений .....	32
4.2	Описание проводимых измерений .....	32
4.3	Параметры оборудования.....	33
4.4	Инструменты измерения времени работы.....	33
4.5	Результаты проведённых измерений .....	33
4.6	Вывод из исследовательского раздела.....	35
	Заключение .....	36
	Список использованных источников .....	37
	Приложение А Функциональная схема работы программы.....	38
	Приложение Б UML диаграмма.....	41
	Приложение В Презентация.....	42

# Введение

В настоящее время компьютерная графика стала неотъемлемой частью сферы информационных технологий и прочно вошла в повседневную жизнь. Сегодня она широко используется в таких областях, как визуализация данных, кинематограф, системы автоматизированного проектирования, компьютерные игры и многие другие. Поэтому перед создателями соответствующего программного обеспечения встаёт задача не только получения требуемой сцены, но и ускорения процесса её построения с помощью различных оптимизаций или допустимого снижения реалистичности.

Особенно остро этот вопрос стоит в САПР и компьютерных играх, выполненных в стиле низко полигональной графики или требующих от игрока быстрых и точных действий. В таких случаях обычно применяются различные технологии моделирования твёрдых тел, одна из которых – конструктивная блочная геометрия (англ. Constructive Solid Geometry, CSG) – широко применяется в вышеописанных случаях.

Цель данной работы – реализовать программу для построения сцены на основе простейших трёхмерных тел (куб, цилиндр, сфера, тор) с использованием операций пересечения, объединения и вычитания.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- 1) описать структуру трехмерной сцены, учитывая особенности подхода CSG;
- 2) выбрать и адаптировать существующие алгоритмы, позволяющие производить над простейшими трёхмерными телами требуемые операции;
- 3) выбрать и адаптировать существующие алгоритмы для визуализации полученной сцены;
- 4) реализовать выбранные и адаптированные алгоритмы;
- 5) исследовать возможности реализованной программы (быстродействие).

# 1 Аналитический раздел

В данном разделе будет рассмотрена предполагаемая структура трёхмерной сцены, построенной с использованием CSG, будут проанализированы существующие алгоритмы, позволяющие решить поставленные задачи и выбраны наиболее подходящие из них.

## 1.1 Структура трёхмерной сцены

Так как CSG позволяет представить модель любой формы, используя булевы операции над примитивами [1], будет целесообразно для наиболее полного описания сцены представить её как множество, содержащее следующие типы объектов:

- 1) базовый объект, представленный одним из примитивов (куб, цилиндр, сфера, тор);
- 2) составной объект, представленный бинарным деревом, каждый узел которого содержит информацию об используемой операции и двух операндах, каждый из которых может быть как базовым, так и составным объектом.

Таким образом, используя предложенную структуру, можно выполнять построение сцены независимо от выбранных методов отрисовки и/или построения CSG моделей.

## 1.2 Алгоритмы построения CSG моделей

Существует два типа алгоритмов построения CSG моделей: работающие в пространстве изображения и работающие в пространстве объекта. Последние являются более гибкими, так как обрабатывают модель до её вывода на экран и могут работать с невыпуклыми примитивами [1]. Исходя из этого, будет рассматриваться только второй подход.

### 1.2.1 Алгоритм, выполняющий разбиение по линии пересечения

Данный алгоритм, предложенный и описанный Погореловым Д. А., полагает, что исходные модели состоят из треугольных полигонов, и состоит из трёх основных этапов:

- 1) нахождение линии пересечения исходных моделей, что в свою очередь сводится к попарному нахождению отрезка пересечения двух треугольников;
- 2) разбиение каждого треугольника вдоль линии пересечения таким образом, чтобы каждый из отрезков являлся стороной двух смежных треугольников, для чего используется метод триангуляции Делоне (см. Рисунок 1);

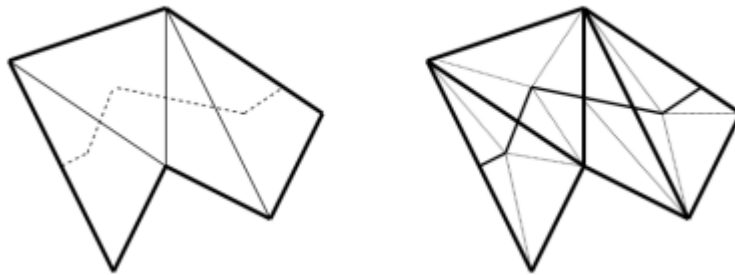


Рисунок 1. Разбиение треугольников вдоль линии пересечения

- 3) пространственная локализация полученных треугольников одного примитива относительно другого, что выполняется с помощью локализации барицентра треугольника с помощью метода бросания лучей.

Результатом алгоритма является связь ориентации треугольников и булевых операций. Относительно того, как расположен тот или иной треугольник, и какая булева операция используется, будут отображены или скрыты определенные треугольники (см. Таблица 1). [1]

Таблица 1. Связь ориентации треугольников с булевыми операциями

	Примитив А	Примитив В
$A \cup B$	снаружи	снаружи
$A \cap B$	внутри	внутри
$A \setminus B$	снаружи	внутри
$B \setminus A$	внутри	снаружи

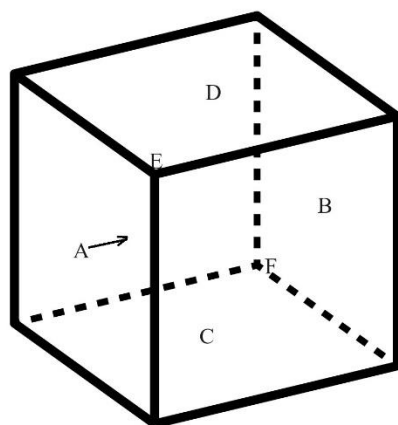
К несомненным достоинствам данного алгоритма относятся эффективность по памяти (память расходуется только на хранение данных о треугольниках и простейших промежуточных структурах) и получение результирующей модели, состоящей исключительно из треугольных полигонов (что значительно упрощает дальнейшую визуализацию).

Среди недостатков же можно выделить высокую трудоёмкость, связанную с выполнением триангуляции и бросания лучей для каждого разбиваемого треугольника, а также невозможность обрабатывать исходные модели, состоящие из полигонов, включающих более трёх вершин.

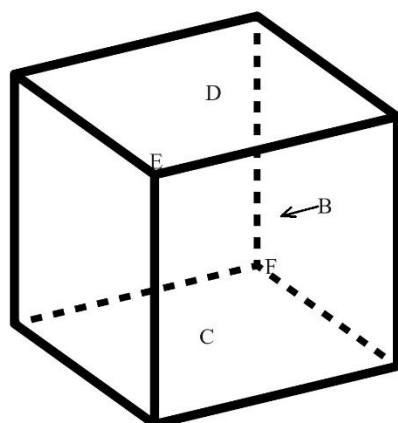
### 1.2.2 Алгоритм, использующий BSP-дерево

Данный алгоритм основан на построении дерева двоичного разбиения пространства (англ. binary space partitioning, BSP) для исходных моделей и дальнейшем объединении полученных деревьев. В алгоритме можно выделить следующие этапы:

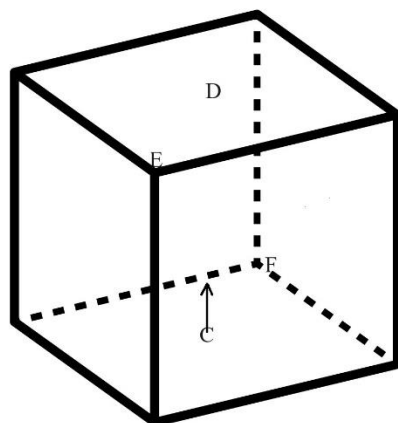
- 1) Рекурсивное разделение набора полигонов, составляющих модель, на два набора – находящиеся перед плоскостью раздела и позади неё [2]. Этот процесс наглядно иллюстрируется на примере куба, состоящего из шести полигонов (см. Рисунок 2).



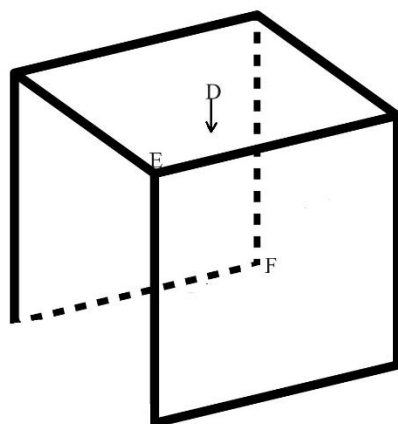
front  $\swarrow$  A  $\searrow$  back



front  $\swarrow$  A  $\searrow$  back  
front  $\swarrow$  B  $\searrow$  back



front  $\swarrow$  A  $\searrow$  back  
front  $\swarrow$  B  $\searrow$  back  
front  $\swarrow$  C  $\searrow$  back



front  $\swarrow$  A  $\searrow$  back  
front  $\swarrow$  B  $\searrow$  back  
front  $\swarrow$  C  $\searrow$  back  
front  $\swarrow$  D  $\searrow$  back

Рисунок 2. Первые четыре этапа разбиения куба



2) Объединение полученных BSP-деревьев в зависимости от требуемой операции.

Выполнение объединения моделей A и B эквивалентно отсечению всех полигонов A, которые находятся внутри B, и отсечению всех полигонов B, которые находятся внутри A (см. Рисунок 3) [2].

Выполнение же вычитания B из A состоит из трёх операций. Во-первых, отсекаются все полигоны A, лежащие внутри B. Во-вторых, отсекаются все полигоны B, лежащие вне A. В-третьих, инвертируется нормаль всех полигонов, полученных в результате второй операции (см. Рисунок 4). Пересечение находится по такому же принципу [2].

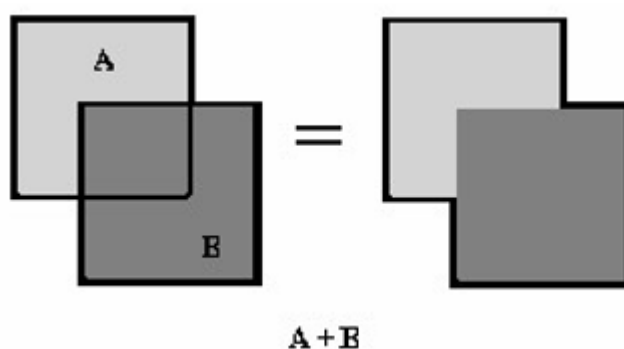


Рисунок 4. Объединение

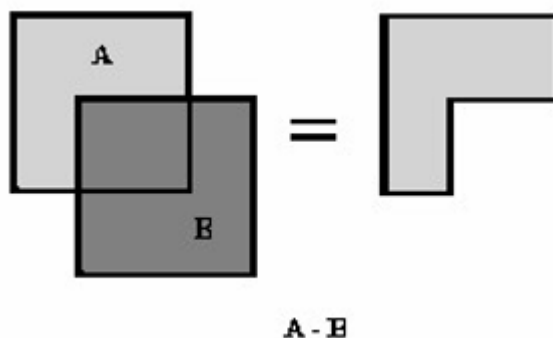


Рисунок 5. Вычитание

Пространственная локализация полигонов производится с помощью поочередного сравнения каждой вершины одного полигона с плоскостью другого и последующего принятия решения о положении всего полигона на основе информации о положении каждой вершины. Отсечение полигонов выполняется по тому же принципу.

Достоинствами данного алгоритма является способность обрабатывать любые виды выпуклых полигонов и невысокая трудоёмкость, обусловленная минимумом выполняемых на каждой итерации затратных операций, а также возможностью оптимизировать построение BSP-дерева.

Среди недостатков стоит отметить низкую эффективность по памяти и получение результирующей модели, состоящей из произвольных полигонов.

## 1.3 Алгоритмы закрашивания

### 1.3.1 Простая закрашка

При использовании простой закрашки весь полигон заполняется цветом одной интенсивности, которая вычисляется по закону косинусов Ламберта [3].

Простая закрашка может быть применена в следующих случаях:

- 1) источник света находится на бесконечном удалении от сцены;
- 2) наблюдатель находится на бесконечном удалении от сцены;
- 3) закрашиваемый полигон является плоским.

Основными недостатками алгоритма является то, что он не подходит для закрашки криволинейных поверхностей и никак не учитывает отражённый свет.

### 1.3.2 Закраска по Гуро

В отличие от простой закрашки, создающей изображение, состоящее из отдельных полигонов, закрашка по Гуро позволяет получить сглаженное изображение с помощью билинейной интерполяции, для чего сначала вычисляется интенсивность в вершинах полигона, после чего она интерполируется для каждого пиксела на сканирующей строке [3].

Несмотря на то, что применение данного метода сильно улучшает качество изображения, алгоритм имеет свои недостатки:

- 1) появляется эффект полос Маха;
- 2) теряется граница между полигонами, что в некоторых случаях может дать неверный результат.

### 1.3.3 Закраска по Фонгу

При закрашке по Фонгу, в отличие от закрашке по Гуро, вдоль сканирующей строки интерполируются не значения интенсивности, а вектора нормали. Такой подход требует больших вычислительных затрат, однако решает многие проблемы метода Гуро и в результате даёт более качественное изображение [3].

Сравнение описанных алгоритмов производится в таблице ниже (см. Таблица 1).

Таблица 1. Сравнение алгоритмов закрашки

Алгоритм	Вычислительная сложность	Особенности
Простая закрашка	Низкая	Не подходит для визуализации изображений с высокой реалистичностью
Закраска по Гуро	Высокая	Появляется эффект полос Маха
Закраска по Фонгу	Высокая	Высокие вычислительные затраты на интерполяцию нормалей

## 1.4 Алгоритмы удаления невидимых линий и поверхностей

Удаление невидимых линий, рёбер, поверхностей или объёмов необходимо выполнять для устранения неоднозначности интерпретации изображения. Сложность этой задачи привела к появлению большего числа различных алгоритмов, работающих как в пространстве изображения, так и в объектном пространстве [3]

### 1.4.1 Алгоритм Робертса

Алгоритм Робертса работает в объектном пространстве и выполняется согласно следующим этапам:

- 1) разбиение невыпуклых тел на выпуклые;
- 2) составление матриц для каждого тела, состоящих из коэффициентов уравнения плоскостей;
- 3) удаление точек, экранируемых каждой плоскостью (видимость точки относительно плоскости проверяется с помощью скалярного произведения её вектора в однородных координатах на вектор, содержащий коэффициенты уравнения плоскости).

К плюсам алгоритма следует отнести крайне высокую точность вычислений, обусловленную работой в объектном пространстве.

Из этого вытекают минусы: применяемые методы вычисления обладают высокой трудоёмкостью и с трудом поддаются оптимизации.

### 1.4.2 Алгоритм Варнока

Алгоритм Варнока работает в пространстве изображения. Рассматривается окно и решается вопрос о том, пусто ли оно, или его содержимое достаточно просто для визуализации. В противном случае окно разбивается на фрагменты до тех пор, пока не станет удовлетворять описанным выше условиям или его размер не достигнет требуемого предела разрешения (тогда информация, содержащаяся в окне, усредняется, а результат изображается с одинаковой интенсивностью).

В зависимости от реализации могут использоваться различные методы разбиения и рассматриваться различные способы определения простоты содержимого (см. Рисунок 6).

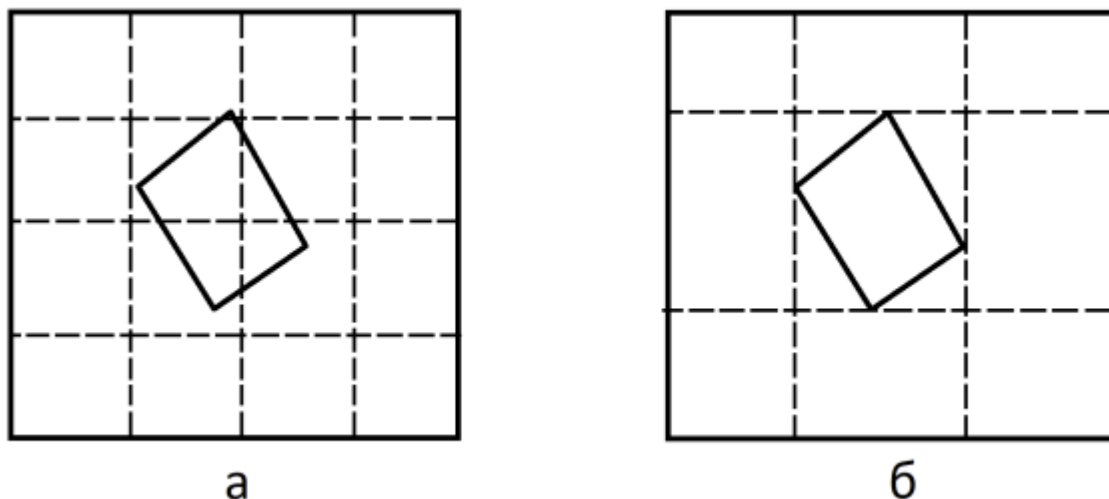


Рисунок 6. Различные способы разбиения окна

Важно отметить, что рекурсивное разбиение окон может стать как плюсом, так и минусом алгоритма, в зависимости от количества пересечений объектов сцены – чем их меньше, тем быстрее.

### 1.4.3 Алгоритм, использующий z-буфер

Данный алгоритм является одним из простейших и одновременно наиболее эффективных и широко распространённых на сегодняшний день алгоритмов, работающих в пространстве изображения. Его идея заключается в введении дополнительного буфера, используемого для запоминания ближайшей координаты  $z$  каждого видимого пикселя.

Плюсы алгоритма включают возможность работы со сценами любой сложности, отсутствие необходимости сортировки по приоритету глубины (элементы можно заносить в буфер кадра в произвольном порядке) и незначительное увеличение трудоёмкости отрисовки, включающее необходимость вычисления  $z$  координаты каждого пикселя и сравнения её со значением в буфере.

К минусам алгоритма относятся увеличение сложности по памяти в два раза (к буферу кадра добавляется  $z$ -буфер такого же размера) и трудоёмкость реализации эффектов прозрачности.

#### 1.4.4 Алгоритм обратной трассировки лучей

Алгоритм работает в пространстве изображения и основан на отслеживании (трассировке) лучей, идущих от наблюдателя к объекту. Такая трассировка называется обратной.

Для определения видимости от наблюдателя, находящегося на бесконечном удалении по оси  $z$ , испускается луч, проходящий через проверяемый пиксель растровой сетки. Траектория луча отслеживается, чтобы определить, какие именно объекты сцены, если таковые существуют, пересекаются с ним. Пересечение, произошедшее в точке с максимальным значением  $z$ , даст информацию о цвете данного пикселя (см. Рисунок 7).

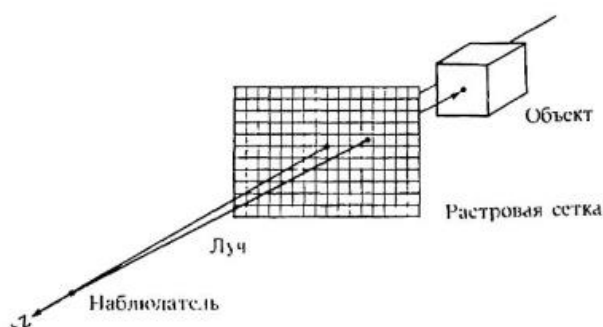


Рисунок 7. Иллюстрация работы алгоритма

Трудоёмкость данного алгоритма зависит не только от метода поиска пересечений и количества объектов сцены, но и размеров растровой сетки, что является одним из главных минусов. Кроме того, в случае реализации эффектов прозрачности сложность вычислений дополнительно возрастет.

Основным плюсом алгоритма являются качество и реалистичность изображения и простота реализации освещения и затенения.

Сравнение описанных алгоритмов приводится в таблице ниже (см. Таблица 2).

Таблица 2. Сравнение алгоритмов удаления невидимых линий

Алгоритм	Пространство	Сложность ( $n$ – количество полигонов, $N$ )	Тело	Особенности

		– количество пикселей)		
Робертса	Объектное	$O(n^2)$	Только выпуклое	Большой объём вычислений
Варнока	Изображение	$O(nN)$	Любое	Худший случай приведёт к разбиению окна до размеров пикселя
Z-буфер	Изображение	$O(nN)$	Любое	Высокое потребление памяти
Обратной трассировки лучей	Изображение	$O(nN)$	Любое	Большое количество вычислений для каждого луча

## 1.5 Вывод из аналитического раздела

В данном разделе была описана структура трёхмерной сцены, рассмотрены алгоритмы построения CSG моделей, а также алгоритмы, необходимые для визуализации трёхмерной сцены, такие как алгоритмы закрашивания и алгоритмы удаления невидимых линий и поверхностей. В качестве алгоритма построения CSG моделей был выбран алгоритм на основе BSP-дерева ввиду его меньшей трудоёмкости, в качестве алгоритма удаления невидимых линий и поверхностей – z-буфер ввиду его гибкости, низкой трудоёмкости и отсутствия эффектов прозрачности, а в качестве алгоритма закраски – простая закрашка по причине её небольшой вычислительной сложности и соответствии требованиям технического задания, предусматривающим наличие бесконечно удалённого источника света и отсутствие криволинейных поверхностей.

## 2 Конструкторский раздел

В данном разделе будут описаны функциональная модель, структура программы, а также алгоритмы и структуры данных, выбранные для решения поставленной задачи.

### 2.3 Структуры данных

Так как планируется независимая реализация модуля, содержащего алгоритм построения CSG моделей, и модуля, отвечающего за визуализацию трёхмерных моделей, необходимо для каждого из них описать соответствующие структуры данных перед дальнейшей формализацией рассматриваемых алгоритмов. Среди общих же структур, необходимых для выполнения математических операций, можно выделить:

- 1) трёхмерный вектор, содержащий три координаты, представленные числами с плавающей точкой;
- 2) четырёхмерный вектор, содержащий четыре координаты, представленные числами с плавающей точкой;
- 3) квадратная матрица размером четыре на четыре, содержащая шестнадцать чисел с плавающей точкой.

#### 2.3.1 Структуры данных, используемые в CSG

Как было упомянуто в анализе алгоритма, основанного на BSP, он способен обрабатывать модели, состоящие из полигонов с любым количеством вершин. Кроме того, требуются также структуры, описывающие соответствующий узел дерева и плоскость, относительно которой будет выполняться отсечение полигонов. Следовательно, определим следующие структуры:

- 1) вершина, содержащая трёхмерный вектор с координатами и трёхмерный вектор, представляющий нормаль к вершине;
- 2) плоскость, содержащая трёхмерный вектор с координатами точки, принадлежащей этой плоскости и трёхмерный вектор нормали к плоскости;



- 3) полигон, содержащий массив вершин и плоскость, которой этот полигон принадлежит;
- 4) узел BSP-дерева, содержащий массив полигонов, плоскость, которой этот узел принадлежит, и ссылки на узлы, находящиеся впереди и позади;
- 5) твёрдое тело, содержащее массив полигонов.

Таким образом мы получаем набор структур данных, полностью описывающий все сущности, используемые в алгоритме.

### 2.3.2 Структуры данных, используемые в визуализации сцены

С учётом выбранных алгоритмов визуализации (простой закрашки и z-буфера), необходимо определить следующие структуры:

- 1) треугольник, содержащий три вектора, содержащие соответственно координаты трёх его вершин;
- 2) полигон, содержащий треугольник и вектор нормали.

Так как по техническому заданию цвет полигонов однородный, а источник света бесконечно удалён, никаких других данных дополнительно не требуется.

## 2.4 Разработка алгоритмов

### 2.4.1 Общий алгоритм работы программы

Общий алгоритм работы программы подразумевает четыре основных этапа:

- 1) считывание данных о примитивах и операциях над ними;
- 2) построение полигональных моделей алгоритмом CSG;
- 3) отрисовка полученных моделей в буфер кадра с учётом камеры и освещения;
- 4) передача полученного буфера кадра графическому API операционной системы для дальнейшей отрисовки.

Диаграмма, отражающая декомпозицию данного алгоритма и оформленная в соответствии с нотацией IDEF0, представлена в приложении А.

## 2.4.2 Алгоритм построения CSG моделей

В основе данного алгоритма, в частности, построении BSP-дерева и последующего отсечения, лежит пространственная локализация одного полигона относительно другого. Автор алгоритма предлагает выполнять её в два этапа. Сначала рассматривается каждая вершина локализуемого полигона, после чего на основе полученной информации принимается решение о положении самого полигона.

Локализация вершины относительно плоскости выполняется путём определения знака следующего выражения [2]:

$$value = \vec{n} * \vec{p} + \vec{n} * \vec{a}, \quad (1)$$

где  $\vec{n}$  – нормаль к плоскости,  $\vec{p}$  – локализуемая вершина,  $\vec{a}$  – точка, принадлежащая плоскости. Соответственно, если полученное значение меньше или равно нулю, точка считается лежащей позади плоскости, иначе – впереди.

Чтобы перейти к локализации полигона, достаточно подсчитать количество обоих классов точек. Тогда позиция полигона определяется, исходя из следующих условий:

- 1) если все точки лежат перед плоскостью, то полигон также лежит перед плоскостью;
- 2) если все точки лежат позади плоскости, то полигон также лежит позади плоскости;
- 3) если присутствуют точки, лежащие как позади, так и сзади, полигон считается пересекающимся.

Теперь, имея алгоритм, позволяющий определить положение одного полигона относительно другого, можно переходить к алгоритму построения BSP-дерева, принимающего на вход корневой узел дерева и массив полигонов и состоящего из четырёх этапов [2]:

- 1) из массива полигонов выбирается плоскость отсечения и заносится в текущий узел;
- 2) выполняется разделение полигонов на лежащие впереди, позади и копланарные (если полигон пересекается, его разбивают на две части);

- 3) копланарные полигоны добавляются в текущий узел;
- 4) если найдены полигоны, лежащие впереди или позади, для каждого из полученных множеств алгоритм рекурсивно повторяется.

Блок схема данного алгоритма показана на Рисунке 8.

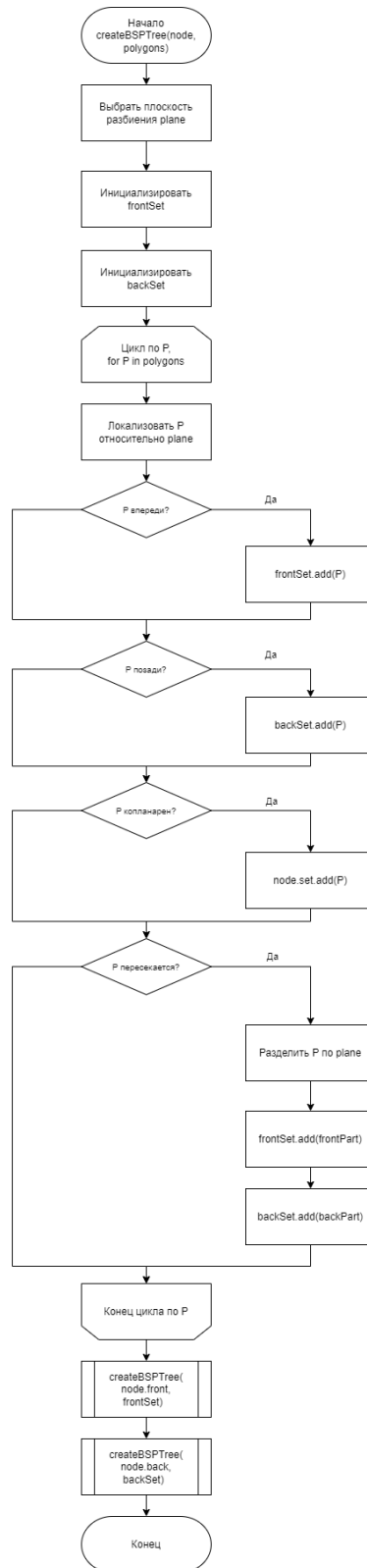


Рисунок 8. Блок схема алгоритма создания BSP-дерева

Разделение полигона относительно плоскости может быть тривиально произведено как разделение полигона по линии пересечения с искомой плоскостью. Для поиска точек, принадлежащих этой линии, воспользуемся следующим способом, основанным на билинейной интерполяции:

$$t = \frac{\vec{n} * \vec{a} - \vec{n} * \vec{V}_1}{\vec{n} * (\vec{V}_2 - \vec{V}_1)}, \quad (2)$$

$$V = \vec{V}_1 + (\vec{V}_2 - \vec{V}_1) * t, \quad (3)$$

где  $\vec{n}$  – нормаль к плоскости,  $\vec{a}$  – точка, принадлежащая плоскости,  $\vec{V}_n$  – соответствующая вершина, принадлежащая к разделяемой грани.

Точки, полученные этим способом, образуют линию пересечения, которая разделит полигон на две части.

Наконец, остаётся реализовать две операции над деревом: инверсию (в ходе которой для инвертируются нормали каждого полигона, а в каждом узле меняется порядок дочерних узлов) и отсечение (в ходе которого все полигоны одного дерева будут отсечены полигонами другого дерева, что выполняется по уже описанным принципам).

После этого можно выразить все требуемые булевы операции через последовательность инверсии и отсечения [2].

Таким образом, объединение, примитивов  $a$  и  $b$  можно представить, как следующую последовательность операций:

- 1) Отсечение  $a$  по  $b$
- 2) Отсечение  $b$  по  $a$
- 3) Инверсия  $b$
- 4) Отсечение  $b$  по  $a$
- 5) Инверсия  $b$

Последовательности для операций вычитания и объединения могут быть получены путём выражения этих операций через объединение:

$$A - B = \sim(\sim A \mid B), \quad (4)$$

$$A \& B = \sim(\sim A \mid \sim B), \quad (5)$$

где « $\sim$ » – оператор дополнения.

### 2.4.3 Выполнение координатных преобразований

Учитывая требование к изменению положения камеры, а также недостатки ортогональной проекции, проявляющиеся для пользователя в сложности восприятия сцены при свободном движении камеры по сцене, необходимо ввести дополнительные преобразования координат вершин полигонов, производимые перед их отрисовкой.

Преобразования будут выполняться следующим образом:

- 1) Получение матрицы, переводящей координаты из объектного пространства в экранное.

$$Viewport = \begin{pmatrix} \frac{w}{2} & 0 & 0 & x + \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y + \frac{h}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (6)$$

где  $w$ ,  $h$  – ширина и высота сетки растра соответственно, а  $x$  и  $y$  – начальные значения координат растра.

- 2) Получение матрицы перспективной проекции, обеспечивающей корректное искажение в зависимости от удаления от наблюдателя.

$$Projection = \begin{pmatrix} \frac{\cot(\frac{fov}{2})}{aspect} & 0 & 0 & 0 \\ 0 & \cot(\frac{fov}{2}) & 0 & 0 \\ 0 & 0 & \frac{zFar + zNear}{zNear - zFar} & \frac{2*zFar*zNear}{zNear - zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \quad (7)$$

где  $fov$  – угол обзора,  $aspect$  – соотношение ширины и высоты растра,  $zNear$  – расстояние до ближайшей плоскости усечённой пирамиды проекции,  $zFar$  – расстояние до дальней плоскости [4].

- 3) Получение матрицы взгляда, перемещающей все объекты сцены так, чтобы создать иллюзию перемещения камеры.

$$LookAt = \begin{pmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (8)$$

где  $R$  – вектор, указывающий в положительном направлении оси  $x$  относительно камеры,  $D$  – вектор, указывающий направление камеры,  $U$  – вектор, указывающий в положительном направлении оси  $y$  относительно камеры,  $P$  – вектор, содержащий позицию камеры [5]

- 4) Получение результирующей матрицы и умножение её на вектор, содержащий координаты преобразуемой точки.

$$Position = Viewport * Projection * LookAt * \begin{pmatrix} V_x \\ V_y \\ V_z \\ 1 \end{pmatrix}, \quad (9)$$

где  $V$  – вектор координат преобразуемой вершины.

- 5) Получение преобразованных координат вершины.

$$Transformed = \begin{pmatrix} \frac{Position_x}{Position_w} & \frac{Position_y}{Position_w} & \frac{Position_z}{Position_w} \end{pmatrix} \quad (10)$$

Таким образом, благодаря применению данных преобразований, реализация возможности перемещения камеры сводится к изменению значений векторов, описывающих позицию камеры.

Однако необходимо отметить, что для получения корректного результата после применения матрицы перспективной проекции необходимо выполнить отсечение полигонов относительно ближней и дальней плоскостей усечённой пирамиды камеры.

#### 2.4.4 Алгоритм z-буфера

В данном случае алгоритм z-буфера может быть реализован с помощью передачи в алгоритм двумерного растеризатора треугольников инициализированного минимально возможными значениями буфера и введения дополнительной проверки, заключающейся в вычислении для каждого пикселя, составляющего треугольник, его  $z$ -координаты и сравнения её с уже хранящимся в буфере значением. Для используемого алгоритма растеризации, основанного

на барицентрических координатах (для каждого пикселя вычисляется соответствующая координата и проверяется на нахождение внутри), глубина точки может быть вычислена по следующей формуле:

$$z = B_x * Z_1 + B_y * Z_2 + B_z * Z_3, \quad (11)$$

где  $B$  – вектор, содержащий барицентрические координаты пикселя, а  $Z_n$  –  $z$ -координата соответствующей вершины треугольника.

#### 2.4.5 Алгоритм простой закрашки

Данный алгоритм производит заполнения полигона цветом одной интенсивности, вычисляемой по закону косинусов Ламберта:

$$I_\alpha = I * \cos(\alpha), \quad (12)$$

где  $I$  – изначальная интенсивность цвета,  $\alpha$  – угол между нормалью полигона и нормалью источника света [3]. В данном случае для бесконечно удалённого источника света, заданного непосредственно только его нормалью, искомый косинус может быть найден следующим способом:

$$\cos(\alpha) = \frac{\vec{p} * \vec{l}}{|\vec{p}| * |\vec{l}|}, \quad (13)$$

где  $p$  – нормаль полигона, а  $l$  – нормаль источника света.

### 2.5 Структура программы

Для повышения переиспользуемости кода и возможности в будущем с лёгкостью заменить или модифицировать один из используемых алгоритмов, программа будет разработана с использованием парадигмы объектно-ориентированного программирования в виде двух основных логических модулей, слабо связанных друг с другом. В приложении Б представлена UML диаграмма, иллюстрирующая выбранную структуру.

### 2.6 Вывод из конструкторского раздела

В данном разделе были разработаны функциональная модель, структура программы, выбранные для решения поставленной задачи алгоритмы и структуры данных, также приведены основные математические соотношения, используемые в их реализации.

## 3 Технологический раздел

В данном разделе будут рассмотрены детали реализации программы, описанной в конструкторской части, и приведены примеры работы программы.

### 3.1 Выбор и обоснование средств разработки

В качестве языка программирования, на котором будет реализована программа, выбран язык Java [6]. Этот выбор обусловлен следующими причинами:

- 1) поддержка языком парадигмы ООП;
- 2) строгая типизация;
- 3) наличие библиотек, реализующих операции над трёхмерными векторами;
- 4) наличие библиотек, позволяющих получить доступ к графическому API системы
- 5) большое количество документации и материалов о разработке графических алгоритмов.

В качестве среды разработки был выбран редактор IntelliJ IDEA [7], созданный специально для разработки программ на языке Java и включающий обширный пакет таких инструментов, как автоматическая интеграция с системами управления версий и различными системами сборки, мощный статический анализатор, позволяющий поддерживать качество кода на высоком уровне, и отладчик с графическим интерфейсом.

### 3.2 Формат файла сцены

Разработанная программа для сохранения и загрузки сцены использует формат JSON, в который записываются объекты двух типов, составляющие сцену: объекты-примитивы и составные объекты, фактически представляющие собой узел бинарного дерева, в котором хранится информация о выполняемой булевой операции и её операндах. Благодаря этому данный формат потребляет гораздо меньше места, нежели форматы, сохраняющие полную информацию о полигонах модели, и может быть легко использован другой программой.



### 3.3 Полученная структура программы

После реализации в программе можно выделить следующие логические модули:

- 1) `math` – модуль, содержащий утилитные классы с реализацией различных математических операций;
- 2) `csg` – модуль, отвечающий за построение CSG объектов;
- 3) `graphic` – модуль визуализации полигонов, содержащий реализации выбранных алгоритмов;
- 4) `scene` – модуль, инкапсулирующий взаимодействие со сценой, её объектами и камерой;
- 5) `json` – модуль чтения и записи файла сцены.

Диаграмма модулей представлена на рисунке 9.

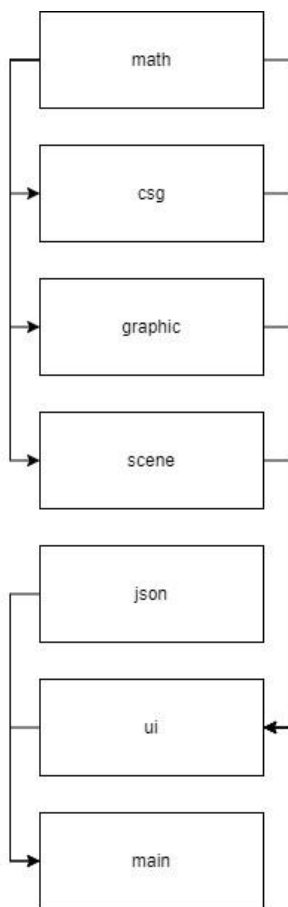


Рисунок 9. Диаграмма модулей

В результате полученная архитектура позволяет изменять реализацию внутри одного модуля, не затрагивая другие, и открывает возможность для переиспользования одного из модулей в других программах.

### 3.4 Исходный код реализованных алгоритмов

В листинге 1 представлен код функции барицентрического растеризатора треугольников с z-буфером.

Листинг 1. Алгоритм растеризации

```
1: public void rasterize(Triangle triangle, int value) {
2:     // Coordinates
3:     // First vertex
4:     var x1 = triangle.first.x();
5:     var y1 = triangle.first.y();
6:     var z1 = triangle.first.z();
7:     // Second vertex
8:     var x2 = triangle.second.x();
9:     var y2 = triangle.second.y();
10:    var z2 = triangle.second.z();
11:    // Third vertex
12:    var x3 = triangle.third.x();
13:    var y3 = triangle.third.y();
14:    var z3 = triangle.third.z();
15:    // Bounds
16:    var minX = (int) Math.max(0, Math.ceil(Math.min(x1, Math.min(x2, x3))));
17:    var maxX = (int) Math.min(width - 1, Math.floor(Math.max(x1, Math.max(x2, x3))));
18:    var minY = (int) Math.max(0, Math.ceil(Math.min(y1, Math.min(y2, y3))));
19:    var maxY = (int) Math.min(height - 1, Math.floor(Math.max(y1, Math.max(y2, y3))));
20:    // Calculate triangle area
21:    var area = (y1 - y3) * (x2 - x3) + (y2 - y3) * (x3 - x1);
22:    // Point-by-point rendering
23:    for (var y = minY; y <= maxY; ++y) {
24:        for (var x = minX; x <= maxX; ++x) {
25:            var b1 = ((y - y3) * (x2 - x3) + (y2 - y3) * (x3 - x)) / area;
26:            var b2 = ((y - y1) * (x3 - x1) + (y3 - y1) * (x1 - x)) / area;
27:            var b3 = ((y - y2) * (x1 - x2) + (y1 - y2) * (x2 - x)) / area;
28:            if (b1 >= 0 && b1 <= 1 && b2 >= 0 && b2 <= 1 && b3 >= 0 && b3 <= 1) {
29:                var depth = b1 * z1 + b2 * z2 + b3 * z3;
30:                var index = y * width + x;
31:                if (buffer[index] < depth) {
32:                    raster.setPixel(x, y, value);
33:                    buffer[index] = depth;
34:                }
35:            }
36:        }
37:    }
38: }
```

Перед растеризацией происходит сохранение координат вершин в локальные переменные, чтобы при частом обращении к ним избежать лишних операций доступа к полям класса, после чего определяются координаты ограничивающей треугольник прямоугольной области и происходит её попиксельный обход, на каждом этапе которого после проверки барицентрических координат и глубины пиксель заносится в растр.

В листингах 2, 3 и 4 представлены реализации функций булевых операций над объектами CSG, реализованные через операции инверсии и отсечения над узлами BSP-дерева.

#### Листинг 2. Функция объединения

```
1: public Solid union(Solid solid, BiFunction<PropertyStorage, PropertyStorage,
  PropertyStorage> merger) {
2:     var a = new BSPNode(copy(polygons));
3:     var b = new BSPNode(copy(solid.getPolygons()));
4:     a.clipTo(b);
5:     b.clipTo(a);
6:     b.invert();
7:     b.clipTo(a);
8:     b.invert();
9:     a.build(b.getAllPolygons());
10:    return new BSPSolid(a.getAllPolygons(), merger.apply(storage,
solid.getStorage()));
11: }
```

#### Листинг 3. Функция вычитания

```
1: public Solid difference(Solid solid, BiFunction<PropertyStorage, PropertyStorage,
  PropertyStorage> merger) {
2:     var a = new BSPNode(copy(this.polygons));
3:     var b = new BSPNode(copy(solid.getPolygons()));
4:     a.invert();
5:     a.clipTo(b);
6:     b.clipTo(a);
7:     b.invert();
8:     b.clipTo(a);
9:     b.invert();
10:    a.build(b.getAllPolygons());
11:    a.invert();
12:    return new BSPSolid(a.getAllPolygons(), merger.apply(storage,
solid.getStorage()));
13: }
```

#### Листинг 4. Функция пересечения

```
14: public Solid intersect(Solid solid, BiFunction<PropertyStorage, PropertyStorage,  
    PropertyStorage> merger) {  
15:     var a = new BSPNode(copy(this.polygons));  
16:     var b = new BSPNode(copy(solid.getPolygons()));  
17:     a.invert();  
18:     b.clipTo(a);  
19:     b.invert();  
20:     a.clipTo(b);  
21:     b.clipTo(a);  
22:     a.build(b.getAllPolygons());  
23:     a.invert();  
24:     return new BSPSolid(a.getAllPolygons(), merger.apply(storage,  
        solid.getStorage()));  
25: }
```

Передаваемый объект функционально интерфейса `merger` здесь нужен для того, чтобы позволить вызывающей стороне управлять объединением метаданных CSG тел (например, при объединении красного и жёлтого тел можно получить как красное, так и жёлтое тело в зависимости от переданной реализации интерфейса).

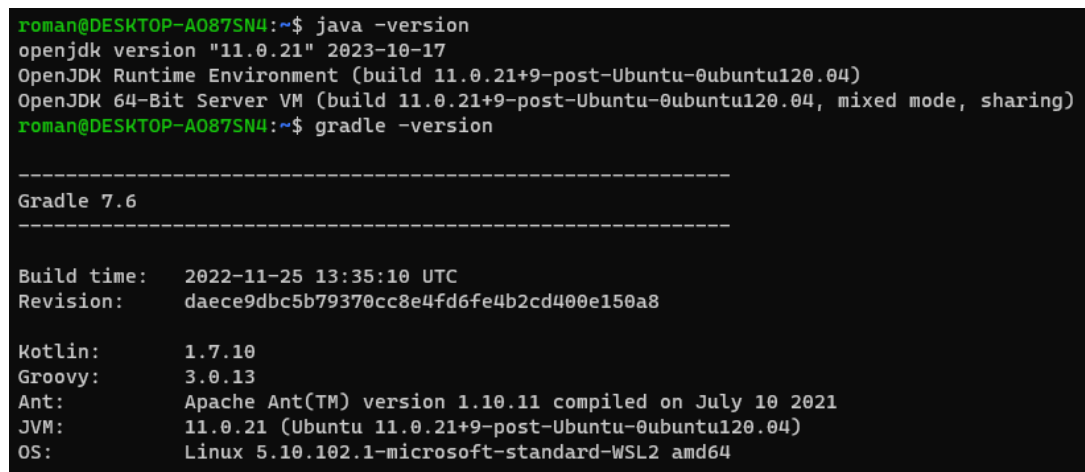
### 3.5 Взаимодействие с программой

В программе был реализован графический пользовательский интерфейс, позволяющий пользователю загружать, сохранять, изменять и визуализировать сцену, а также менять положение камеры. Операции над сценой осуществляются с помощью элементов верхнего и правого контекстного меню, наглядно представляющих каждый объект и позволяющих изменять его параметры и применять к нему преобразования (перемещения и повороты). Управление камерой осуществляется как с помощью элементов контекстного меню путём ввода позиции камеры и углов Эйлера, так и с помощью горячих клавиш, позволяющих за одно нажатие изменять положение камеры на фиксированное значение.

Для получения функционирующей программы и её запуска необходимо выполнить три шага: установить зависимости, собрать исполняемый файл из проекта и запустить `jar`-файл.

### 3.5.1 Установка зависимостей

Так как программа разработана с использованием языка Java, для её сборки и запуска необходимо установить Java Development Kit не менее чем 11 версии [8]. Также программа создана с помощью системы автоматической сборки Gradle 7 версии, которую необходимо установить и корректно сконфигурировать [9]. Проверить работоспособность полученных пакетов можно так, как показано на рисунке 10.



```
roman@DESKTOP-A087SN4:~$ java -version
openjdk version "11.0.21" 2023-10-17
OpenJDK Runtime Environment (build 11.0.21+9-post-Ubuntu-0ubuntu120.04)
OpenJDK 64-Bit Server VM (build 11.0.21+9-post-Ubuntu-0ubuntu120.04, mixed mode, sharing)
roman@DESKTOP-A087SN4:~$ gradle -version

-----
Gradle 7.6
-----

Build time:   2022-11-25 13:35:10 UTC
Revision:    daece9dbc5b79370cc8e4fd6fe4b2cd400e150a8

Kotlin:      1.7.10
Groovy:      3.0.13
Ant:         Apache Ant(TM) version 1.10.11 compiled on July 10 2021
JVM:         11.0.21 (Ubuntu 11.0.21+9-post-Ubuntu-0ubuntu120.04)
OS:          Linux 5.10.102.1-microsoft-standard-WSL2 amd64
```

Рисунок 10. Проверка работоспособности и версии java и gradle

### 3.5.2 Сборка программы

Для сборки программы необходимо перейти в директорию, содержащую проект (файл build.gradle), и выполнить команду «gradle shadowJar», так, как показано на рисунке 11. Полученный в результате сборки jar-файл будет находиться в директории build/libs.



```
roman@DESKTOP-A087SN4:~$ cd csg
roman@DESKTOP-A087SN4:~/csg$ gradle shadowJar
Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused, use --status for details

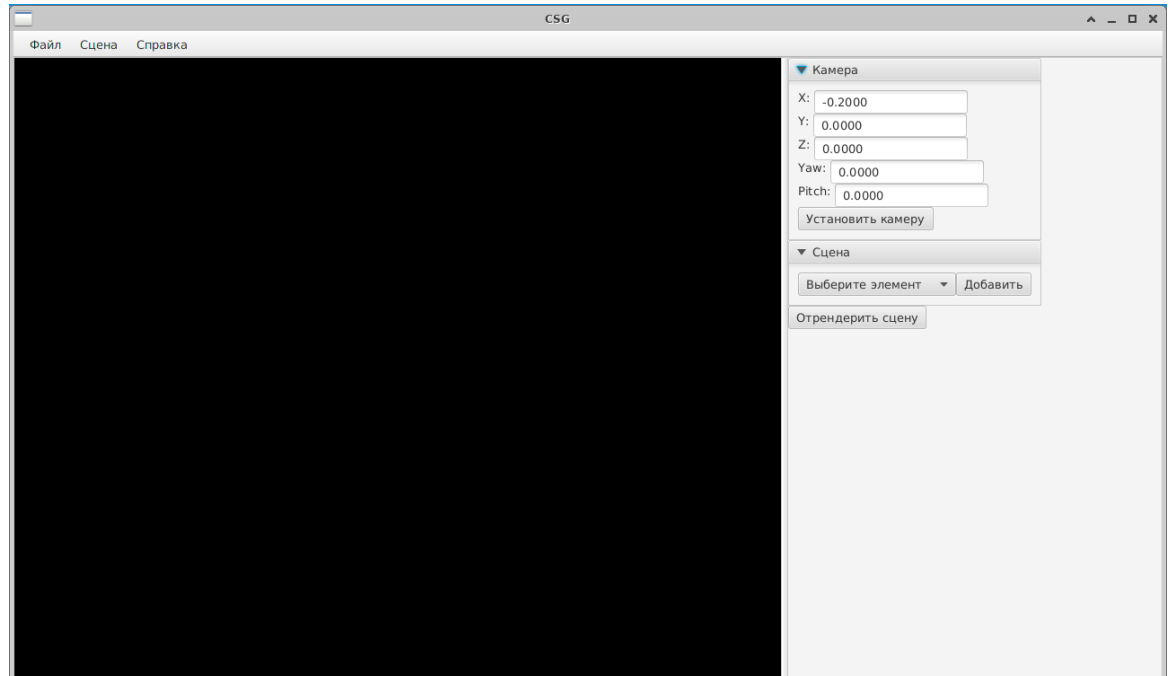
BUILD SUCCESSFUL in 16s
3 actionable tasks: 3 executed
roman@DESKTOP-A087SN4:~/csg$ ls -la build/libs/
total 8640
drwxr-xr-x 2 roman roman 4096 Dec 19 12:54 .
drwxr-xr-x 7 roman roman 4096 Dec 19 12:54 ..
-rw-r--r-- 1 roman roman 8838219 Dec 19 12:54 csg.jar
```

Рисунок 11. Сборка программы

### 3.5.3 Запуск программы

После получения исполняемого jar-файла для запуска достаточно просто запустить его с помощью Java Virtual Machine. Единственным моментом,

требующим дополнительного внимания, может быть настройка максимально допустимого размера стека, так как программа реализована с помощью рекурсивных алгоритмов. Интерфейс запущенной программы выглядит, как показано на рисунке 12.



### 3.6 Примеры работы программы

На рисунках 13, 14 и 15 приведены полученные изображения сцены, состоящей из двух примитивов (куба и сферы), к которым были применены соответственно объединение, вычитание и пересечение.

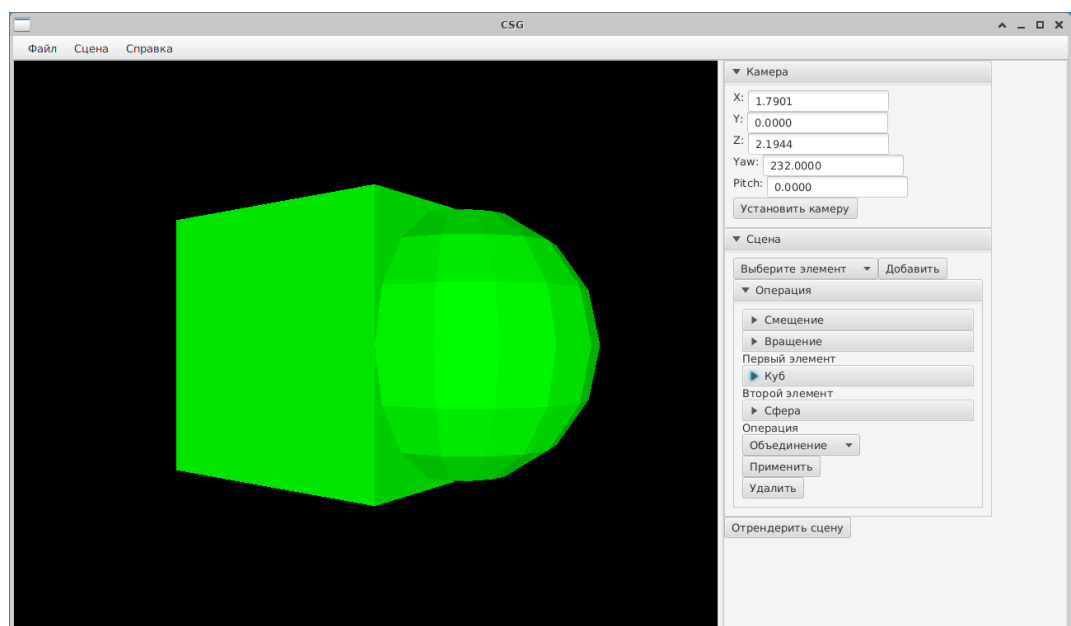


Рисунок 13. Объединение куба и сферы

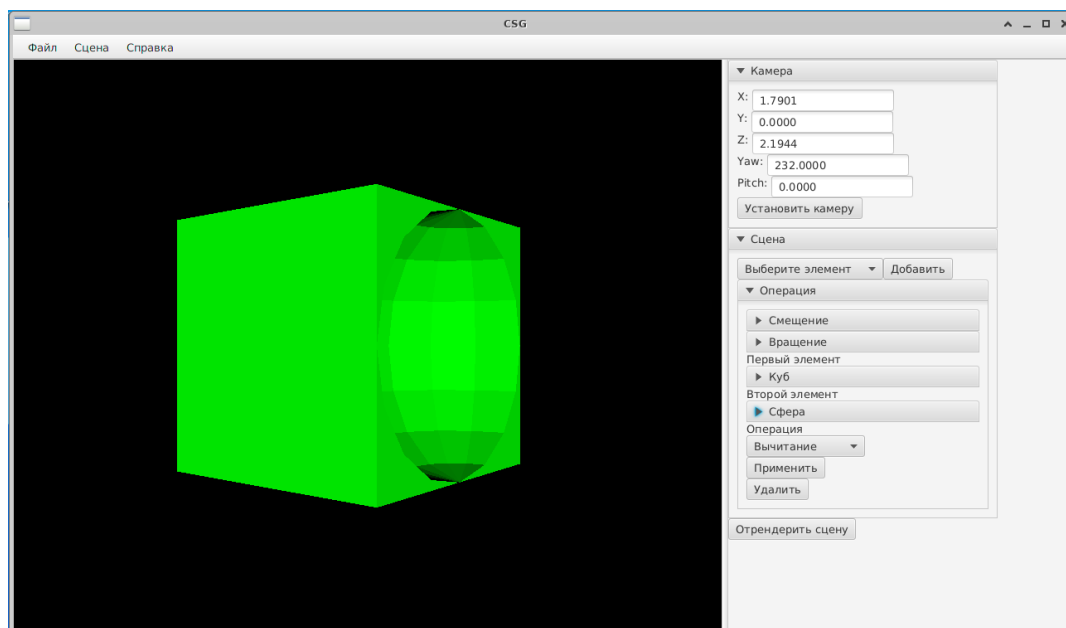


Рисунок 14. Вычитание сферы из куба

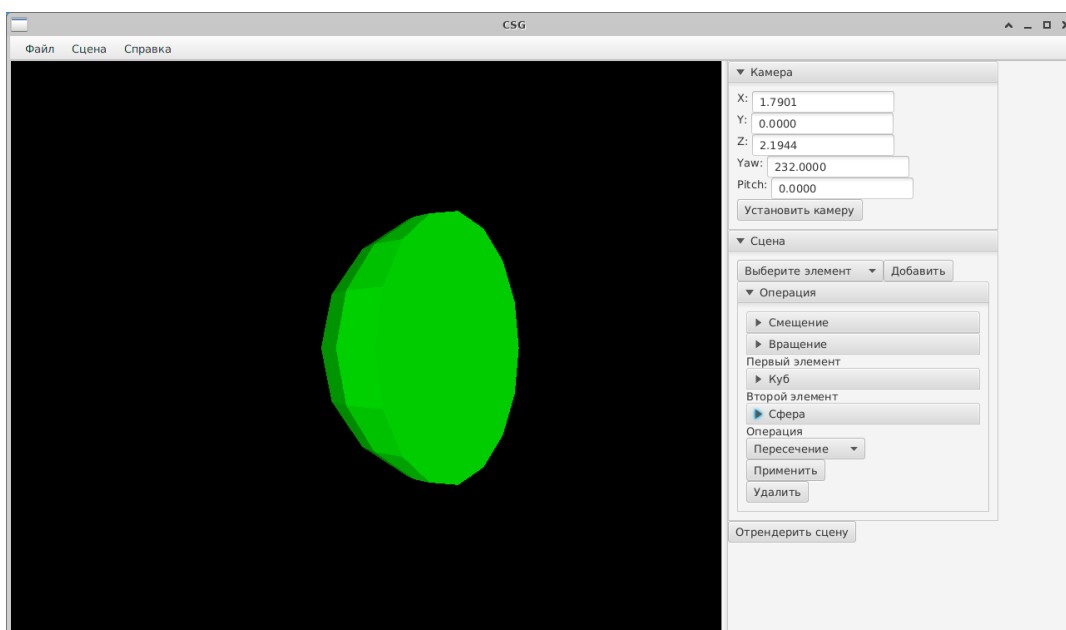


Рисунок 15. Пересечение куба и сферы

### 3.7 Вывод из технологического раздела

В данном разделе была рассмотрена реализация программы по визуализации CSG сцен, разработанной с использованием алгоритма построения CSG моделей на основе BSP-дерева, z-буфера и метода простой закраски. Также была рассмотрена сборка исполняемого файла, взаимодействие с программой и приведены примеры её работы.

## 4 Исследовательский раздел

В данном разделе будет произведено исследование разработанной программы с целью определения её характеристик. Будет измерена производительность реализованных алгоритмов построения CSG моделей и визуализации сцены в зависимости от количества обрабатываемых элементов.

### 4.1 Цель проводимых измерений

Учитывая сферу применения CSG (САПР и игры), программа, выполняющая построение сцены по данной технологии, должна обладать высокой отзывчивостью и сравнительно малыми задержками. В связи с этим необходимо понимать, как разработанные алгоритмы будут справляться с возрастающим количеством тел и полигонов, составляющих сцену.

Таким образом, целью проводимых измерений будет проведение нагрузочного тестирования и установление зависимости между количеством полигонов и временем, затраченным на их визуализацию, и зависимости между количеством CSG примитивов, участвующих в операции, и временем, затраченным на построение результирующей модели.

### 4.2 Описание проводимых измерений

Для установления зависимости между количеством полигонов и временем, затраченным на их визуализацию, будет получена CSG модель, являющаяся результатом вычитания двух шаров из куба. Затем сначала будет произведён замер небольшой части полигонов модели, после чего к ней постепенно будут добавляться отсутствующие полигоны до тех пор, пока не будет получена полная модель. Каждая такая итерация будет измерена.

Установление зависимости между количеством CSG примитивов и временем применения к ним булевой операции будет выполняться схожим образом. Сначала будет получен куб единичного размера и несколько сфер радиуса 0.5, случайных сдвинутых в пределах кубической области размером 2 на 2 на 2. Затем будет замерено применение к кубу и первой сфере операции объединения, после чего к операндам будет добавлена следующая сфера. Это



будет выполняться до тех пор, пока все примитивы не станут операндами. Затем аналогичная операция будет повторена для вычитания и пересечения.

Время будет замеряться в наносекундах, передача буфера кадра графическому API учитываться не будет.

### 4.3 Параметры оборудования

Для проведения исследования производительности программы использовалось оборудование со следующими характеристиками:

- 1) процессор – Intel Core i5-9400F;
- 2) объём оперативной памяти – 16 Гб;
- 3) разрешение экрана –  $1920 \times 1080$  пикселей.

### 4.4 Инструменты измерения времени работы

Для измерения времени, затраченного на выполнение измеряемых участков кода, будет использоваться фреймворк JMH [10], созданный для проведения бенчмарков на языке Java.

### 4.5 Результаты проведённых измерений

Полученные результаты проведённых измерений представлены в графическом формате на рисунках 12, 13, 14 и 15.

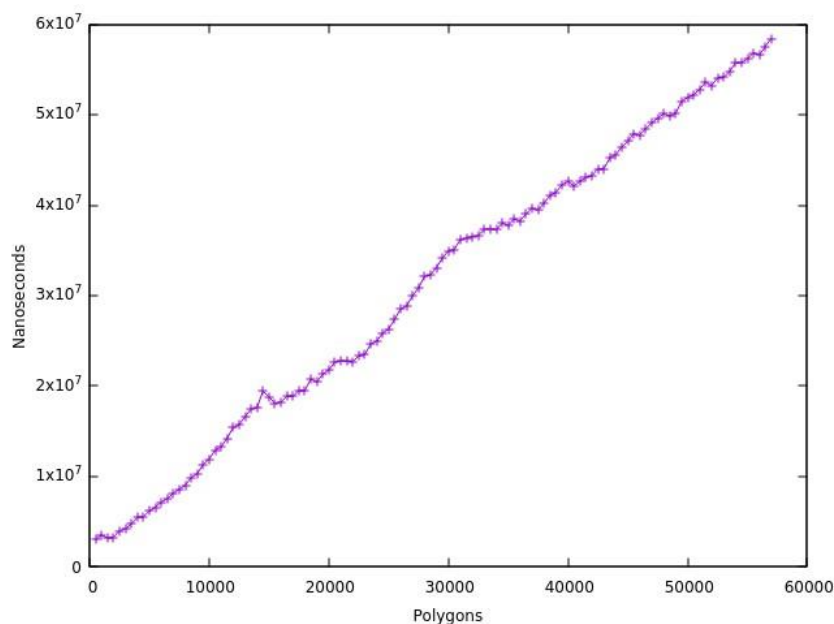


Рисунок 12. Затраты времени на получение буфера кадра

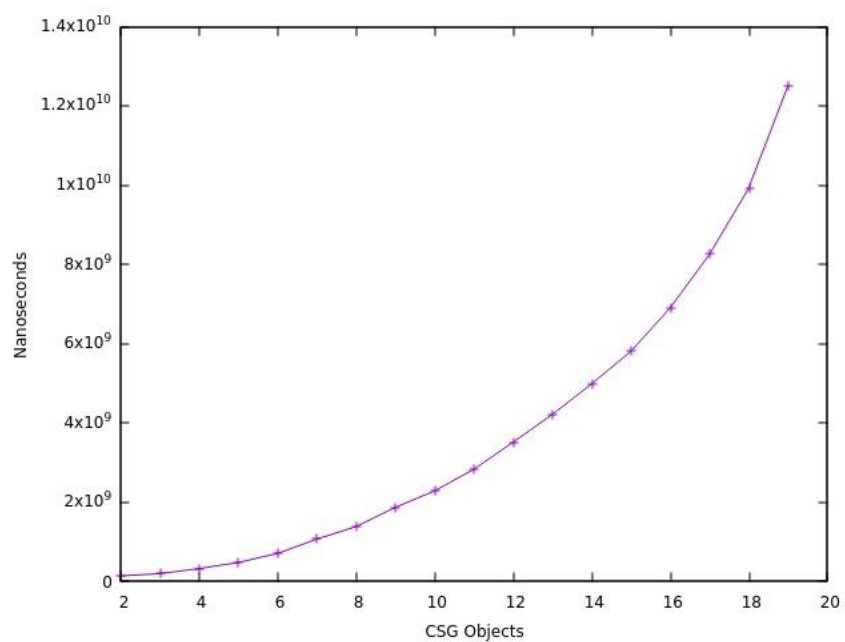


Рисунок 13. Затраты времени на построение CSG модели путём объединения

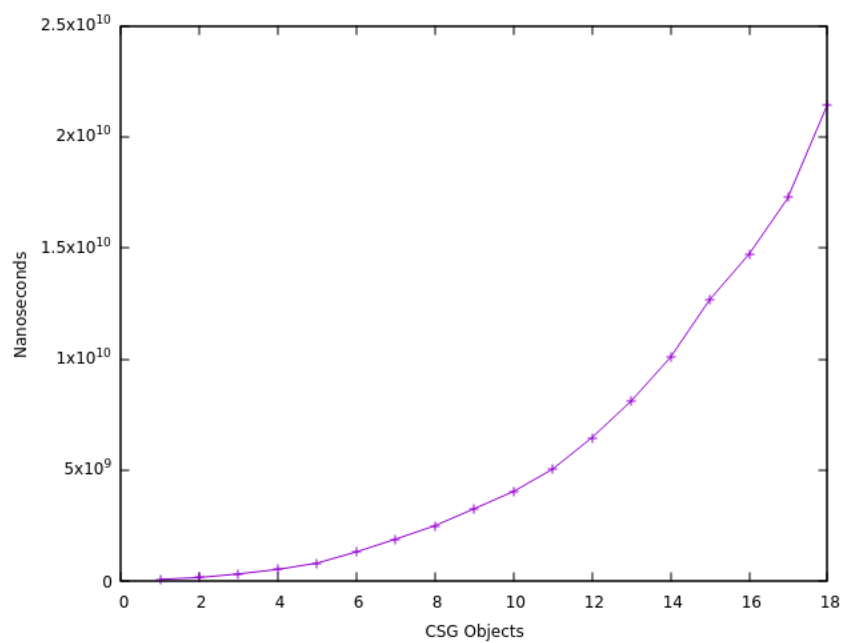


Рисунок 14. Затраты времени на построение CSG модели путём вычитания

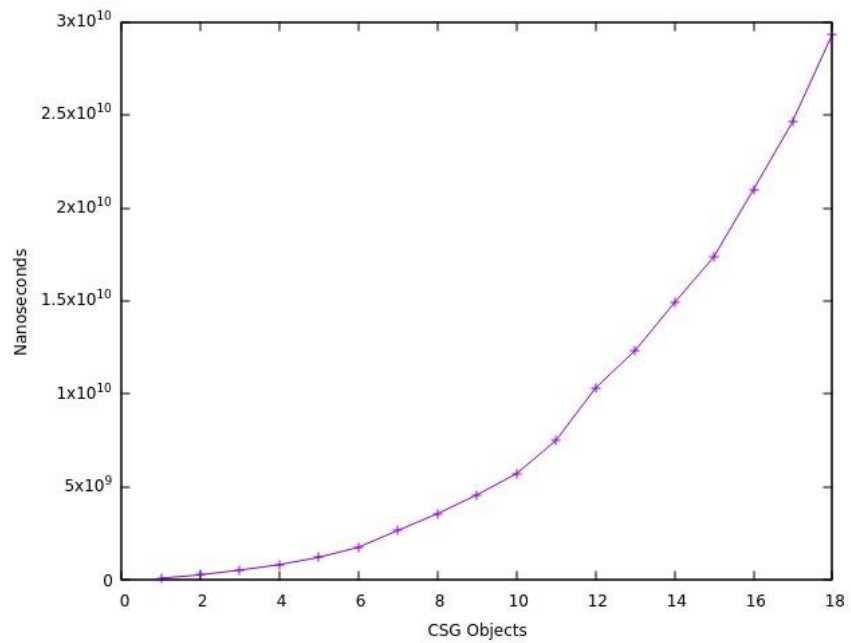


Рисунок 15. Затраты времени на построение CSG модели путём пересечения

#### 4.6 Вывод из исследовательского раздела

В данном разделе было произведено исследование характеристик разработанной программы. Были определены зависимость между количеством полигонов и временем, затраченным на их отрисовку, и зависимость между числом CSG объектов и временем, затраченным на получение результирующей модели, для каждой из реализованных булевых операций.

Измерения показали, что трудоёмкость процесса отрисовки сцены растёт линейно, а трудоёмкость выполнения булевых операций экспоненциально, что является ожидаемым результатом, исходя из специфики выбранных алгоритмов.

## Заключение

В рамках курсового проекта была реализована программа, позволяющая строить и визуализировать сцены на основе простейших трёхмерных тел (куб, цилиндр, сфера, тор).

Были рассмотрены существующие алгоритмы, позволяющие производить булевы операции над CSG телами, алгоритмы удаления невидимых линий и поверхностей и алгоритмы закраски, проанализированы их плюсы, минусы и возможность применения в рамках поставленных задач. С учётом всех заявленных особенностей были разработаны структуры данных, позволяющие реализовать выбранные алгоритмы.

Разработанная программа удовлетворяет всем требованиям технического задания: обеспечивает пользователю возможность сохранять, загружать, строить и визуализировать трёхмерные сцены с использованием CSG, а также свободно перемещать камеру для их осмотра.

В ходе выполнения исследовательской части работы были получены отношения количества визуализируемых полигонов ко времени построения буфера кадра и количества CSG объектов ко времени применения к ним булевой операции. Полученные отношения совпали с ожидаемыми и были признаны достаточными для обеспечения необходимой отзывчивости программы.

Программа, разработанная в рамках курсового проекта, имеет по меньшей мере два направления дальнейшего развития:

- 1) создание компьютерных игр в низко полигональном стиле;
- 2) создание САПР, использующих CSG для визуализации объектов.

## Список использованных источников

1. Погорелов Д. А. [и др.] Булевы операции на трёхмерных моделях в компьютерной графе [Электронный ресурс] – Режим доступа: [https://alley-science.ru/domains\\_data/files/january\\_1/BULEVY%20OPERACII%20NA%20TRYoHMERNYH%20MODELYaH%20V%20KOMPYuTERNOY%20GRAFIKE.pdf](https://alley-science.ru/domains_data/files/january_1/BULEVY%20OPERACII%20NA%20TRYoHMERNYH%20MODELYaH%20V%20KOMPYuTERNOY%20GRAFIKE.pdf) (дата обращения: 16.09.2023)
2. Carr D. Computation of Potentially Visible Set for Occluded Three-Dimensional Environments [Электронный ресурс] – Режим доступа: [https://www.bc.edu/content/dam/files/schools/cas\\_sites/cs/pdf/academics/honors/04DerekCarr.pdf](https://www.bc.edu/content/dam/files/schools/cas_sites/cs/pdf/academics/honors/04DerekCarr.pdf) (дата обращения: 20.10.2023)
3. Роджерс Д. Алгоритмические основы машинной графики [Текст]. Пер. с англ. С.А.Вичеса, Г.В. Олохтоновой, П.А. Монахова. – М.: Мир, 1989. – 512 с.
4. OpenGL Perspective [Электронный ресурс] – Режим доступа: <https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/gluPerspective.xml> (дата обращения: 20.10.2023)
5. OpenGL LookAt [Электронный ресурс] – Режим доступа: <https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml> (дата обращения 20.10.2023)
6. The Java Language Specification [Электронный ресурс] – Режим доступа: <https://docs.oracle.com/javase/specs/jls/se11/html/index.html> (дата обращения 14.11.2023)
7. IntelliJ IDEA [Электронный ресурс] – Режим доступа: <https://www.jetbrains.com/ru-ru/idea/> (дата обращения 14.11.2023)
8. Java SE 11 [Электронный ресурс] – Режим доступа: <https://www.oracle.com/java/technologies/javase/jdk11-archive-downloads.html> (дата обращения 14.11.2023)
9. Gradle Build Tool [Электронный ресурс] – Режим доступа: <https://gradle.org> (дата обращения 14.11.2023)
10. JMH [Электронный ресурс] – Режим доступа: <https://openjdk.org/projects/code-tools/jmh/> (дата обращения 17.11.2023)

# Приложение А Функциональная схема работы программы

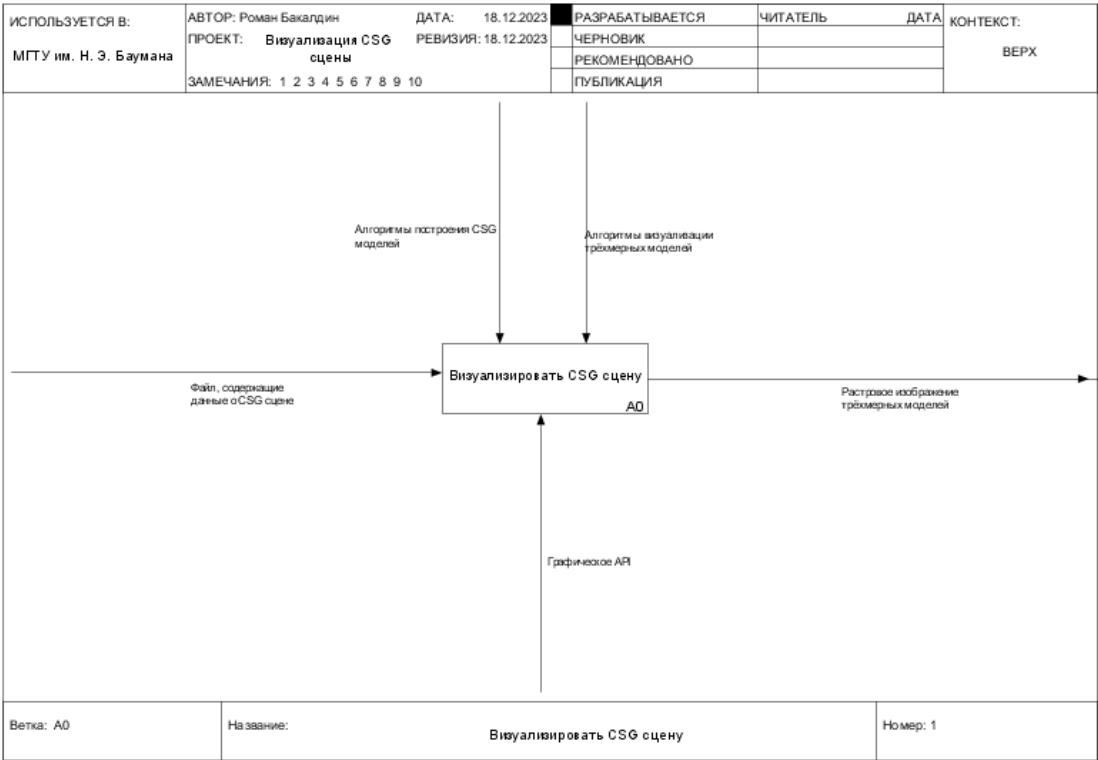


Рисунок А.1. Верхний уровень функциональной схемы

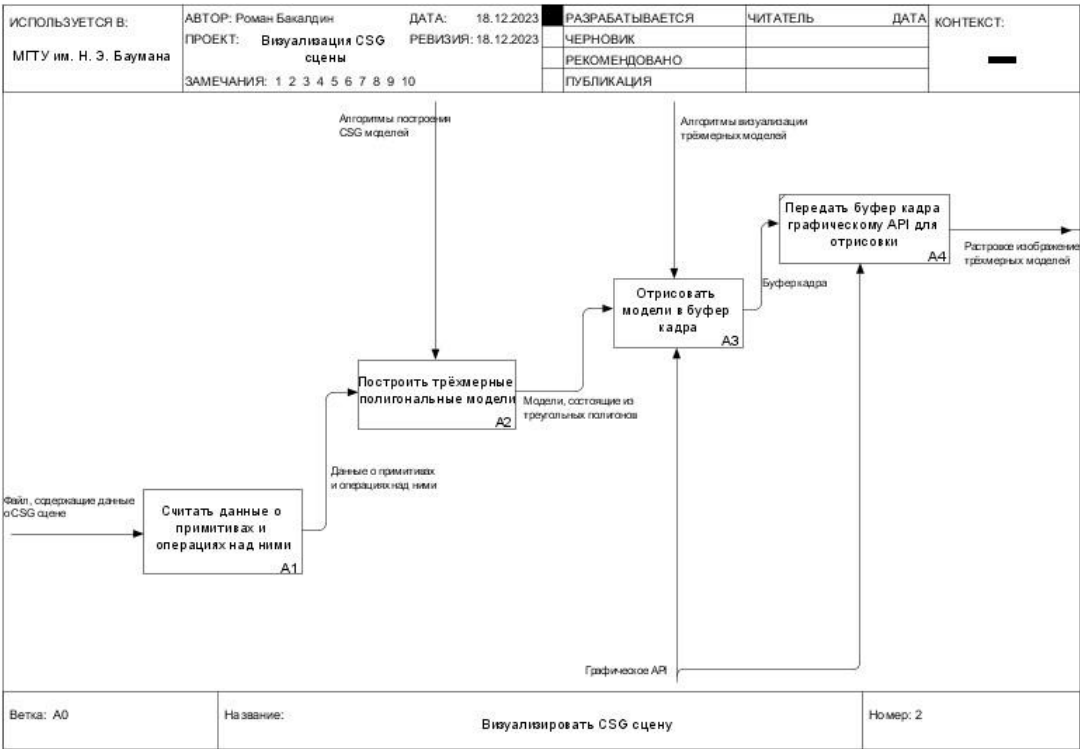


Рисунок А.2. Декомпозиция уровня A0

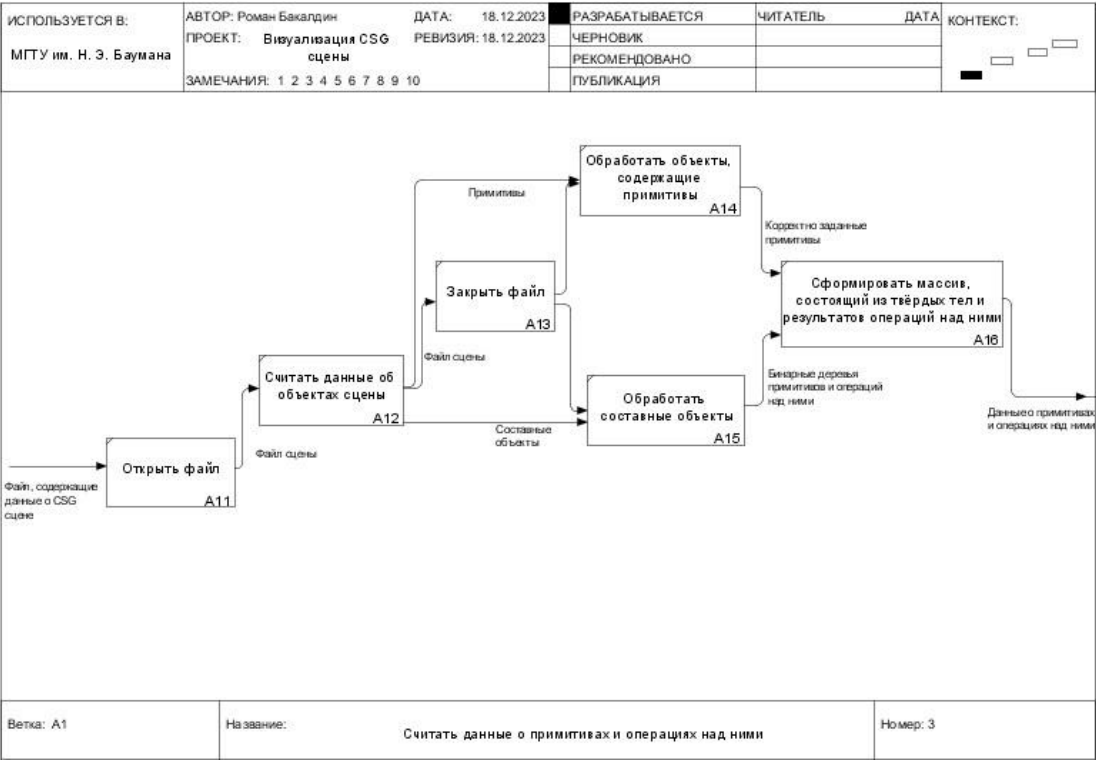


Рисунок А.3. Декомпозиция уровня A1

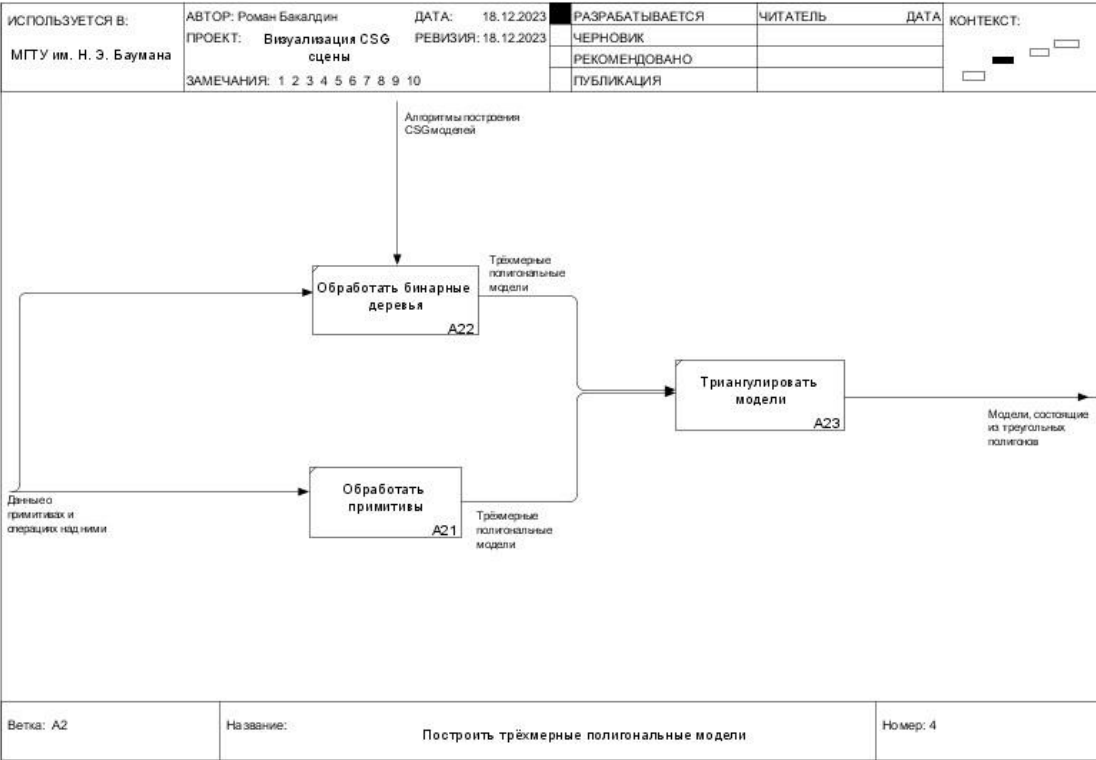


Рисунок А.4. Декомпозиция уровня A2

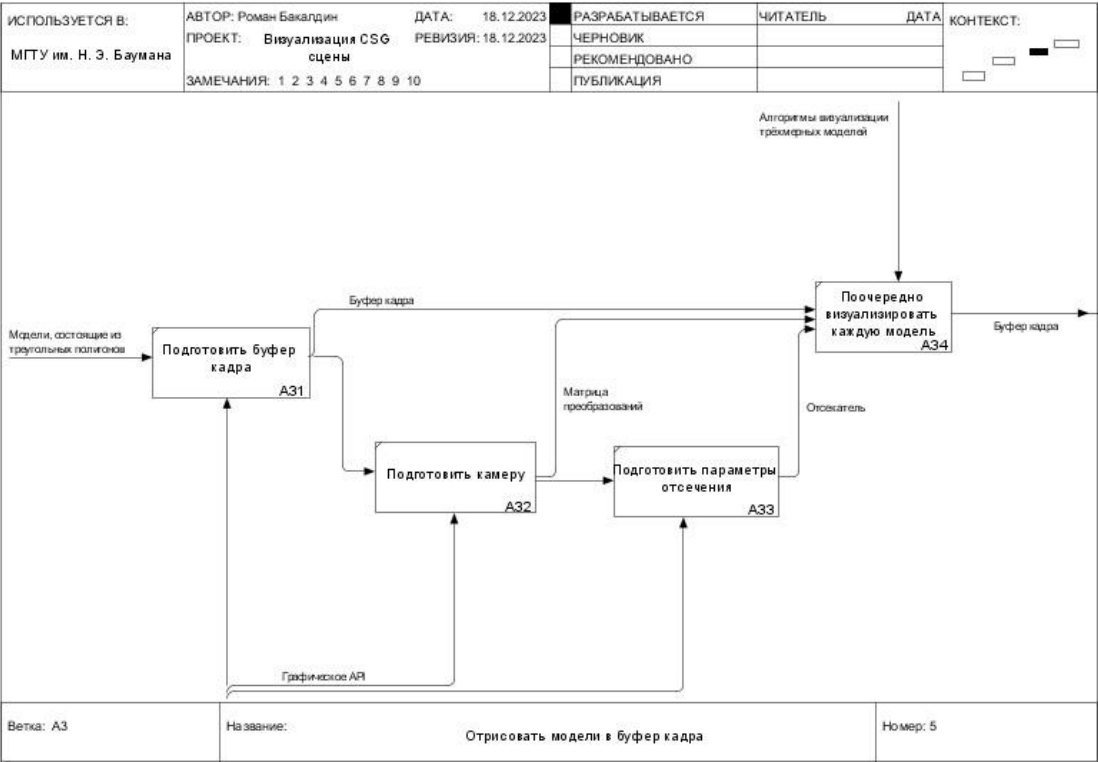


Рисунок А.5. Декомпозиция уровня А3



# Приложение Б UML диаграмма

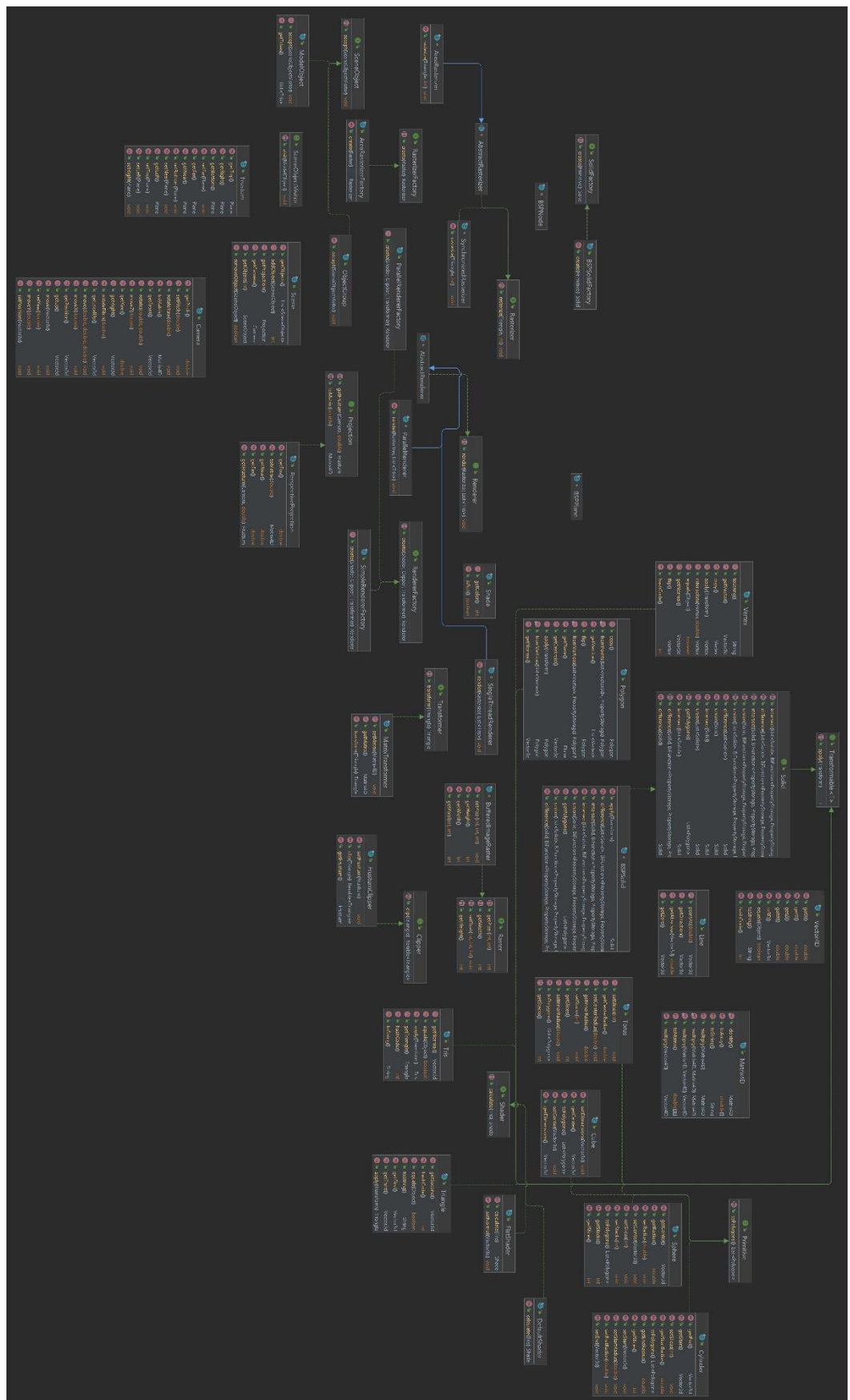


Рисунок Б.1. UML диаграмма

## Приложение В Презентация